NVAX CPU Chip Functional Specification

The NVAX CPU Chip is a high-performance, single-chip implementation of the VAX Architecture for use in low-end and mid-range systems.

Revision/Update Information: This is Revision 1.2 of this specification, which supersedes Revision 1.1 released in August 1991. The information in this specification reflects pass 2 of the NVAX CPU chip. Only the Electrical Characteristics Chapter was updated from Revision 1.1 to Revision 1.2.

Semiconductor Engineering Group
Digital Equipment Corporation, Hudson, Massachusetts

December 1991

The drawings and specifications in this document are the property of Digital Equipment Corporation and shall not be reproduced or copied or used in whole or in part as the basis for the manufacture or sale of items without written permission.

The information in this document may be changed without notice and is not a commitment by Digital Equipment Corporation. Digital Equipment Corporation is not responsible for any errors in this document.

This specification does not describe any program or product that is currently available from Digital Equipment Corporation, nor is Digital Equipment Corporation committed to implement this specification in any program or product. Digital Equipment Corporation makes no commitment that this document accurately describes any product it might ever make.

Contents

FIGURES

TABLES

Chapter 1

Introduction

The NVAX CPU is a high-performance, single-chip implementation of the VAX architecture. It is partitioned into multiple sections which cooperate to execute the VAX base instruction group. The CPU chip includes the first levels of the memory subsystem hierarchy in an on-chip virtual instruction cache and an on-chip physical instruction and data cache, as well as the controller for a large second-level cache implemented in static RAMs on the CPU module.

1.1 Scope and Organization of this Specification

This specification describes the operation of the NVAX CPU chip. It contains a description of the interface to the chip, an overview of the operation of the instruction pipeline, and extensive detail about the functional operation of each section of the chip. In addition, the specification contains discussions of error handling, chip initialization, and testability features.

1.2 Related Documents

The following documents are related to or were used in the preparation of this document:

- DEC Standard 032 VAX Architecture Standard.
- NVAX CPU Chip Design Methodology.

1.3 Terminology and Conventions

1.3.1 Numbering

All numbers are decimal unless otherwise indicated. Where there is ambiguity, numbers other than decimal are indicated with the name of the base following the number in parentheses, e.g., FF (hex).

1.3.2 UNPREDICTABLE and UNDEFINED

RESULTS specified as UNPREDICTABLE may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. Software can never depend on results specified as UNPREDICTABLE.

OPERATIONS specified as UNDEFINED may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations. The operation may vary in effect from nothing, to stopping system operation. UNDEFINED operations must not cause the processor to hang., i.e., reach a state from which there is no transition to a normal state in which the machine executes instructions.

Note the distinction between result and operation. Non-privileged software can not invoke UNDEFINED operations.

1.3.3 Ranges and Extents

Ranges are specified by a pair of numbers separated by a ".." and are inclusive, e.g., a range of integers 0..4 includes the integers 0, 1, 2, 3, and 4.

Extents are specified by a pair of numbers in angle brackets separated by a colon and are inclusive, e.g., bits <7:3> specify an extent of bits including bits 7, 6, 5, 4, and 3.

1.3.4 Must be Zero (MBZ)

Fields specified as Must Be Zero (MBZ) must never be filled by software with a non-zero value. If the processor encounters a non-zero value in a field specified as MBZ, a Reserved Operand exception occurs.

1.3.5 Should be Zero (SBZ)

Fields specified as Should Be Zero (SBZ) should be filled by software with a zero value. These fields may be used at some future time. Non-zero values in SBZ fields produce UNPREDICTABLE results.

1.3.6 Register Format Notation

This specification contains a number of figures that show the format of various registers, followed by a description of each field. In general, the fields on the register are labeled with either a name or a mnemonic. The description of each field includes the name or mnemonic, the bit extent, and the type. An example of a register is shown in Figure 1–1. Table 1–1 is an example of the description of the fields in this register.

Figure 1–1: Register Format Example

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1  0  0  0  0  0  0  0|       FAULT_CMD       | x  x  x  x|IE| 0  0  0  0  0  0  0  0|  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                                                         |  |  |
                                                                                 TRAP ---+  |  |
                                                                                 INTERRUPT -+  |
                                                                                 BUS_ERROR ----+
```

Table 1–1: Register Field Description Example

Name	Extent	Type	Description
BUS_ERROR	0	WC,0	The BUS_ERROR bit is set when a bus error is detected.
INTERRUPT	1	WC,0	The INTERRUPT bit is set when an error that is reported as an interrupt is detected.
TRAP	2	WC,0	The TRAP bit is set when an error that is reported as a trap is detected.
IE	11	RW,0	The IE bit enables error reporting interrupts. When IE is 0, interrupts are disabled. When IE is a 1, interrupts are enabled.
FAULT_CMD	23:16	RO	The FAULT_CMD field latches the command that was in progress when an error is detected.

The "Type" column in the field description includes both the actual type of the field, and an optional initialized value, separated from the type by a comma. The type denotes the functional operation of the field, and may be one of the values shown in Table 1–2. If present, the initialized value indicates that the field is initialized by hardware or microcode to the specified value at powerup. If the initialized value is not present, the field is not initialized at powerup.

Table 1–2: Register Field Type Notation

Notation	Description
RW	A read-write bit or field. The value may be read and written by software, microcode, or hardware.
RO	A read-only bit or field. The value may be read by software, microcode, or hardware. It is written by hardware; software or microcode writes are ignored.
WO	A write-only bit or field. The value may be written by software or microcode. It is read by hardware and reads by software or microcode return an UNPREDICTABLE result.
WZ	A write-only bit or field. The value may be written by software or microcode. It is read by hardware and reads by software or microcode return a 0.
WC	A write-one-to-clear bit. The value may be read by software or microcode. Software or microcode writes of a 1 cause the bit to be cleared by hardware. Software or microcode writes of a 0 do not modify the state of the bit.
RC	A read-to-clear field. The value is written by hardware and remains unchanged until read. The value may be read by software or microcode, at which point, hardware may write a new value into the field.

In addition to named fields in registers, other bits of the register may be labeled with one of the three symbols listed in Table 1–3. These symbols denote the type of the unnamed fields in the register.

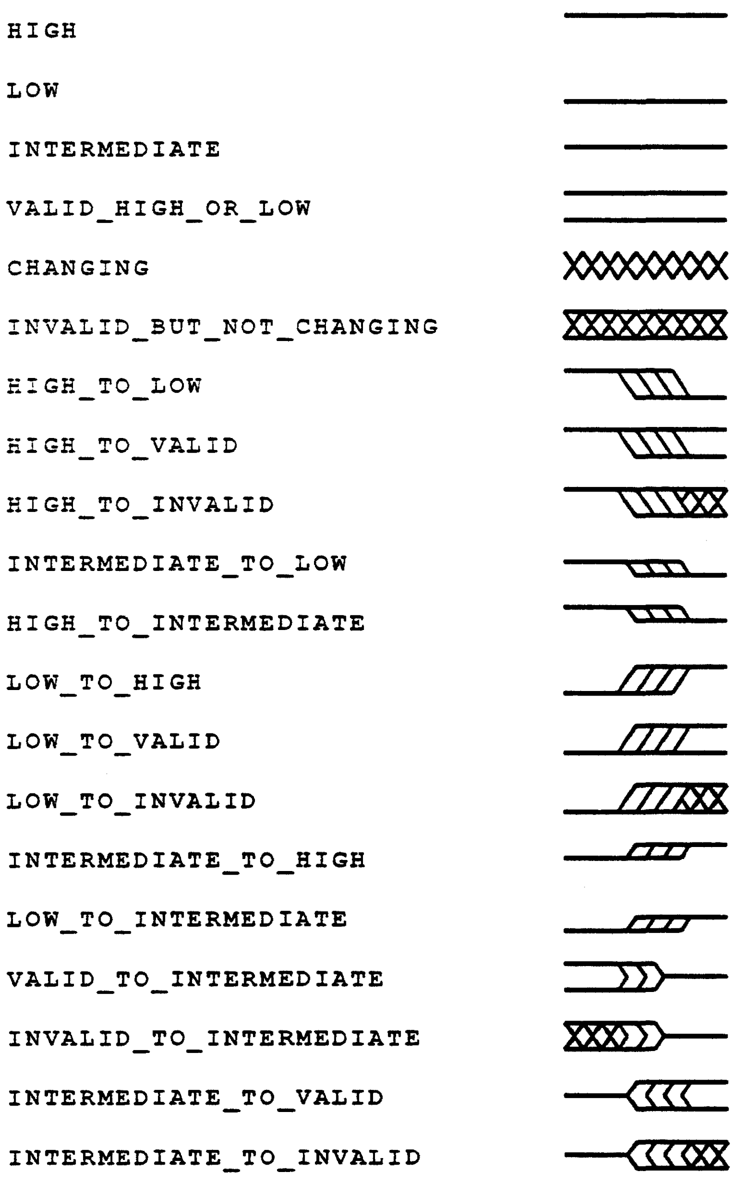
Table 1–3: Register Field Notation

Notation	Description
0	A "0" in a bit position denotes a register bit that is read as a 0 and ignored on write.
1	A "1" in a bit position denotes a register bit that is read as a 1 and ignored on write.
x	An "x" in a bit position denotes a register bit that does not exist in hardware. The value is UNPREDICTABLE when read, and ignored on write.

1.3.7 Timing Diagram Notation

This specification contains a number of timing diagrams that show the timing of various signals, including NDAL signals. The notation used in these timing diagrams is shown in Figure 1–2.

Figure 1–2: Timing Diagram Notation



1.4 Revision History

Table 1–4: Revision History

Who	When	Description of change
Mike Uhler	06-Mar-1989	Release for external review.
Mike Uhler	15-Dec-1989	Update for second-pass release.

Chapter 2

Architectural Summary

2.1 Overview

This chapter provides a summary of the VAX architectural features of the NVAX CPU Chip. It is not intended as a complete reference but rather to give an overview of the user-visible features. For a complete description of the architecture, consult the *VAX Architecture Standard* (DEC Standard 032).

2.2 Visible State

The visible state of the processor consists of memory, both virtual and physical, the general registers, the processor status longword (PSL), and the privileged internal processor registers (IPRs).

2.2.1 Virtual Address Space

The virtual address space is four gigabytes (2**32), separated into three accessable regions (P0, P1, and S0) and one reserved page , as shown in Figure 2–1.

Figure 2–1: Virtual Address Space Layout

```
          +--------------------------+
00000000  |                          |  length of P0 Region in
          |                          |  pages (P0LR)
          |  P0     -----------------|
          |  Region          |       |
3FFFFFFF  |                  V       |  P0 Region growth direction
          +--------------------------+
40000000  |                  ^       |  P1 Region growth direction
          |                  |       |
          |  P1     -----------------|
          |  Region                  |  length of P1 Region in
7FFFFFFF  |                          |  pages (2**21-P1LR)
          +--------------------------+
80000000  |                          |  length of System Region
          |                          |  in pages (SLR)
          |  System ---------------- |
          |  Region          |       |
          |                  |       |  System Region growth
          |                  |       |  direction
          |                  |       |
          |                  |       |
FFFFFDFF  |                  V       |
          +--------------------------+
FFFFFE00  |  Reserved                |
FFFFFFFF  |  Page                    |
          +--------------------------+
```

NOTE

NVAX CPU chips at revision 1 implement the original VAX memory management architecture in which any reference to a virtual address above BFFFFFFF (hex) causes a length violation. NVAX CPU chips at revision 2 or later implement the extended S0 space addressing described above.

2.2.2 Physical Address Space

The NVAX CPU naturally generates 32-bit physical addresses. This corresponds to a four gigabyte physical address space as shown in Figure 2–2. Memory space occupies the first seven-eighths (3.5GB) of the physical address space. I/O space occupies the last one-eighth (512MB) of the physical address space and can be distinguished from memory space by the fact that bits <31:29> of the physical address are all ones.

Figure 2–2: 32-bit Physical Address Space Layout

```
            +--------------------------+
00000000    |                          |
            |                          |
            |                          |
            |                          |
            |                          |
            +-                        -+
            |          Memory          |
            |          Space           |
            |                          |
            |                          |
            |                          | 3.5 GB
            +-                        -+
            |                          |
            |                          |
            |                          |
            |                          |
            |                          |
            +-                        -+
            |                          |
DFFFFFFF    |                          |
            +--------------------------+
E0000000    |           I/O            | 512 MB
FFFFFFFF    |          Space           |
            +--------------------------+
```

In addition to the natural 32-bit physical address, the CPU may be configured to generate 30-bit physical addresses. In this mode, only 512MB of memory space can be referenced, as shown in Figure 2–3.

Figure 2–3: 30-bit Physical Address Space Layout

```
            +--------------------------+
00000000    |          Memory          | 512 MB
1FFFFFFF    |          Space           |
            +--------------------------+
20000000    |                          |
            |                          |
            +-                        -+
            |                          |
            |                          |
            |                          |
            |                          |
            |                          |
            +-     Inaccessable       -+ 3.0 GB
            |         Region           |
            |                          |
            |                          |
            |                          |
            |                          |
            +-                        -+
            |                          |
DFFFFFFF    |                          |
            +--------------------------+
E0000000    |           I/O            | 512 MB
FFFFFFFF    |          Space           |
            +--------------------------+
```

The translation from 30-bit addresses to 32-bit addresses is accomplished by sign-extending PA<29> to PA<31:30>. In this mode, the programmer sees a 1GB address space, split evenly between memory and I/O space, which is mapped to the actual 32-bit physical address space as shown in Table 2–1. Unless explicitly stated otherwise, addresses that are given in the remainder of this specification are the full 32-bit addresses (which, of course, may have been generated from a 30-bit program address via the mapping shown).

Table 2–1: 30-bit Mapping of Program Addresses to 32-bit Hardware Addresses

Program Address	Hardware Address
00000000..1FFFFFFF	00000000..1FFFFFFF
20000000..3FFFFFFF	E0000000..FFFFFFFF

2.2.2.1 Physical Address Control Registers

During powerup, microcode configures the CPU to generate 30-bit physical addresses. Console firmware may then reconfigure the CPU and optional vector unit to generate either 30-bit or 32-bit physical addresses by writing to the MODE bit in the PAMODE register. The PAMODE register is shown in Figure 2–4.

Figure 2–4: IPR E7 (hex), PAMODE

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0|  | :PAMODE
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                                                              |
                                                                                      MODE --+
```

The PAMODE register also determines how PTEs are to be interpreted. In 30-bit mode, PTEs are interpreted in 21-bit PFN format. In 32-bit mode, PTEs are interpreted in 25-bit PFN format (although the two upper bits of the PFN field are ignored). The different PTE formats are described in Section 2.6.4.

The PAMODE register is described in more detail in Chapter 12.

2.2.3 Registers

There are 16 32-bit General Purpose Registers (GPRs). The format is shown in Figure 2–5, and the use of each GPR is shown in Table 2–2.

Figure 2–5: General Purpose Registers

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                                                                               | :Rn
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Table 2–2: General Purpose Register Usage

GPR	Synonym	Use
R0–R11		General Purpose
R12	AP	Argument Pointer
R13	FP	Frame Pointer
R14	SP	Stack Pointer
R15	PC	Program Counter

The Processor Status Longword (PSL) is a 32-bit register which contains processor state. The PSL format is shown in Figure 2–6, and the fields of the PSL are shown in Table 2–3.

Figure 2–6: Processor Status Longword Fields

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |MB|FP|  | CUR | PRV |MB|              |                       |  |  |  |  |  |  |  |  |
|CM|TP|VM|Z |D |IS| MOD | MOD |Z |      IPL     |          MBZ          |DV|FU|IV| T| N| Z| V| C| :PSL
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Table 2–3: Processor Status Longword

Name	Bit(s)	Description
CM	31	Compatability Mode[1]
TP	30	Trace Pending
VM	29	Virtual Machine Mode[1]
FPD	27	First Part Done
IS	26	Interrupt Stack
CUR_MOD	25:24	Current Mode
PRV_MOD	23:22	Previous Mode
IPL	20:16	Interrupt Priority Level
DV	7	Decimal Overflow Trap Enable
FU	6	Floating Underflow Fault Enable
IV	5	Integer Overflow Trap Enable
T	4	Trace Trap Enable

[1] MBZ in current implementation

Table 2–3 (Cont.): Processor Status Longword

Name	Bit(s)	Description
N	3	Negative Condition Code
Z	2	Zero Condition Code
V	1	Overflow Condition Code
C	0	Carry Condition Code

2.3 Data Types

The NVAX CPU supports nine data types: byte, word, longword, quadword, character string, variable length bit field, F_floating, D_floating, and G_floating. These are summarized in Figure 2–7.

Figure 2–7: Data Types

```
 07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+
|                       | :A
+--+--+--+--+--+--+--+--+

Data Type: Byte
Length:    8 bits
Use:       Signed or unsigned integer

 15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               | :A
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

Data Type: Word
Length:    16 bits
Use:       Signed or unsigned integer

 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                                                                               | :A
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

Data Type: Longword
Length:    32 bits
Use:       Signed or unsigned integer
```

Figure 2–7 Cont'd on next page

Figure 2–7 (Cont.): Data Types

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                                                                               | :A
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                                                                               | :A+4
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

Data Type: Quadword
Length:    64 bits
Use:       Signed integer

 07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+
|                       | :A
+--+--+--+--+--+--+--+--+
|                       | :A+1
+--+--+--+--+--+--+--+--+
            .
            .
            .
+--+--+--+--+--+--+--+--+
|                       | :A+length-1
+--+--+--+--+--+--+--+--+

Data Type: Character String
Length:    0-64K bytes
Use:       Byte string

 31                  P+S P+S-1                P P-1                                           00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                       |///////////////////////|                                               | :A
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

Data Type: Variable length bit field
Length:    0-32 bits
Use:       Bit string

 15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| s|       exponent        |      fraction      | :A
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    fraction                   | :A+2
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16

Data Type: F_floating
Length:    32 bits
Use:       Floating point

 15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| s|       exponent        |      fraction      | :A
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    fraction                   | :A+2
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    fraction                   | :A+4
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    fraction                   | :A+6
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 63 62 61 60|59 58 57 56|55 54 53 52|51 50 49 48

Data Type: D_floating
Length:    64 bits
Use:       Floating point
```

Figure 2–7 Cont'd on next page

Figure 2–7 (Cont.): Data Types

```
 15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| s|            exponent            | fraction  | :A
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                   fraction                    | :A+2
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                   fraction                    | :A+4
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                   fraction                    | :A+6
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 63 62 61 60|59 58 57 56|55 54 53 52|51 50 49 48

Data Type: G_floating
Length:    64 bits
Use:       Floating point
```

2.4 Instruction Formats and Addressing Modes

VAX instructions consist of a one- or two-byte opcode, followed by zero to six operand specifiers.

2.4.1 Opcode Formats

An opcode may be either one or two contiguous bytes. The two-byte format begins with an FD (hex) byte and is followed by a second opcode byte. The one-byte format is indicated by an opcode byte whose value is anything other than FD (hex). The one- or two-byte opcode format is shown in Figure 2–8.

Figure 2–8: Opcode Formats

```
                     07 06 05 04|03 02 01 00
                    +--+--+--+--+--+--+--+--+
One-byte opcode:    |         opcode        | :A
                    +--+--+--+--+--+--+--+--+

                     15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
                    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
Two-byte opcode:    |         opcode        |          FD           | :A
                    +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

2.4.2 Addressing Modes

An operand specifier starts with a specifier byte and may be followed by a specifier extension. Bits <3:0> of the specifier byte contain a GPR number and bits <7:4> of the specifier byte indicate the addressing mode of the specifier. If the register number in the specifier byte does not contain 15, the addressing mode is a general register addressing mode. If the register number in the specifier byte does contain 15, the addressing mode is a PC-relative addressing mode. The different addressing modes are shown graphically in Figure 2–9. General register addressing modes are listed in Table 2–4 and PC-relative addressing modes are listed in Table 2–5.

Figure 2–9: Addressing Modes

```
                      07 06 05 04|03 02 01 00
General register     +--+--+--+--+--+--+--+--+
addressing mode:     |   mode    | register  |
                     +--+--+--+--+--+--+--+--+

                      07 06 05 04|03 02 01 00
PC-relative          +--+--+--+--+--+--+--+--+
addressing mode:     |   mode    | 1  1  1  1|
                     +--+--+--+--+--+--+--+--+
```

Table 2–4: General Register Addressing Modes

			Access			
Mode	**Name**	**Assembler**	**r m w a v**	**PC**	**SP**	**Indexable?**
0-3	literal	S^#literal	y f f f f	x	x	f
4	index	i[Rx]	y y y y y	u	y	f
5	register	Rn	y y y f y	u	uq	f
6	register deferred	(Rn)	y y y y y	u	y	y
7	autodecrement	-(Rn)	y y y y y	u	y	ux
8	autoincrement	(Rn)+	y y y y y	p	y	ux
9	autoincrement deferred	@(Rn)+	y y y y y	p	y	ux
A	byte displacement	B^d(Rn)	y y y y y	p	y	y
B	byte displacement deferred	@B^d(Rn)	y y y y y	p	y	y
C	word displacement	W^d(Rn)	y y y y y	p	y	y
D	word displacement deferred	@W^d(Rn)	y y y y y	p	y	y
E	longword displacement	L^d(Rn)	y y y y y	p	y	y
F	longword displacement deferred	@L^d(Rn)	y y y y y	p	y	y

Access Types

r = read
m = modify
w = write
a = address
v = variable bit field

Syntax

i = any indexable address mode
d = displacement
Rn = general register, n = 0 to 15
Rx = general register, n = 0 to 14

Results

y = yes, always valid address mode
f = reserved addressing mode fault
x = logically impossible
p = program counter addressing
u = unpredictable
ud = unpredictable for destination of CALLG, CALLS, JMP and JSB
uq = unpredictable for quad, D/G_floating and field if pos+size > 32
ux = unpredictable if index register = base register

Table 2–5: PC-Relative Addressing Modes

			Access			
Mode	Name	Assembler	r m w a v	PC	SP	Indexable?
8	immediate	I^#constant	y u u y ud			u
9	absolute	@#address	y y y y y			y
A	byte relative	B^address	y y y y y			y
B	byte relative deferred	@B^address	y y y y y			y
C	word relative	W^address	y y y y y			y
D	word relative deferred	@W^address	y y y y y			y
E	longword relative	L^address	y y y y y			y
F	longword relative deferred	@L^address	y y y y y			y

For notation, refer to the key in Table 2–4

2.4.3 Branch Displacements

Branch instructions contain a one- or two-byte signed branch displacement after the final specifier (if any). The branch displacement is shown in Figure 2–10.

Figure 2–10: Branch Displacements

```
                    07 06 05 04|03 02 01 00
Signed byte        +--+--+--+--+--+--+--+--+
displacement:      |      displacement     |
                   +--+--+--+--+--+--+--+--+

                    15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
Signed word        +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
displacement:      |                  displacement                 |
                   +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

2.5 Instruction Set

The NVAX CPU supports the VAX Base Instruction Group as defined in DEC Standard 032. These instructions are listed in Table 2–6.

Table 2–6: NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Integer, Arithmetic and Logical Instructions						
58	ADAWI add.rw, sum.mw	*	*	*	*	iov
80	ADDB2 add.rb, sum.mb	*	*	*	*	iov
C0	ADDL2 add.rl, sum.ml	*	*	*	*	iov
A0	ADDW2 add.rw, sum.mw	*	*	*	*	iov
81	ADDB3 add1.rb, add2.rb, sum.wb	*	*	*	*	iov
C1	ADDL3 add1.rl, add2.rl, sum.wl	*	*	*	*	iov
A1	ADDW3 add1.rw, add2.rw, sum.ww	*	*	*	*	iov
D8	ADWC add.rl, sum.ml	*	*	*	*	iov
78	ASHL cnt.rb, src.rl, dst.wl	*	*	*	0	iov
79	ASHQ cnt.rb, src.rq, dst.wq	*	*	*	0	iov
8A	BICB2 mask.rb, dst.mb	*	*	0	–	
CA	BICL2 mask.rl, dst.ml	*	*	0	–	
AA	BICW2 mask.rw, dst.mw	*	*	0	–	
8B	BICB3 mask.rb, src.rb, dst.wb	*	*	0	–	
CB	BICL3 mask.rl, src.rl, dst.wl	*	*	0	–	
AB	BICW3 mask.rw, src.rw, dst.ww	*	*	0	–	
88	BISB2 mask.rb, dst.mb	*	*	0	–	
C8	BISL2 mask.rl, dst.ml	*	*	0	–	
A8	BISW2 mask.rw, dst.mw	*	*	0	–	
89	BISB3 mask.rb, src.rb, dst.wb	*	*	0	–	
C9	BISL3 mask.rl, src.rl, dst.wl	*	*	0	–	
A9	BISW3 mask.rw, src.rw, dst.ww	*	*	0	–	
93	BITB mask.rb, src.rb	*	*	0	–	
D3	BITL mask.rl, src.rl	*	*	0	–	
B3	BITW mask.rw, src.rw	*	*	0	–	

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Integer, Arithmetic and Logical Instructions						
94	CLRB dst.wb	0	1	0	–	
D4	CLRL{=F} dst.wl	0	1	0	–	
7C	CLRQ{=D=G} dst.wq	0	1	0	–	
B4	CLRW dst.ww	0	1	0	–	
91	CMPB src1.rb, src2.rb	*	*	0	*	
D1	CMPL src1.rl, src2.rl	*	*	0	*	
B1	CMPW src1.rw, src2.rw	*	*	0	*	
98	CVTBL src.rb, dst.wl	*	*	0	0	
99	CVTBW src.rb, dst.ww	*	*	0	0	
F6	CVTLB src.rl, dst.wb	*	*	*	0	iov
F7	CVTLW src.rl, dst.ww	*	*	*	0	iov
33	CVTWB src.rw, dst.wb	*	*	*	0	iov
32	CVTWL src.rw, dst.wl	*	*	0	0	
97	DECB dif.mb	*	*	*	*	iov
D7	DECL dif.ml	*	*	*	*	iov
B7	DECW dif.mw	*	*	*	*	iov
86	DIVB2 divr.rb, quo.mb	*	*	*	0	iov, idvz
C6	DIVL2 divr.rl, quo.ml	*	*	*	0	iov, idvz
A6	DIVW2 divr.rw, quo.mw	*	*	*	0	iov, idvz
87	DIVB3 divr.rb, divd.rb, quo.wb	*	*	*	0	iov, idvz
C7	DIVL3 divr.rl, divd.rl, quo.wl	*	*	*	0	iov, idvz
A7	DIVW3 divr.rw, divd.rw, quo.ww	*	*	*	0	iov, idvz
7B	EDIV divr.rl, divd.rq, quo.wl, rem.wl	*	*	*	0	iov, idvz
7A	EMUL mulr.rl, muld.rl, add.rl, prod.wq	*	*	0	0	
96	INCB sum.mb	*	*	*	*	iov
D6	INCL sum.ml	*	*	*	*	iov

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Integer, Arithmetic and Logical Instructions						
B6	INCW sum.mw	*	*	*	*	iov
92	MCOMB src.rb, dst.wb	*	*	0	–	
D2	MCOML src.rl, dst.wl	*	*	0	–	
B2	MCOMW src.rw, dst.ww	*	*	0	–	
8E	MNEGB src.rb, dst.wb	*	*	*	*	iov
CE	MNEGL src.rl, dst.wl	*	*	*	*	iov
AE	MNEGW src.rw, dst.ww	*	*	*	*	iov
90	MOVB src.rb, dst.wb	*	*	0	–	
D0	MOVL src.rl, dst.wl	*	*	0	–	
7D	MOVQ src.rq, dst.wq	*	*	0	–	
B0	MOVW src.rw, dst.ww	*	*	0	–	
9A	MOVZBW src.rb, dst.wb	0	*	0	–	
9B	MOVZBL src.rb, dst.wl	0	*	0	–	
3C	MOVZWL src.rw, dst.wl	0	*	0	–	
84	MULB2 mulr.rb, prod.mb	*	*	*	0	iov
C4	MULL2 mulr.rl, prod.ml	*	*	*	0	iov
A4	MULW2 mulr.rw, prod.mw	*	*	*	0	iov
85	MULB3 mulr.rb, muld.rb, prod.wb	*	*	*	0	iov
C5	MULL3 mulr.rl, muld.rl, prod.wl	*	*	*	0	iov
A5	MULW3 mulr.rw, muld.rw, prod.ww	*	*	*	0	iov
DD	PUSHL src.rl, {-(SP).wl}	*	*	0	–	
9C	ROTL cnt.rb, src.rl, dst.wl	*	*	0	–	
D9	SBWC sub.rl, dif.ml	*	*	*	*	iov
82	SUBB2 sub.rb, dif.mb	*	*	*	*	iov

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Integer, Arithmetic and Logical Instructions						
C2	SUBL2 sub.rl, dif.ml	*	*	*	*	iov
A2	SUBW2 sub.rw, dif.mw	*	*	*	*	iov
83	SUBB3 sub.rb, min.rb, dif.wb	*	*	*	*	iov
C3	SUBL3 sub.rl, min.rl, dif.wl	*	*	*	*	iov
A3	SUBW3 sub.rw, min.rw, dif.ww	*	*	*	*	iov
95	TSTB src.rb	*	*	0	0	
D5	TSTL src.rl	*	*	0	0	
B5	TSTW src.rw	*	*	0	0	
8C	XORB2 mask.rb, dst.mb	*	*	0	–	
CC	XORL2 mask.rl, dst.ml	*	*	0	–	
AC	XORW2 mask.rw, dst.mw	*	*	0	–	
8D	XORB3 mask.rb, src.rb, dst.wb	*	*	0	–	
CD	XORL3 mask.rl, src.rl, dst.wl	*	*	0	–	
AD	XORW3 mask.rw, src.rw, dst.ww	*	*	0	–	
Address Instructions						
9E	MOVAB src.ab, dst.wl	*	*	0	–	
DE	MOVAL{=F} src.al, dst.wl	*	*	0	–	
7E	MOVAQ{=D=G} src.aq, dst.wl	*	*	0	–	
3E	MOVAW src.aw, dst.wl	*	*	0	–	
9F	PUSHAB src.ab, {-(SP).wl}	*	*	0	–	
DF	PUSHAL{=F} src.al, {-(SP).wl}	*	*	0	–	
7F	PUSHAQ{=D=G} src.aq, {-(SP).wl}	*	*	0	–	
3F	PUSHAW src.aw, {-(SP).wl}	*	*	0	–	
Variable-Length Bit Field Instructions						
EC	CMPV pos.rl, size.rb, base.vb, {field.rv}, src.rl	*	*	0	*	rsv
ED	CMPZV pos.rl, size.rb, base.vb, {field.rv}, src.rl	*	*	0	*	rsv

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Variable-Length Bit Field Instructions						
EE	EXTV pos.rl, size.rb, base.vb, {field.rv}, dst.wl	*	*	0	–	rsv
EF	EXTZV pos.rl, size.rb, base.vb, {field.rv}, dst.wl	*	*	0	–	rsv
F0	INSV src.rl, pos.rl, size.rb, base.vb, {field.wv}	–	–	–	–	rsv
EB	FFC startpos.rl, size.rb, base.vb, {field.rv}, findpos.wl	0	*	0	0	rsv
EA	FFS startpos.rl, size.rb, base.vb, {field.rv}, findpos.wl	0	*	0	0	rsv
Control Instructions						
9D	ACBB limit.rb, add.rb, index.mb, displ.bw	*	*	*	–	iov
F1	ACBL limit.rl, add.rl, index.ml, displ.bw	*	*	*	–	iov
3D	ACBW limit.rw, add.rw, index.mw, displ.bw	*	*	*	–	iov
F3	AOBLEQ limit.rl, index.ml, displ.bb	*	*	*	–	iov
F2	AOBLSS limit.rl, index.ml, displ.bb	*	*	*	–	iov
1E	BCC{=BGEQU} displ.bb	–	–	–	–	
1F	BCS{=BLSSU} displ.bb	–	–	–	–	
13	BEQL{=BEQLU} displ.bb	–	–	–	–	
18	BGEQ displ.bb	–	–	–	–	
14	BGTR displ.bb	–	–	–	–	
1A	BGTRU displ.bb	–	–	–	–	
15	BLEQ displ.bb	–	–	–	–	
1B	BLEQU displ.bb	–	–	–	–	
19	BLSS displ.bb	–	–	–	–	
12	BNEQ{=BNEQU} displ.bb	–	–	–	–	
1C	BVC displ.bb	–	–	–	–	
1D	BVS displ.bb	–	–	–	–	
E1	BBC pos.rl, base.vb, displ.bb, {field.rv}	–	–	–	–	rsv
E0	BBS pos.rl, base.vb, displ.bb, {field.rv}	–	–	–	–	rsv

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Control Instructions						
E5	BBCC pos.rl, base.vb, displ.bb, {field.mv}	–	–	–	–	rsv
E3	BBCS pos.rl, base.vb, displ.bb, {field.mv}	–	–	–	–	rsv
E4	BBSC pos.rl, base.vb, displ.bb, {field.mv}	–	–	–	–	rsv
E2	BBSS pos.rl, base.vb, displ.bb, {field.mv}	–	–	–	–	rsv
E7	BBCCI pos.rl, base.vb, displ.bb, {field.mv}	–	–	–	–	rsv
E6	BBSSI pos.rl, base.vb, displ.bb, {field.mv}	–	–	–	–	rsv
E9	BLBC src.rl, displ.bb	–	–	–	–	
E8	BLBS src.rl, displ.bb	–	–	–	–	
11	BRB displ.bb	–	–	–	–	
31	BRW displ.bw	–	–	–	–	
10	BSBB displ.bb, {-(SP).wl}	–	–	–	–	
30	BSBW displ.bw, {-(SP).wl}	–	–	–	–	
8F	CASEB selector.rb, base.rb, limit.rb, displ.bw-list	*	*	0	*	
CF	CASEL selector.rl, base.rl, limit.rl, displ.bw-list	*	*	0	*	
AF	CASEW selector.rw, base.rw, limit.rw, displ.bw-list	*	*	0	*	
17	JMP dst.ab	–	–	–	–	
16	JSB dst.ab, {-(SP).wl}	–	–	–	–	
05	RSB {(SP)+.rl}	–	–	–	–	
F4	SOBGEQ index.ml, displ.bb	*	*	*	–	iov
F5	SOBGTR index.ml, displ.bb	*	*	*	–	iov

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Procedure Call Instructions						
FA	CALLG arglist.ab, dst.ab, {-(SP).w*}	0	0	0	0	rsv
FB	CALLS numarg.rl, dst.ab, {-(SP).w*}	0	0	0	0	rsv
04	RET {(SP)+.r*}	*	*	*	*	rsv
Miscellaneous Instructions						
B9	BICPSW mask.rw	*	*	*	*	rsv
B8	BISPSW mask.rw	*	*	*	*	rsv
03	BPT {-(KSP).w*}	0	0	0	0	
00	HALT {-(KSP).w*}	–	–	–	–	prv
0A	INDEX subscript.rl, low.rl, high.rl, size.rl, indexin.rl, indexout.wl	*	*	0	0	sub
DC	MOVPSL dst.wl	–	–	–	–	
01	NOP	–	–	–	–	
BA	POPR mask.rw, {(SP)+.r*}	–	–	–	–	
BB	PUSHR mask.rw, {-(SP).w*}	–	–	–	–	
FC	XFC {unspecified operands}	0	0	0	0	
Queue Instructions						
5C	INSQHI entry.ab, header.aq	0	*	0	*	rsv
5D	INSQTI entry.ab, header.aq	0	*	0	*	rsv
0E	INSQUE entry.ab, pred.ab	*	*	0	*	
5E	REMQHI header.aq, addr.wl	0	*	*	*	rsv
5F	REMQTI header.aq, addr.wl	0	*	*	*	rsv
0F	REMQUE entry.ab, addr.wl	*	*	*	*	

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Operating System Support Instructions						
BD	CHME param.rw, {-(ySP).w*}	0	0	0	0	
BC	CHMK param.rw, {-(ySP).w*}	0	0	0	0	
BE	CHMS param.rw, {-(ySP).w*}	0	0	0	0	
BF	CHMU param.rw, {-(ySP).w*}	0	0	0	0	
06	LDPCTX {PCB.r*, -(KSP).w*}	–	–	–	–	rsv, prv
DB	MFPR procreg.rl, dst.wl	*	*	0	–	rsv, prv
DA	MTPR src.rl, procreg.rl	*	*	0	–	rsv, prv
0C	PROBER mode.rb, len.rw, base.ab	0	*	0	–	
0D	PROBEW mode.rb, len.rw, base.ab	0	*	0	–	
02	REI {(SP)+.r*}	*	*	*	*	rsv
07	SVPCTX {(SP)+.r*, PCB.w*}	–	–	–	–	prv
Character String Instructions						
29	CMPC3 len.rw, src1addr.ab, src2addr.ab	*	*	0	*	
2D	CMPC5 src1len.rw, src1addr.ab, fill.rb,src2len.rw, src2addr.ab	*	*	0	*	
3A	LOCC char.rb, len.rw, addr.ab	0	*	0	0	
28	MOVC3 len.rw, srcaddr.ab, dstaddr.ab, {R0-5.wl}	0	1	0	0	
2C	MOVC5 srclen.rw, srcaddr.ab, fill.rb, dstlen.rw, dstaddr.ab,{R0-5.wl}	*	*	0	*	
2A	SCANC len.rw, addr.ab, tbladdr.ab, mask.rb	0	*	0	0	

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Character String Instructions						
3B	SKPC char.rb, len.rw, addr.ab	0	*	0	0	
2B	SPANC len.rw, addr.ab, tbladdr.ab, mask.rb	0	*	0	0	
Floating Point Instructions						
60	ADDD2 add.rd, sum.md	*	*	0	0	rsv, fov, fuv
40	ADDF2 add.rf, sum.mf	*	*	0	0	rsv, fov, fuv
40FD	ADDG2 add.rg, sum.mg	*	*	0	0	rsv, fov, fuv
61	ADDD3 add1.rd, add2.rd, sum.wd	*	*	0	0	rsv, fov, fuv
41	ADDF3 add1.rf, add2.rf, sum.wf	*	*	0	0	rsv, fov, fuv
41FD	ADDG3 add1.rg, add2.rg, sum.wg	*	*	0	0	rsv, fov, fuv
71	CMPD src1.rd, src2.rd	*	*	0	0	rsv
51	CMPF src1.rf, src2.rf	*	*	0	0	rsv
51FD	CMPG src1.rg, src2.rg	*	*	0	0	rsv
6C	CVTBD src.rb, dst.wd	*	*	0	0	
4C	CVTBF src.rb, dst.wf	*	*	0	0	
4CFD	CVTBG src.rb, dst.wg	*	*	0	0	
68	CVTDB src.rd, dst.wb	*	*	*	0	rsv, iov
76	CVTDF src.rd, dst.wf	*	*	0	0	rsv, fov
6A	CVTDL src.rd, dst.wl	*	*	*	0	rsv, iov
69	CVTDW src.rd, dst.ww	*	*	*	0	rsv, iov
48	CVTFB src.rf, dst.wb	*	*	*	0	rsv, iov
56	CVTFD src.rf, dst.wd	*	*	0	0	rsv
99FD	CVTFG src.rf, dst.wg	*	*	0	0	rsv
4A	CVTFL src.rf, dst.wl	*	*	*	0	rsv, iov
49	CVTFW src.rf, dst.ww	*	*	*	0	rsv, iov
48FD	CVTGB src.rg, dst.wb	*	*	*	0	rsv, iov
33FD	CVTGF src.rg, dst.wf	*	*	0	0	rsv, fov, fuv
4AFD	CVTGL src.rg, dst.wl	*	*	*	0	rsv, iov

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Floating Point Instructions						
49FD	CVTGW src.rg, dst.ww	*	*	*	0	rsv, iov
6E	CVTLD src.rl, dst.wd	*	*	0	0	
4E	CVTLF src.rl, dst.wf	*	*	0	0	
4EFD	CVTLG src.rl, dst.wg	*	*	0	0	
6D	CVTWD src.rw, dst.wd	*	*	0	0	
4D	CVTWF src.rw, dst.wf	*	*	0	0	
4DFD	CVTWG src.rw, dst.wg	*	*	0	0	
6B	CVTRDL src.rd, dst.wl	*	*	*	0	rsv, iov
4B	CVTRFL src.rf, dst.wl	*	*	*	0	rsv, iov
4BFD	CVTRGL src.rg, dst.wl	*	*	*	0	rsv, iov
66	DIVD2 divr.rd, quo.md	*	*	0	0	rsv, fov, fuv, fdvz
46	DIVF2 divr.rf, quo.mf	*	*	0	0	rsv, fov, fuv, fdvz
46FD	DIVG2 divr.rg, quo.mg	*	*	0	0	rsv, fov, fuv, fdvz
67	DIVD3 divr.rd, divd.rd, quo.wd	*	*	0	0	rsv, fov, fuv, fdvz
47	DIVF3 divr.rf, divd.rf, quo.wf	*	*	0	0	rsv, fov, fuv, fdvz
47FD	DIVG3 divr.rg, divd.rg, quo.wg	*	*	0	0	rsv, fov, fuv, fdvz
72	MNEGD src.rd, dst.wd	*	*	0	0	rsv
52	MNEGF src.rf, dst.wf	*	*	0	0	rsv
52FD	MNEGG src.rg, dst.wg	*	*	0	0	rsv
70	MOVD src.rd, dst.wd	*	*	0	-	rsv
50	MOVF src.rf, dst.wf	*	*	0	-	rsv
50FD	MOVG src.rg, dst.wg	*	*	0	-	rsv
64	MULD2 mulr.rd, prod.md	*	*	0	0	rsv, fov, fuv
44	MULF2 mulr.rf, prod.mf	*	*	0	0	rsv, fov, fuv
44FD	MULG2 mulr.rg, prod.mg	*	*	0	0	rsv, fov, fuv
65	MULD3 mulr.rd, muld.rd, prod.wd	*	*	0	0	rsv, fov, fuv
45	MULF3 mulr.rf, muld.rf, prod.wf	*	*	0	0	rsv, fov, fuv
45FD	MULG3 mulr.rg, muld.rg, prod.wg	*	*	0	0	rsv, fov, fuv

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Floating Point Instructions						
62	SUBD2 sub.rd, dif.md	*	*	0	0	rsv, fov, fuv
42	SUBF2 sub.rf, dif.mf	*	*	0	0	rsv, fov, fuv
42FD	SUBG2 sub.rg, dif.mg	*	*	0	0	rsv, fov, fuv
63	SUBD3 sub.rd, min.rd, dif.wd	*	*	0	0	rsv, fov, fuv
43	SUBF3 sub.rf, min.rf, dif.wf	*	*	0	0	rsv, fov, fuv
43FD	SUBG3 sub.rg, min.rg, dif.wg	*	*	0	0	rsv, fov, fuv
73	TSTD src.rd	*	*	0	0	rsv
53	TSTF src.rf	*	*	0	0	rsv
53FD	TSTG src.rg	*	*	0	0	rsv
Microcode-Assisted Emulated Instructions						
20	ADDP4 addlen.rw, addaddr.ab, sumlen.rw, sumaddr.ab	*	*	*	0	rsv, dov
21	ADDP6 add1len.rw, add1addr.ab, add2len.rw, add2addr.ab, sumlen.rw, sumaddr.ab	*	*	*	0	rsv, dov
F8	ASHP cnt.rb, srclen.rw, srcaddr.ab, round.rb, dstlen.rw, dstaddr.ab	*	*	*	0	rsv, dov
35	CMPP3 len.rw, src1addr.ab, src2addr.ab	*	*	0	0	
37	CMPP4 src1len.rw, src1addr.ab, src2len.rw, src2addr.ab	*	*	0	0	
0B	CRC tbl.ab, inicrc.rl, strlen.rw, stream.ab	*	*	0	0	
F9	CVTLP src.rl, dstlen.rw, dstaddr.ab	*	*	*	0	rsv, dov
36	CVTPL srclen.rw, srcaddr.ab, dst.wl	*	*	*	0	rsv, iov
08	CVTPS srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab	*	*	*	0	rsv, dov
09	CVTSP srclen.rw, srcaddr.ab, dstlen.rw, dstaddr.ab	*	*	*	0	rsv, dov

Table 2–6 (Cont.): NVAX Instruction Set

Opcode	Instruction	N	Z	V	C	Exceptions
Microcode-Assisted Emulated Instructions						
24	CVTPT srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw, dstaddr.ab	*	*	*	0	rsv, dov
26	CVTTP srclen.rw, srcaddr.ab, tbladdr.ab, dstlen.rw, dstaddr.ab	*	*	*	0	rsv, dov
27	DIVP divrlen.rw, divraddr.ab, divdlen.rw, divdaddr.ab, quolen.rw, quoaddr.ab	*	*	*	0	rsv, dov, ddvz
38	EDITPC srclen.rw, srcaddr.ab, pattern.ab, dstaddr.ab	*	*	*	*	rsv, dov
39	MATCHC objlen.rw, objaddr.ab, srclen.rw, srcaddr.ab	0	*	0	0	
34	MOVP len.rw, srcaddr.ab, dstaddr.ab	*	*	0	0	
2E	MOVTC srclen.rw, srcaddr.ab, fill.rb, tbladdr.ab, dstlen.rw, dstaddr.ab	*	*	0	*	
2F	MOVTUC srclen.rw, srcaddr.ab, esc.rb, tbladdr.ab, dstlen.rw, dstaddr.ab	*	*	*	*	
25	MULP mulrlen.rw, mulraddr.ab, muldlen.rw, muldaddr.ab, prodlen.rw, prodaddr.ab	*	*	*	0	rsv, dov
22	SUBP4 sublen.rw, subaddr.ab, diflen.rw, difaddr.ab	*	*	*	0	rsv, dov
23	SUBP6 sublen.rw, subaddr.ab, minlen.rw, minaddr.ab, diflen.rw, difaddr.ab	*	*	*	0	rsv, dov

Table 2–6 (Cont.): NVAX Instruction Set

The notation used for operand specifiers is <name>.<access type><data type>. Implied operands (those locations that are referenced by the instruction but not specified by an operand) are denoted by curly braces {}.

Access Type

a = address operand
b = branch displacement
m = modified operand (both read and written)
r = read only operand
v = if not "Rn", same as a, otherwise R[n+1]'R[n]
w = write only operand

Data Type

b = byte
d = D_floating
f = F_floating
g = G_floating
l = longword
q = quadword
v = field (used only in implied operands)
w = word
* = multiple longwords (used only in implied operands)

Condition Codes Modification

* = conditionally set/cleared
– = not affected
0 = cleared
1 = set

Exceptions

rsv = reserved operand fault
iov = integer overflow trap
idvz = integer divide by zero trap
fov = floating overflow fault
fuv = floating underflow fault
fdvz = floating divide by zero fault
dov = decimal overflow trap
ddvz = decimal divide by zero trap
sub = subscript range trap
prv = privileged instruction fault
vec = vector unit disabled fault

2.6 Memory Management

The NVAX CPU Chip supports a four gigabyte (2**32) virtual address space, divided into two sections, system space and process space. Process space is further subdivided into the P0 region and the P1 region.

2.6.1 Memory Management Control Registers

Memory management is controlled by three processor registers: Memory Management Enable (MAPEN), Translation Buffer Invalidate Single (TBIS), and Translation Buffer Invalidate All (TBIA).

Bit <0> of the MAPEN register enables memory management if written with a 1 and disables memory management if written with a 0. The MAPEN register is shown in Figure 2–11.

Figure 2–11: IPR 38 (hex), MAPEN

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0|  | :MAPEN
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                                                             |
                                                                                      MME --+
```

The TBIS register controls translation buffer invalidation. Writing a virtual address into TBIS invalidates any entry which maps that virtual address. The TBIS format is shown in Figure 2–12.

Figure 2–12: IPR 3A (hex), TBIS

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                         Virtual Address                                        | :TBIS
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

The TBIA register also controls translation buffer invalidation. Writing a zero into TBIA invalidates the entire translation buffer. The TBIA format is shown in Figure 2–13.

Figure 2–13: IPR 39 (hex), TBIA

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0| :TBIA
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

2.6.2 System Space Address Translation

A virtual address with bit <31> = 1 is an address in the system virtual address space.

System virtual address space is mapped by the System Page Table (SPT), which is defined by the System Base Register (SBR) and the System Length Register (SLR). The SBR contains the page-aligned physical address of the the System Page Table. The SLR contains the size of the SPT in longwords, that is, the number of Page Table Entries. The Page Table Entry addressed by the System Base Register maps the first page of system virtual address space, that is, virtual byte address 80000000 (hex). These registers are shown in Figure 2–14.

With a 22-bit SLR width, $2^{22} - 1$ pages in system space may be addressed. As a result, the last page of system space (beginning at virtual address FFFFFE00 (hex)) is not addressable. As a result, this page is reserved and a reference to any address in that page will result in a length violation.

NOTE

NVAX CPU chips at revision 1 implement the original VAX memory management architecture in which any reference to a virtual address above BFFFFFFF (hex) causes a length violation. NVAX CPU chips at revision 2 or later implement the extended S0 space addressing described above.

NOTE

When the CPU is configured to generate 30-bit physical addresses, SBR<31:30> are ignored.

Figure 2–14: IPR 0C (hex), SBR and IPR 0D (hex), SLR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     Physical Page Address of SPT                   | 0  0  0  0  0  0  0  0  0| :SBR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0  0  0  0  0  0  0  0  0  0|                    Length of SPT in Longwords                  | :SLR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

The system space translation algorithm is shown graphically in Figure 2–15.

Figure 2–15: System Space Translation Algorithm

```
                       3 3
                       1 0                   9 8      0
system-space          +-+---------------------+--------+
virtual address:      |1| virtual page number |  byte  |
                      +-+---------------------+--------+
                        |                     |\        \
                        |      extract VPN,   | \        \
                        |      check length,  |  \        \
                  3    2|2     and add        |   \        \
                  1    4|3                   2|1 0 \        \
                 +------+---------------------+---+ \        \
SBR:             | physical address of SPT base   |  \        \
                 +--------------------------------+   |        |
                 | sign-extend PA<29> to PA<31:30>|   |        |
                 | if in 30-bit mode              |   |        |
                 |                                |   |        |
                 |3             yields            |   |        |
                 |1                              0|   |        |
                 +--------------------------------+   |        |
                 |    physical address of SPTE    |   |        |
                 +--------------------------------+   |        |
                                                      |        |
                                                      |        |
                               fetch                  |        |
                  3    2 2                            |        |
                  1    3 2                        0   |        |
                 +------+-------------------------+   |        |
SPTE:            |      |    page frame number    |   |        |
                 +------+-------------------------+   |        |
                        | check access in current |   |        |
                        | mode,                   |   |        |
                        | sign-extend PTE<20> to  |   |        |
                        | PTE<22:21> if in 30-bit |   |        |
                        | mode                    |   |        |
                        |          merge          |  /        /
                        |3                        | /        /
                        |1                       9|/8     0 /
                        +-------------------------+--------+
physical address:       |    page frame number    |  byte  |
                        +-------------------------+--------+
```

2.6.3 Process Space Address Translation

A virtual address with bit <31> = 0 is an address in the process virtual address space. Process space is divided into two equal sized, separately mapped regions. If virtual address bit <30> = 0, the address is in region P0. If virtual address bit <30> = 1, the address is in region P1.

2.6.3.1 P0 Region Address Translation

The P0 region of the address space is mapped by the P0 Page Table (P0PT), which is defined by the P0 Base Register (P0BR) and the P0 Length Register (P0LR). The P0BR contains the system page-aligned virtual address of the P0 Page Table. The P0LR contains the size of the P0PT in longwords, that is, the number of Page Table Entries. The Page Table Entry addressed by the P0 Base Register maps the first page of the P0 region of the virtual address space, that is, virtual byte address 0. The P0 base and length registers are shown in Figure 2–16.

The P0 space translation algorithm is shown graphically in Figure 2–17.

Figure 2–16: IPR 08 (hex), P0BR and IPR 09 (hex), P0LR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1  0|              System Virtual Page Address of P0PT             | 0  0  0  0  0  0  0  0  0| : P0BR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0  0  0  0  0  0  0  0  0  0|                   Length of P0PT in Longwords                   | :P0LR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 2–17: P0 Space Translation Algorithm

```
                       3 3 2
                       1 0 9                 9 8     0
process-space         +---+-------------------+-------+
virtual address:      | 0 |virtual page number|  byte |
                      +---+-------------------+-------+
                          |                   |\       \
                          |    extract VPN,   | \       \
                          |    check length,  |  \       \
                   3     2|2   and add        |   \       \
                   1     3|2                 2|1 0 \       \
                  +-------+-------------------+---+ \       \
P0BR:             | virtual address of P0PT base  |  \       \
                  +-------------------------------+   |       |
                  |                               |   |       |
                  |                               |   |       |
                  |             yields            |   |       |
                  |3 3 2                          |   |       |
                  |1 0 9                 9 8     0|   |       |
virtual address   +---+---------------------------+   |       |
of P0PTE:         |   |virtual page number| byte  |   |       |
                  +---+---------------------------+   |       |
                 fetch using system-space translation |       |
                 algorithm, including length check,   |       |
                 but without access check             |       |
                   3    2 2                           |       |
                   1    3 2                       0   |       |
                  +------+-------------------------+  |       |
P0PTE:            |      |    page frame number    |  |       |
                  +------+-------------------------+  |       |
                         | check access in current |  |       |
                         | mode,                   |  |       |
                         | sign-extend PTE<20> to  |  |       |
                         | PTE<22:21> if in 30-bit |  |       |
                         | mode                    |  |       |
                         |         merge           |  /       /
                         |3                        | /       /
                         |1                       9|/8    0 /
                         +-------------------------+-------+
physical address:        |    page frame number    | byte  |
                         +-------------------------+-------+
```

2.6.3.2 P1 Region Address Translation

The P1 region of the address space is mapped by the P1 Page Table (P1PT), which is defined by the P1 Base Register (P1BR) and the P1 Length Register (P1LR). Because P1 space grows towards smaller addresses, and because a consistent hardware interpretation of the base and length registers is desirable, P1BR and P1LR describe the portion of P1 space that is NOT accessible.

Note that P1LR contains the number of nonexistent PTEs. P1BR contains the page-aligned virtual address of what would be the PTE for the first page of P1, that is, virtual byte address 40000000 (hex). The address in P1BR is not necessarily an address in system space,but all the addresses of PTEs must be in system space.

The P1 space translation algorithm is shown graphically in Figure 2–19.

Figure 2–18: IPR 0A (hex), P1BR and IPR 0B (hex), P1LR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    Virtual Page Address of P1PT                    | 0  0  0  0  0  0  0  0  0| : P1BR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0  0  0  0  0  0  0  0  0  0|              (2 ** 21) - Length of P1PT in Longwords            | :P1LR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 2–19: P1 Space Translation Algorithm

```
                        3 3 2
                        1 0 9                 9 8     0
process-space          +---+-------------------+-------+
virtual address:       | 0 |virtual page number|  byte |
                       +---+-------------------+-------+
                           |                   |\       \
                           |    extract VPN,   | \       \
                           |    check length,  |  \       \
                   3      2|2   and add        |   \       \
                   1      3|2                 2|1 0 \       \
                  +--------+-------------------+---+ \       \
P1BR:             | virtual address of P1PT base   |  \       \
                  +--------------------------------+   |       |
                  |                                |   |       |
                  |                                |   |       |
                  |              yields            |   |       |
                  |3 3 2                           |   |       |
                  |1 0 9                  9 8     0|   |       |
virtual address   +---+----------------------------+   |       |
of P1PTE:         |   |virtual page number|  byte  |   |       |
                  +---+----------------------------+   |       |
                 fetch using system-space translation  |       |
                 algorithm, including length check,    |       |
                 but without access check              |       |
                  3     2 2                            |       |
                  1     3 2                       0    |       |
                 +-------+-------------------------+   |       |
P1PTE:           |       |    page frame number    |   |       |
                 +-------+-------------------------+   |       |
                         | check access in current |   |       |
                         | mode,                   |   |       |
                         | sign-extend PTE<20> to  |   |       |
                         | PTE<22:21> if in 30-bit |   |       |
                         | mode                    |   |       |
                         |          merge          |  /       /
                         |3                        | /       /
                         |1                       9|/8    0 /
                         +-------------------------+--------+
physical address:        |    page frame number    | byte   |
                         +-------------------------+--------+
```

2.6.4 Page Table Entry

If the CPU is configured to generate 30-bit physical addresses, it interprets PTEs in the 21-bit PFN format shown in Figure 2–20. Conversely, if the CPU is configured to generate 32-bit physical addresses, it interprets PTEs in the 25-bit PFN format shown in Figure 2–21. Note that bits <24:23> of the 25-bit PFN format are ignored by the NVAX CPU chip, which implements only 32-bit physical addresses. The PTE formats shown below are described both in DEC Standard 032, and in Chapter 12.

Figure 2–20: PTE Format (21-bit PFN)

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| V|   PROT    | M| Z| OWN | S| S|                      Page Frame Number                       | :PTE
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 2–21: PTE Format (25-bit PFN)

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| V|   PROT    | M| S| S| S|                         Page Frame Number                          | :PTE
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Table 2–7: PTE Protection Code Access Matrix

Code			Current Mode				
Decimal	Binary	Mnemonic	K	E	S	U	Comment
0	0000	NA	–	–	–	–	no access
1	0001			unpredictable			reserved
2	0010	KW	RW	–	–	–	
3	0011	KR	R	–	–	–	
4	0100	UW	RW	RW	RW	RW	all access
5	0101	EW	RW	RW	–	–	
6	0110	ERKW	RW	R	–	–	
7	0111	ER	R	R	–	–	
8	1000	SW	RW	RW	RW	–	
9	1001	SREW	RW	RW	R	–	
10	1010	SRKW	RW	R	R	–	
11	1011	SR	R	R	R	–	
12	1100	URSW	RW	RW	RW	R	
13	1101	UREW	RW	RW	R	R	
14	1110	URKW	RW	R	R	R	
15	1111	UR	R	R	R	R	

Access Modes

K = Kernel
E = Executive
S = Supervisor
U = User

Access Types

R = Read
W = Write
– = No access

2.6.5 Translation Buffer

In order to save actual memory references when repeatedly referencing pages, the NVAX CPU Chip uses a translation buffer to remember successful virtual address translations and page status. The translation buffer contains 96 fully associative entries. Both system and process references share these entries.

Translation buffer entries are replaced using a not-last-used (NLU) algorithm. This algorithm guarantees that the replacement pointer is not pointing at the last translation buffer entry to be used. This is accomplished by rotating the replacement pointer to the next sequential translation buffer entry if it is pointing to an entry that has just been accessed. Both D-stream and I-stream references can cause the NLU to cycle. When the translation buffer does not contain a reference's virtual address and page status, the machine updates the translation buffer by replacing the entry that is selected by the replacement pointer.

2.7 Exceptions and Interrupts

At certain times during the operation of a system, events within the system require the execution of software routines outside the explicit flow of control of instruction execution. An exception is an event that is relevant primarily to the currently executing process and normally invokes a software routine in the context of the current process. An interrupt is an event which is usually due to some activity outside the current process and invokes a software routine outside the context of the current process.

Exceptions and interrupts are reported by constructing a frame on the stack and then dispatching to the service routine through an event-specific vector in the System Control Block (SCB). The minimum stack frame for any interrupt or exception is a PC/PSL pair as shown in Figure 2–22.

Figure 2–22: Minimum Exception Stack Frame

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+-----------------------------------------------------------------------------------------------+
|                                              PC                                               | :(SP)
+-----------------------------------------------------------------------------------------------+
|                                              PSL                                              |
+-----------------------------------------------------------------------------------------------+
```

This minimum stack frame is used for all interrupts. Certain exceptions expand the stack frame by pushing additional parameters on the stack above the PC/PSL pair as shown in Figure 2–23.

Figure 2–23: General Exception Stack Frame

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                          Parameter n                                          | :(SP)
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 :                                              :                                              :
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                          Parameter 1                                          |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              PC                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              PSL                                              |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

What parameters, if any, are pushed on the stack above the PC/PSL pair is a function of the specific exception being reported.

2.7.1 Interrupts

DEC Standard 032 defines 31 interrupt priority levels, a subset of which is implemented by the NVAX CPU. When an interrupt request is generated, the hardware compares the request with the current IPL of the CPU. If the new request is of higher priority an internal request is generated. At the completion of the current instruction (or at selected points during the execution of interruptible instructions), a microcode interrupt handler is invoked to process the request. With hardware assistance, the microcode handler determines the highest priority interrupt, updates the IPL, pushes a PC/PSL pair on the stack, and dispatches to a macrocode interrupt handler through the appropriate location in the SCB.

Of the 31 interrupt priority levels defined by DEC Standard 032, the NVAX CPU makes use of 24 of them, as shown in Table 2–8.

Table 2–8: Interrupt Priority Levels

IPL (hex)	IPL (decimal)	Interrupt Condition
1F	31	**HALT_L** asserted (non maskable)
1E	30	**PWRFL_L** asserted
1D	29	**H_ERR_L** asserted (or internal hard error detected)
1C	28	Unused
1B	27	Performance monitoring interrupt (internally handled by microcode)
1A	26	**S_ERR_L** asserted (or internal soft error detected)
18–19	24–25	Unused
17	23	**IRQ_L<3>** asserted
16	22	**IRQ_L<2>** or **INT_TIM_L** asserted (**IRQ_L<2>** takes priority)
15	21	**IRQ_L<1>** asserted
14	20	**IRQ_L<0>** asserted
10–13	16-19	Unused
01–0F	01-15	Software interrupt asserted

Interrupts are discussed in more detail in Chapter 10.

2.7.1.1 Interrupt Control Registers

The interrupt system is controlled by three processor registers: the Interrupt Priority Level Register (IPL), the Software Interrupt Request Register (SIRR), and the Software Interrupt Summary Register (SISR).

A new interrupt priority level may be loaded into PSL<20:16> by writing the new value to IPL<4:0>. The IPL register is shown in Figure 2–24.

Figure 2–24: IPR 12 (hex), IPL

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0|  PSL<20:16>  | :IPL
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

A software interrupt may be requested by writing the desired level to SIRR<3:0>. The SIRR register is shown in Figure 2–25.

Figure 2–25: IPR 14 (hex), SIRR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0|Request IPL| :SIRR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

The SISR register records pending software interrupt requests at levels 01 through 0F (hex). The SISR register is shown in Figure 2–26.

Figure 2–26: IPR 15 (hex), SISR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 0| :SISR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                  |  |                                   |  |
                                 IPL 15 request --+  |         . . .     IPL 2 request --+  |
                                    IPL 14 request --+                      IPL 1 request --+
```

2.7.2 Exceptions

The VAX architecture recognizes six classes of exceptions. Table 2–9 lists instances of exceptions in each class.

Table 2–9: Exception Classes

Exception Class	Instances
Arithmetic traps/faults	Integer overflow trap Integer divide-by-zero trap Subscript range trap Floating overflow fault Floating divide-by-zero fault Floating underflow fault

Table 2–9 (Cont.): Exception Classes

Exception Class	Instances
Memory management exceptions	Access control violation fault Translation not valid fault M=0 fault
Operand reference exceptions	Reserved addressing mode fault Reserved operand fault or abort
Instruction execution exceptions	Reserved/privileged instruction fault Emulated instruction faults. XFC fault Change-mode trap Breakpoint fault Vector disabled fault
Tracing exceptions	Trace fault
System failure exceptions	Kernel-stack-not-valid abort Interrupt-stack-not-valid halt Console error halt Machine check abort

A trap is an exception that occurs at the end of the instruction that caused the exception. Therefore, the PC saved on the stack is the address of the next instruction that would normally have been executed.

A fault is an exception that occurs during an instruction and that leaves the registers and memory in a consistent state such that elimination of the fault condition and restarting the instruction will give correct results. After the instruction faults, the PC saved on the stack points to the instruction that faulted.

An abort is an exception that occurs during an instruction. An abort leaves the value of registers and memory UNPREDICTABLE such that the instruction cannot necessarily be correctly restarted, completed, simulated, or undone. In most instances, the NVAX microcode attempts to convert an abort into a fault by restoring the state that was present at the start of the instruction which caused the abort.

The following sections describe only those exceptions which are unique to the NVAX CPU, or where DEC Standard 032 is not clear about the implementation.

2.7.2.1 Arithmetic Exceptions

Arithmetic exceptions are detected during the execution of instructions that perform integer or floating point arithmetic manipulations. Whether the exception is reported as a trap or a fault is a function of the specific event. In any case, the exception is reported through SCB vector 34 (hex) with the stack frame shown in Figure 2–27. Table 2–10 lists the exceptions reported by this mechanism.

Figure 2–27: Arithmetic Exception Stack Frame

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                          Type Code                                            | :(SP)
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                             PC                                                |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                             PSL                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Table 2–10: Arithmetic Exceptions

Type Code			
Decimal	Hex	Type	Exception
1	1	Trap	Integer overflow
2	2	Trap	Integer divide-by-zero
7	7	Trap	Subscript range
8	8	Fault	Floating overflow
9	9	Fault	Floating divide-by-zero
10	A	Fault	Floating underflow

2.7.2.2 Memory Management Exceptions

Memory management exceptions are detected during a memory reference and are always reported as faults. The three memory management exceptions are listed in Table 2–11. All three exceptions push the same frame on the stack, as shown in Figure 2–28. The top longword of the stack frame contains a fault parameter whose bits are described in Table 2–12.

Table 2–11: Memory Management Exceptions

SCB Vector	Exception
20 (hex)	Access control violation
24 (hex)	Translation not valid
3C (hex)	Modify fault

Figure 2–28: Memory Management Exception Stack Frame

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0| M| P| L| :(SP)
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                          Some Virtual Address in the Faulting Page                            |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              PC                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              PSL                                              |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Table 2–12: Memory Management Exception Fault Parameter

Bit	Mnemonic	Meaning
0	L	Length violation
1	P	PTE reference
2	M	Modify or write intent

2.7.2.3 Emulated Instruction Exceptions

The NVAX CPU implements the VAX base instruction group. For certain instructions outside that group, the NVAX microcode provides support for the macrocode emulation of instructions. There are two types of emulation exceptions, depending on whether PSL<FPD> is set at the beginning of the instruction.

If PSL<FPD>=0 at the beginning of the instruction, the exception is reported through SCB vector C8 (hex) as a trap with the stack frame shown in Figure 2–29. The longwords in the stack frame are described in Table 2–13.

Figure 2–29: Instruction Emulation Trap Stack Frame

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                            Opcode                                             | :(SP)
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                            Old PC                                             |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                         Specifier #1                                          |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                         Specifier #2                                          |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                         Specifier #3                                          |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                         Specifier #4                                          |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                         Specifier #5                                          |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                         Specifier #6                                          |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                         Specifier #7                                          |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                         Specifier #8                                          |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              PC                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              PSL                                              |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Table 2–13: Instruction Emulation Trap Stack Frame

Location	Use
Opcode	Zero-extended opcode of the emulated instruction
Old PC	PC of the opcode of the emulated instruction
Specifiers	Address of the specified operand for specifiers of access type write (.wx) or address (.ax). Operand value for specifiers of access type read (.rx). For read-type operands whose size is smaller than a longword, the remaining bits are UNPREDICTABLE. For those instructions that don't have 8 specifiers, the remaining specifier longwords contain UNPREDICTABLE values
New PC	PC of the instruction following the emulated instruction
PSL	PSL saved at the time of the trap

If PSL<FPD>=1 at the beginning of the instruction, the exception is reported through SCB vector CC (hex) as a fault with the stack frame shown in Figure 2–30. In this case, PC is that of the opcode of the emulated instruction.

Figure 2–30: Suspended Emulation Fault Stack Frame

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              PC                                               | :(SP)
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              PSL                                              |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

2.7.2.4 Vector Unit Disabled Fault

When the NVAX CPU attempts to issue a vector instruction to the optional vector processor, it may discover that the vector unit is disabled. In this case, a vector unit disabled fault is initiated through SCB vector 68 (hex). There are no parameters for this exception (besides the usual PC/PSL pair), and the reason for the exception must be determined by reading the appropriate vector unit registers.

2.7.2.5 Machine Check Exceptions

A machine check exception is reported through SCB vector 04 (hex) when the NVAX CPU detects an error condition. The frame pushed on the stack for a machine check indicates the type of error and provides internal state information that may help identify the cause of the error. The generic machine check stack frame is shown in Figure 2–31. Machine checks are discussed at length in Chapter 15.

Figure 2–31: Generic Machine Check Stack Frame

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      Byte Count of Parameters, Excluding This Longword                        | :(SP)
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
  :                                              :                                              :
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              PC                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              PSL                                              |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

2.7.2.6 Console Halts

In certain microcode flows, the NVAX microcode may detect an inconsistency in internal state, a kernel-mode HALT, or a system reset. In these instances, the microcode initiates a hardware restart sequence which passes control to the console program.

When a hardware restart sequence is initiated, the NVAX microcode saves the current CPU state, partially initializes the CPU, and passes control to the console program at physical address E0040000 (hex).

During a hardware restart sequence, the stack pointer is saved in the appropriate stack pointer IPR (0 through 4), the current PC is saved in IPR 42 (SAVPC), and the current PSL, halt code, and validity flag are saved in IPR 43 (SAVPSL). The format of SAVPC and SAVPSL are shown in Figure 2–32.

Figure 2–32: IPR 2A (hex), SAVPC and IPR 2B (hex), SAVPSL

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                                                           Saved PC                                                            | :SAVPC
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                          PSL<31:16>                           |   |   |   Halt Code           |           PSL<7:0>            | :SAVPSL
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                                                                  |   |
                                                       MAPEN<0> --+   |
                                                Invalid SAVPSL if 1 --+
```

Console halts are discussssed in detail in Chapter 15.

2.8 System Control Block

The System Control Block (SCB) is a page containing the vectors for servicing interrupts and exceptions. The SCB is pointed to by the System Control Block Base Register (SCBB), whose format is shown in Figure 2–33. For best performance, SCBB should contain a page-aligned address. Microcode forces a longword-aligned SCBB by clearing bits <1:0> of the new value before loading the register.

NOTE

> When the CPU is configured to generate 30-bit physical addresses, SCBB<31:30> are ignored.

Figure 2–33: IPR 11 (hex), SCBB

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                                  Physical Page Address of SCB                             |            SBZ            |  0   0| :SCBB
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

2.8.1 System Control Block Vectors

An SCB vector is an aligned longword in the SCB through which the NVAX microcode dispatches interrupts and exceptions. Each SCB vector has the format shown in Figure 2–34. The fields of the vector are described in Table 2–14.

Figure 2–34: System Control Block Vector

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                           longword address of service routine                           |code |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Table 2–14: System Control Block Vector

Bits	Contents
31:2	Virtual address of the service routine for the interrupt or exception. The routine must be longword aligned, as the microcode forces the lower two bits of the address to 00
1:0	Code, interpreted as follows:

Value	Meaning
00	The event is to be serviced on the kernel stack unless the CPU is already on the interrupt stack, in which case the event is serviced on the interrupt stack
01	The event is to be serviced on the interrupt stack. If the event is an exception, the IPL is raised to 1F (hex)
10	Unimplemented, results in a console error halt
11	Unimplemented, results in a console error halt

2.8.2 System Control Block Layout

The System Control Block layout is shown in Table 2–15.

Table 2–15: System Control Block Layout

Vector	Name	Type	Param	Notes
00	passive release	interrupt	0	IPL is raised to request IPL
04	machine check	abort	6	parameters reflect machine state; must be serviced on interrupt stack
08	kernel stack not valid	abort	0	must be serviced on interrupt stack
0C	power fail	interrupt	0	IPL is raised to 1E (hex)
10	reserved/privileged instruction	fault	0	
14	customer reserved instruction	fault	0	XFC instruction
18	reserved operand	fault/abort	0	not always recoverable
1C	reserved addressing mode	fault	0	
20	access control violation/vector alignment fault	fault	2	parameters are virtual address, status code

Table 2–15 (Cont.): System Control Block Layout

Vector	Name	Type	Param	Notes
24	translation not valid	fault	2	parameters are virtual address, status code
28	trace pending	fault	0	
2C	breakpoint instruction	fault	0	
30	unused	–	–	compatibility mode in other VAXes
34	arithmetic trap/fault	trap/fault	1	parameter is type code
38–3C	unused	–	–	–
40	CHMK	trap	1	parameter is sign-extended operand word
44	CHME	trap	1	parameter is sign-extended operand word
48	CHMS	trap	1	parameter is sign-extended operand word
4C	CHMU	trap	1	parameter is sign-extended operand word
50	unused	–	–	–
54	soft error notification	interrupt	0	IPL is 1A (hex)
58	Performance monitoring counter overflow	interrupt	–	Internal interrupt at IPL 1B (hex). This vector supplies the physical base address of the block of performance monitoring counts in memory. See Chapter 18 for details.
5C	unused	–	–	–
60	hard error notification	interrupt	0	IPL is 1D (hex)
64	unused	–	–	–
68	vector unit disabled	fault	0	vector instructions
6C–7C	unused	–	–	–
80	interprocessor interrupt	interrupt	0	IPL is 16 (hex)
84	software level 1	interrupt	0	
88	software level 2	interrupt	0	ordinarily used for AST delivery
8C	software level 3	interrupt	0	ordinarily used for process scheduling
90–BC	software levels 4–15	interrupt	0	
C0	interval timer	interrupt	0	IPL is 16 (hex)
C4	unused	–	–	–

Table 2–15 (Cont.): System Control Block Layout

Vector	Name	Type	Param	Notes
C8	emulation start	fault	10	same mode exception, FPD=0; parameters are opcode, PC, specifiers
CC	emulation continue	fault	0	same mode exception, FPD=1; no parameters
D0–F4	unused	–	–	–
F8	console receiver	interrupt	0	IPL is 15 (hex)
FC	console transmitter	interrupt	0	IPL is 15 (hex)
100–FFFC	device vectors	interrupt	0	Device interrupt vectors

2.9 CPU Identification

Software may quickly determine on which CPU it is executing in a multi-processor system by reading the CPUID processor register. The format of this register is shown in Figure 2–35.

Figure 2–35: IPR 0E (hex), CPUID

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0|   CPU Identification  | :CPUID
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

The CPUID processor register is implemented internally as an 8-bit read-write register. The source of the CPU ID information is system-specific, and it is the responsibility of the console firmware at powerup to determine the CPU ID from the system-specific source, and write the CPU ID register to the correct value.

2.10 System Identification

The System Identification Register (SID) is a read-only register which includes the the system (actually the CPU) type, and the microcode revision number. The format of the SID register is shown in Figure 2–36.

Figure 2–36: IPR 3E (hex), SID

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|        CPU Type       | 0  0  0  0  0  0  0  0  0  0|Patch Revision|NS|   Microcode Revision  | :SID
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Table 2–16: SID Field Descriptions

Name	Extent	Type	Description
Microcode Revision	7:0	RO	This field contains the microcode (chip) revision number. This number is incremented for each pass of the chip.
NS	8	RO,0	If this bit is a zero, there is either no microcode patch loaded, or the patch is a standard patch. If this bit is a one, a non-standard microcode patch is loaded. A non-standard patch is one which goes beyond the formally released patches, such as a patch used for performance analysis. This bit is cleared on chip reset.
Patch Revision	13:9	RO,0	If this field is zero, no microcode patch is loaded. If this field is non-zero, a microcode patch is loaded and this field indicates the patch number. This field is cleared on chip reset.
CPU Type	31:24	RO	This field contains 19 (decimal), indicating that this is an NVAX CPU.

NOTE

The patch revision and non-standard patch fields (SID<13:8>) were added in pass 2 of the NVAX chip.

2.11 Process Structure

A process is a single thread of execution. The context of the current process is contained in the Process Control Block (PCB). The PCB is pointed to by the Process Control Block Base register (PCBB), which is shown in Figure 2–37. The format of the process control block is shown in Figure 2–38. Microcode forces a longword-aligned PCBB by clearing bits <1:0> of the new value before loading the register.

NOTE

When the CPU is configured to generate 30-bit physical addresses, PCBB<31:30> are ignored.

Figure 2–37: IPR 10 (hex), PCBB

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                         Physical Longword Address of the PCB                           | 0  0| :PCBB
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 2–38: Process Control Block

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              KSP                                              | :PCB
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              ESP                                              |  +4
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              SSP                                              |  +8
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              USP                                              |  +12
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              R0                                               |  +16
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              R1                                               |  +20
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              R2                                               |  +24
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              R3                                               |  +28
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              R4                                               |  +32
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              R5                                               |  +36
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              R6                                               |  +40
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              R7                                               |  +44
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              R8                                               |  +48
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              R9                                               |  +52
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              R10                                              |  +56
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              R11                                              |  +60
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                           AP(R12)                                             |  +64
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                           FP(R13)                                             |  +68
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              PC                                               |  +72
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              PSL                                              |  +76
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                             P0BR                                              |  +80
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0  0  0  0  0| ASTLVL | 0  0|                                P0LR                             |  +84
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              P1BR                                             |  +88
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|PME 0  0  0  0  0  0  0  0  0|                                P1LR                             |  +92
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
```

2.12 Processor Registers

The processor registers that are implemented by the NVAX CPU chip, and those that are required of the system environment, are logically divided into five groups, as follows:

- Normal—Those IPRs that address individual registers in the NVAX CPU chip or system environment.
- Bcache tag IPRs—The read-write block of IPRs that allow direct access to the Bcache tags.
- Bcache deallocate IPRs—The write-only block of IPRs by which a Bcache block may be deallocated.
- Pcache tag IPRs—The read-write block of IPRs that allow direct access to the Pcache tags.
- Pcache data parity IPRs—The read-write block of IPRs that allow direct access to the Pcache data parity bits.

Each group of IPRs is distinguished by a particular pattern of bits in the IPR address, as shown in Figure 2–39.

Figure 2–39: IPR Address Space Decoding

```
Normal IPR Address

 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|       SBZ          | 0|                     SBZ                       |      IPR Number       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

Bcache Tag IPR Address

 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|       SBZ          | 1| 0| 0| x|               Bcache Tag Index                |    SBZ       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

Bcache Deallocate IPR Address

 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|        SBZ         | 1| 0| 1| x|         Bcache Tag Deallocate Index           |    SBZ       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

Pcache Tag IPR Address

 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|        SBZ         | 1| 1| 0|         SBZ              |  | Pcache Tag Index   |     SBZ      |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                          |
                     Pcache Set Select (0=left, 1=right) -+

Pcache Data Parity IPR Address

 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|        SBZ         | 1| 1| 1|         SBZ              |  | Pcache Tag Index   |     |  SBZ   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                          |                         |
                     Pcache Set Select (0=left, 1=right) -+         Subblock select +
```

The numeric range for each of the four groups is shown in Table 2–17.

Table 2–17: IPR Address Space Decoding

IPR Group	Mnemonic[2]	IPR Address Range (hex)	Contents
Normal		00000000..000000FF[1]	256 individual IPRs.
Bcache Tag	BCTAG	01000000..011FFFE0[1]	64k Bcache tag IPRs, each separated by 20(hex) from the previous one.
Bcache Deallocate	BCFLUSH	01400000..015FFFE0[1]	64k Bcache tag deallocate IPRs, each separated by 20(hex) from the previous one.
Pcache Tag	PCTAG	01800000..01801FE0[1]	256 Pcache tag IPRs, 128 for each Pcache set, each separated by 20(hex) from the previous one.
Pcache Data Parity	PCDAP	01C00000..01C01FF8[1]	1024 Pcache data parity IPRs, 512 for each Pcache set, each separated by 8(hex) from the previous one.

[1] Unused fields in the IPR addresses for these groups should be zero. Neither hardware nor microcode detects and faults on an address in which these bits are non-zero. Although non-contiguous address ranges are shown for these groups, the entire IPR address space maps into one of the these groups. If these fields are non-zero, the operation of the CPU is UNDEFINED.

[2] The mnemonic is for the first IPR in the block

NOTE

> The address ranges shown above are those used by the programmer. When processing normal IPRs, the microcode shifts the IPR number left by 2 bits for use as an IPR command address. This positions the IPR number to bits <9:2> and modifies the address range as seen by the hardware to 0..3FC, with bits <1:0>=00. No shifting is performed for the other groups of IPR addresses.

Because of the sparse addressing used for IPRs in groups other than the normal group, valid IPR addresses are not separated by one. Rather, valid IPR addresses are separated by either 8 or 20(hex). For example, the IPR address for Bcache tag 0 is 01000000 (hex), and the IPR address for Bcache tag 1 is 01000020 (hex). In this group, bits <4:0> of the IPR address are ignored, so IPR numbers 01000001 through 0100001F all address Bcache tag 0. Similarly, the IPR address for the first subblock of Pcache data parity is 01C00000 (hex), and the IPR address for the second subblock of Pcache data parity is 01C00008 (hex).

Processor registers in all groups except the normal group are processed entirely by the NVAX CPU chip and will never appear on the NDAL. This is also true for a number of the IPRs in the normal group. IPRs in the normal group that are not processed by the NVAX CPU chip are converted into I/O space references and passed to the system environment via a read or write command on the NDAL.

Each of the 256 possible IPRs in the normal group are of longword length, so a 1KB block of I/O space is required to convert each possible IPR to a unique I/O space longword. This block starts at address E1000000 (hex). Conversion of an IPR address to an I/O space address in this block is done by shifting the IPR address left into bits <9:2>, filling bits <1:0> with zeros, and merging in the base address of the block. This can be expressed by the equation

$$IO\ ADDRESS = E1000000 + (IPR\ NUMBER * 4)$$

The actual hardware implementation of this is different in that the IPR number is shifted left by 2 bits, and bits <31:30,24> are set. There is no multiply or add done as one might conclude from the equation.

Because many of the 256 possible IPRs in the normal group are processed entirely by the NVAX CPU chip, the corresponding I/O space location in the 1KB block is never referenced as a result of an MTPR/MFPR to or from these IPRs. However, note that a programmer can indeed reference these locations via an explicit I/O space reference with, e.g., MOVL. References to this block of I/O space locations with instructions other than MTPR/MFPR may result in UNDEFINED behavior.

The processor registers implemented by the NVAX CPU are are shown in Table 2–18.

NOTE

> Many of the processor registers listed in Table 2–18 are used internally by the microcode during normal operation of the CPU, and are not intended to be referenced by software except during test or diagnosis of the system. These registers are flagged with the notation "Testability and diagnostic use only; not for software use in normal operation". References by software to these registers during normal operation can cause UNDEFINED behavior of the CPU.

Table 2–18: Processor Registers

		Number					
Register Name	**Mnemonic**	**(Dec)**	**(Hex)**	**Type**	**Impl**	**Cat**	**I/O Addr**
Kernel Stack Pointer	KSP	0	0	RW	NVAX	1-1	
Executive Stack Pointer	ESP	1	1	RW	NVAX	1-1	
Supervisor Stack Pointer	SSP	2	2	RW	NVAX	1-1	
User Stack Pointer	USP	3	3	RW	NVAX	1-1	
Interrupt Stack Pointer	ISP	4	4	RW	NVAX	1-1	
Reserved		5	5			3	E1000014
Reserved		6	6			3	E1000018
Reserved		7	7			3	E100001C
P0 Base Register	P0BR	8	8	RW	NVAX	1-2	
P0 Length Register	P0LR	9	9	RW	NVAX	1-2	
P1 Base Register	P1BR	10	A	RW	NVAX	1-2	
P1 Length Register	P1LR	11	B	RW	NVAX	1-2	
System Base Register	SBR	12	C	RW	NVAX	1-2	
System Length Register	SLR	13	D	RW	NVAX	1-2	
CPU Identification[1]	CPUID	14	E	RW	NVAX	2-1	
Reserved		15	F			3	E100003C
Process Control Block Base	PCBB	16	10	RW	NVAX	1-1	
System Control Block Base	SCBB	17	11	RW	NVAX	1-1	
Interrupt Priority Level[1]	IPL	18	12	RW	NVAX	1-1	
AST Level[1]	ASTLVL	19	13	RW	NVAX	1-1	
Software Interrupt Request Register	SIRR	20	14	W	NVAX	1-1	
Software Interrupt Summary Register[1]	SISR	21	15	RW	NVAX	1-1	
Reserved		22	16			3	E1000058
Reserved		23	17			3	E100005C
Interval Counter Control/Status[1,2]	ICCS	24	18	RW	NVAX	2-7	E1000060
Next Interval Count	NICR	25	19	W	System	3-7	E1000064
Interval Count	ICR	26	1A	R	System	3-7	E1000068
Time of Year Register	TODR	27	1B	RW	System	2-3	E100006C
Console Storage Receiver Status	CSRS	28	1C	RW	System	2-3	E1000070
Console Storage Receiver Data	CSRD	29	1D	R	System	2-3	E1000074
Console Storage Transmitter Status	CSTS	30	1E	RW	System	2-3	E1000078
Console Storage Transmitter Data	CSTD	31	1F	W	System	2-3	E100007C
Console Receiver Control/Status	RXCS	32	20	RW	System	2-3	E1000080

[1] Initialized on reset

[2] Subset or full implementation depending on ECR control bit

Table 2–18 (Cont.): Processor Registers

		Number					
Register Name	**Mnemonic**	**(Dec)**	**(Hex)**	**Type**	**Impl**	**Cat**	**I/O Address**
Console Receiver Data Buffer	RXDB	33	21	R	System	2-3	E1000084
Console Transmitter Control/Status	TXCS	34	22	RW	System	2-3	E1000088
Console Transmitter Data Buffer	TXDB	35	23	W	System	2-3	E100008C
Reserved		36	24			3	E1000090
Reserved		37	25			3	E1000094
Machine Check Error Register	MCESR	38	26	W	NVAX	2-1	
Reserved		39	27			3	E100009C
Reserved		40	28			3	E10000A0
Reserved		41	29			3	E10000A4
Console Saved PC	SAVPC	42	2A	R	NVAX	2-1	
Console Saved PSL	SAVPSL	43	2B	R	NVAX	2-1	
Reserved		44	2C			3	E10000B0
Reserved		45	2D			3	E10000B4
Reserved		46	2E			3	E10000B8
Reserved		47	2F			3	E10000BC
Reserved		48	30			3	E10000C0
Reserved		49	31			3	E10000C4
Reserved		50	32			3	E10000C8
Reserved		51	33			3	E10000CC
Reserved		52	34			3	E10000D0
Reserved		53	35			3	E10000D4
Reserved		54	36			3	E10000D8
I/O System Reset Register	IORESET	55	37	W	System	2-3	E10000DC
Memory Management Enable[1]	MAPEN	56	38	RW	NVAX	1-2	
Translation Buffer Invalidate All	TBIA	57	39	W	NVAX	1-1	
Translation Buffer Invalidate Single	TBIS	58	3A	W	NVAX	1-1	
Reserved		59	3B			3	E10000EC
Reserved		60	3C			3	E10000F0
Performance Monitor Enable[1]	PME	61	3D	RW	NVAX	2-1	
System Identification	SID	62	3E	R	NVAX	2-1	
Translation Buffer Check	TBCHK	63	3F	W	NVAX	1-1	

[1] Initialized on reset

Table 2–18 (Cont.): Processor Registers

		Number					
Register Name	Mnemonic	(Dec)	(Hex)	Type	Impl	Cat	I/O Addre
IPL 14 Interrupt ACK[3]	IAK14	64	40	R	System	2-3	E1000100
IPL 15 Interrupt ACK[3]	IAK15	65	41	R	System	2-3	E1000104
IPL 16 Interrupt ACK[3]	IAK16	66	42	R	System	2-3	E1000108
IPL 17 Interrupt ACK[3]	IAK17	67	43	R	System	2-3	E100010C
Clear Write Buffer[3]	CWB	68	44	RW	System	2-3	E1000110
Reserved		69	45			3	E1000114
Reserved		70	46			3	E1000118
Reserved		71	47			3	E100011C
Reserved		72	48			3	E1000120
Reserved		73	49			3	E1000124
Reserved		74	4A			3	E1000128
Reserved		75	4B			3	E100012C
Reserved		76	4C			3	E1000130
Reserved		77	4D			3	E1000134
Reserved		78	4E			3	E1000138
Reserved		79	4F			3	E100013C
Reserved		80	50			3	E1000140
Reserved		81	51			3	E1000144
Reserved		82	52			3	E1000148
Reserved		83	53			3	E100014C
Reserved		84	54			3	E1000150
Reserved		85	55			3	E1000154
Reserved		86	56			3	E1000158
Reserved		87	57			3	E100015C
Reserved		88	58			3	E1000160
Reserved		89	59			3	E1000164
Reserved		90	5A			3	E1000168
Reserved		91	5B			3	E100016C
Reserved		92	5C			3	E1000170
Reserved		93	5D			3	E1000174
Reserved		94	5E			3	E1000178
Reserved		95	5F			3	E100017C

[3] Testability and diagnostic use only; not for software use in normal operation

Table 2–18 (Cont.): Processor Registers

		Number					
Register Name	Mnemonic	(Dec)	(Hex)	Type	Impl	Cat	I/O Address
Reserved		96	60			3	E1000180
Reserved		97	61			3	E1000184
Reserved		98	62			3	E1000188
Reserved		99	63			3	E100018C
Reserved for VM		100	64			3	E1000190
Reserved for VM		101	65			3	E1000194
Reserved for VM		102	66			3	E1000198
Reserved		103	67			3	E100019C
Reserved		104	68			3	E10001A0
Reserved		105	69			3	E10001A4
Reserved		106	6A			3	E10001A8
Reserved		107	6B			3	E10001AC
Reserved		108	6C			3	E10001B0
Reserved		109	6D			3	E10001B4
Reserved		110	6E			3	E10001B8
Reserved		111	6F			3	E10001BC
Reserved		112	70			3	E10001C0
Reserved		113	71			3	E10001C4
Reserved		114	72			3	E10001C8
Reserved		115	73			3	E10001CC
Reserved		116	74			3	E10001D0
Reserved		117	75			3	E10001D4
Reserved		118	76			3	E10001D8
Reserved		119	77			3	E10001DC
Reserved for Ebox		120	78			2-6	E10001E0
Reserved for Ebox		121	79			2-6	E10001E4
Interrupt System Status Register[3]	INTSYS	122	7A	RW	NVAX	2-1	
Performance Monitoring Facility Count	PMFCNT	123	7B	RW	NVAX	2-1	
Patchable Control Store Control Register[3]	PCSCR	124	7C	RW	NVAX	2-1	
Ebox Control Register	ECR	125	7D	RW	NVAX	2-1	
Mbox TB Tag Fill[3]	MTBTAG	126	7E	W	NVAX	2-1	
Mbox TB PTE Fill[3]	MTBPTE	127	7F	W	NVAX	2-1	

[3] Testability and diagnostic use only; not for software use in normal operation

Table 2–18 (Cont.): Processor Registers

		Number					
Register Name	Mnemonic	(Dec)	(Hex)	Type	Impl	Cat	I/O Addre
Reserved for Vectors		128	80			3	E1000230
Reserved for Vectors		129	81			3	E1000230
Reserved for Vectors		130	82			3	E1000230
Reserved for Vectors		131	83			3	E1000230
Reserved for Vectors		132	84			3	E1000230
Reserved for Vectors		133	85			3	E1000230
Reserved for Vectors		134	86			3	E1000230
Reserved for Vectors		135	87			3	E1000230
Reserved for Vectors		136	88			3	E1000230
Reserved for Vectors		137	89			3	E1000230
Reserved for Vectors		138	8A			3	E1000230
Reserved for Vectors		139	8B			3	E1000230
Reserved for Vectors		140	8C			3	E1000230
Reserved for Vectors		141	8D			3	E1000234
Reserved for Vectors		142	8E			3	E1000238
Reserved for Vectors		143	8F			3	E100023C
Vector Processor Status Register	VPSR	144	90	RW	Vector	3	E1000240
Vector Arithmetic Exception Register	VAER	145	91	R	Vector	3	E1000244
Vector Memory Activity Register	VMAC	146	92	R	Vector	3	E1000248
Vector Trans. Buffer Invalidate All	VTBIA	147	93	W	Vector	3	E100024C
Reserved for Vectors		148	94			3	E1000250
Reserved for Vectors		149	95			3	E1000254
Reserved for Vectors		150	96			3	E1000258
Reserved for Vectors		151	97			3	E100025C
Reserved for Vectors		152	98			3	E1000260
Reserved for Vectors		153	99			3	E1000264
Reserved for Vectors		154	9A			3	E1000268
Reserved for Vectors		155	9B			3	E100026C
Reserved for Vectors		156	9C			3	E1000270
Reserved for Vectors		157	9D			3	E1000274
Reserved for Vectors		158	9E			3	E1000278
Reserved for Vectors		159	9F			3	E100027C

Table 2–18 (Cont.): Processor Registers

Register Name	Mnemonic	Number (Dec)	(Hex)	Type	Impl	Cat	I/O Address
Cbox Control Register	CCTL	160	A0	RW	NVAX	2-5	
Reserved for Cbox		161	A1		NVAX	2-6	
Bcache Data ECC	BCDECC	162	A2	W	NVAX	2-5	
Bcache Error Tag Status	BCETSTS	163	A3	RW	NVAX	2-5	
Bcache Error Tag Index	BCETIDX	164	A4	R	NVAX	2-5	
Bcache Error Tag	BCETAG	165	A5	R	NVAX	2-5	
Bcache Error Data Status	BCEDSTS	166	A6	RW	NVAX	2-5	
Bcache Error Data Index	BCEDIDX	167	A7	R	NVAX	2-5	
Bcache Error ECC	BCEDECC	168	A8	R	NVAX	2-5	
Reserved for Cbox		169	A9		NVAX	2-6	
Reserved for Cbox		170	AA		NVAX	2-6	
Fill Error Address	CEFADR	171	AB	R	NVAX	2-5	
Fill Error Status	CEFSTS	172	AC	RW	NVAX	2-5	
Reserved for Cbox		173	AD		NVAX	2-6	
NDAL Error Status	NESTS	174	AE	RW	NVAX	2-5	
Reserved for Cbox		175	AF		NVAX	2-6	
NDAL Error Output Address	NEOADR	176	B0	R	NVAX	2-5	
Reserved for Cbox		177	B1		NVAX	2-6	
NDAL Error Output Command	NEOCMD	178	B2	R	NVAX	2-5	
Reserved for Cbox		179	B3		NVAX	2-6	
NDAL Error Data High	NEDATHI	180	B4	R	NVAX	2-5	
Reserved for Cbox		181	B5		NVAX	2-6	
NDAL Error Data Low	NEDATLO	182	B6	R	NVAX	2-5	
Reserved for Cbox		183	B7		NVAX	2-6	
NDAL Error Input Command	NEICMD	184	B8	R	NVAX	2-5	
Reserved for Cbox		185	B9		NVAX	2-6	
Reserved for Cbox		186	BA		NVAX	2-6	
Reserved for Cbox		187	BB		NVAX	2-6	
Reserved for Cbox		188	BC		NVAX	2-6	
Reserved for Cbox		189	BD		NVAX	2-6	
Reserved for Cbox		190	BE		NVAX	2-6	
Reserved for Cbox		191	BF		NVAX	2-6	

Table 2–18 (Cont.): Processor Registers

		Number					
Register Name	Mnemonic	(Dec)	(Hex)	Type	Impl	Cat	I/O Addr
Reserved		192	C0			3	E1000300
Reserved		193	C1			3	E1000304
Reserved		194	C2			3	E1000308
Reserved		195	C3			3	E100030C
Reserved		196	C4			3	E1000310
Reserved		197	C5			3	E1000314
Reserved		198	C6			3	E1000318
Reserved		199	C7			3	E100031C
Reserved		200	C8			3	E1000320
Reserved		201	C9			3	E1000324
Reserved		202	CA			3	E1000328
Reserved		203	CB			3	E100032C
Reserved		204	CC			3	E1000330
Reserved		205	CD			3	E1000334
Reserved		206	CE			3	E1000338
Reserved		207	CF			3	E100033C
VIC Memory Address Register	VMAR	208	D0	RW	NVAX	2-5	
VIC Tag Register	VTAG	209	D1	RW	NVAX	2-5	
VIC Data Register	VDATA	210	D2	RW	NVAX	2-5	
Ibox Control and Status Register	ICSR	211	D3	RW	NVAX	2-5	
Ibox Branch Prediction Control Register[3]	BPCR	212	D4	RW	NVAX	2-5	
Reserved for Ibox		213	D5		NVAX	2-6	
Ibox Backup PC[4]	BPC	214	D6	R	NVAX	2-5	
Ibox Backup PC with RLOG Unwind[4]	BPCUNW	215	D7	R	NVAX	2-5	
Reserved for Ibox		216	D8		NVAX	2-6	
Reserved for Ibox		217	D9		NVAX	2-6	
Reserved for Ibox		218	DA		NVAX	2-6	
Reserved for Ibox		219	DB		NVAX	2-6	
Reserved for Ibox		220	DC		NVAX	2-6	
Reserved for Ibox		221	DD		NVAX	2-6	
Reserved for Ibox		222	DE		NVAX	2-6	
Reserved for Ibox		223	DF		NVAX	2-6	

[3] Testability and diagnostic use only; not for software use in normal operation

[4] Chip test use only; not for software use

Table 2–18 (Cont.): Processor Registers

		Number					
Register Name	**Mnemonic**	**(Dec)**	**(Hex)**	**Type**	**Impl**	**Cat**	**I/O Address**
Mbox P0 Base Register[3]	MP0BR	224	E0	RW	NVAX	2-5	
Mbox P0 Length Register[3]	MP0LR	225	E1	RW	NVAX	2-5	
Mbox P1 Base Register[3]	MP1BR	226	E2	RW	NVAX	2-5	
Mbox P1 Length Register[3]	MP1LR	227	E3	RW	NVAX	2-5	
Mbox System Base Register[3]	MSBR	228	E4	RW	NVAX	2-5	
Mbox System Length Register[3]	MSLR	229	E5	RW	NVAX	2-5	
Mbox Memory Management Enable[3]	MMAPEN	230	E6	RW	NVAX	2-5	
Mbox Physical Address Mode	PAMODE	231	E7	RW	NVAX	2-5	
Mbox MME Address	MMEADR	232	E8	R	NVAX	2-5	
Mbox MME PTE Address	MMEPTE	233	E9	R	NVAX	2-5	
Mbox MME Status	MMESTS	234	EA	R	NVAX	2-5	
Reserved for Mbox		235	EB		NVAX	2-6	
Mbox TB Parity Address	TBADR	236	EC	R	NVAX	2-5	
Mbox TB Parity Status	TBSTS	237	ED	RW	NVAX	2-5	
Reserved for Mbox		238	EE		NVAX	2-6	
Reserved for Mbox		239	EF		NVAX	2-6	
Reserved for Mbox		240	F0		NVAX	2-6	
Reserved for Mbox		241	F1		NVAX	2-6	
Mbox Pcache Parity Address	PCADR	242	F2	R	NVAX	2-5	
Reserved for Mbox		243	F3		NVAX	2-6	
Mbox Pcache Status	PCSTS	244	F4	RW	NVAX	2-5	
Reserved for Mbox		245	F5		NVAX	2-6	
Reserved for Mbox		246	F6		NVAX	2-6	
Reserved for Mbox		247	F7		NVAX	2-6	
Mbox Pcache Control	PCCTL	248	F8	RW	NVAX	2-5	
Reserved for Mbox		249	F9		NVAX	2-6	
Reserved for Mbox		250	FA		NVAX	2-6	
Reserved for Mbox		251	FB		NVAX	2-6	
Reserved for Mbox		252	FC		NVAX	2-6	
Reserved for Mbox		253	FD		NVAX	2-6	
Reserved for Mbox		254	FE		NVAX	2-6	
Reserved for Mbox		255	FF		NVAX	2-6	

[3] Testability and diagnostic use only; not for software use in normal operation

Table 2–18 (Cont.): Processor Registers

		Number					
Register Name	Mnemonic	(Dec)	(Hex)	Type	Impl	Cat	I/O Addre
Unimplemented			100– 00FFFFFF			3	
See Table 2–17			01000000– FFFFFFFF			2	

Type:

R = Read-only register
RW = Read-write register
W = Write-only register

Impl(emented):

NVAX = Implemented in the NVAX CPU chip
System = Implemented in the system environment
Vector = Implemented in the optional vector unit or its NDAL interface

Cat(egory), *class-subclass*, where:
class is one of:

1 = Implemented as per DEC standard 032
2 = NVAX-specific implementation which is unique or different from the DEC standard 032 implementation
3 = Not implemented internally; converted to I/O space read or write and passed to system environment

subclass is one of:

1 = Processed as appropriate by Ebox microcode
2 = Converted to Mbox IPR number and processed via internal IPR command
3 = Processed by internal IPR command, then converted to I/O space read or write and passed to system environment
4 = If virtual machine option is implemented, processed as in 1, otherwise as in 3
5 = Processed by internal IPR command
6 = May be block decoded; reference causes UNDEFINED behavior
7 = Full interval timer may be implemented in the system environment. Subset ICCS is implemented in NVAX CPU chip

2.13 I/O space Addresses

As noted above, processor registers that are not implemented on the NVAX CPU chip are converted to I/O space reads or writes. Most of these IPRs are optional and may be implemented or not, as dictated by the needs of the system environment. The I/O space registers that must be implemented by the system environment are shown in Table 2–19.

Table 2–19: I/O Space Registers

I/O Space Address (Hex)	Type	Definition
E0040000	RO	Powerup boot ROM address from which the first instruction is fetched.
E1000100	RO	Interrupt acknowledge for an IPL 14 (hex) interrupt requested via the **IRQ_L<0>** pin.
E1000104	RO	Interrupt acknowledge for an IPL 15 (hex) interrupt requested via the **IRQ_L<1>** pin.
E1000108	RO	Interrupt acknowledge for an IPL 16 (hex) interrupt requested via the **IRQ_L<2>** pin.
E100010C	RO	Interrupt acknowledge for an IPL 17 (hex) interrupt requested via the **IRQ_L<3>** pin.
E1000110	RW	Location which invokes a write buffer flush in the system environment. When this location is read, the CPU is waiting for confirmation that the flush has completed. The returned data is ignored.

2.14 Revision History

Table 2–20: Revision History

Who	When	Description of change
Mike Uhler	06-Mar-1989	Release for external review.
Mike Uhler	15-Dec-1989	Update for second-pass release.
Mike Uhler	20-Jul-1990	Update to reflect implementation.
Mike Uhler	04-Dec-1990	Update after pass 1 PG.

Chapter 3

NVAX Chip Interface

3.1 Introduction

The NVAX chip communicates through five interfaces: the NDAL (NVAX data-address lines), the backup cache interface, the interrupt lines, the clocking interface, and the test interface.

This chapter begins by listing all the NVAX pins and giving a brief description of each. The rest of the chapter describes the NDAL protocol in detail. The other interfaces are described as follows: the backup cache interfaces in Chapter 13, the interrupt lines in Chapter 10, the test interface in Chapter 19, and the clocking interface in Chapter 17.

The NDAL is a 64-bit pended bidirectional bus which is used by the NVAX CPU to communicate with the system environment. The NDAL cycle time is three times longer than the NVAX CPU cycle time. The NVAX CPU cycle time is targeted to 14ns, making the NDAL cycle time 42 ns. Binned CPU parts may run at 10ns, resulting in an NDAL cycle time of 30ns. The NDAL supports up to four (4) nodes with a maximum of one (1) NVAX CPU. In this spec, these four nodes are referred to as CPU (NVAX), IO1_NODE, IO2_NODE, and the memory interface.

The NVAX CPU contains a writethrough primary cache and a writeback backup cache. The NDAL is designed to support the writeback cache and cache coherency in a multiprocessor system.

NOTE

IMPORTANT INFORMATION REGARDING THE NVAX CHIP INTERFACE IS ALSO CONTAINED IN Chapter 10 (The Interrupt Section), Chapter 13 (The Cbox), Chapter 17 (Chip Clocking), AND Chapter 19 (Testability Micro-Architecture). THE READER MUST CONSULT THOSE CHAPTERS IN ORDER TO OBTAIN COMPLETE INFORMATION.

3.2 NVAX CPU pinout

The NVAX CPU chip contains the pins listed in Table 3–1. Following the table, each pin is described in more detail.

Table 3–1: NVAX CPU pinout

Pin	I/O[1]	Type[2]	Function	Number	Running Total
NDAL SIGNALS (80 total)[3]					
P%CPU_REQ_L	O	SS,1D1R	NVAX Request	1	1
P%CPU_HOLD_L	O	SS,1D1R	NVAX Hold	1	2
P%CPU_SUPPRESS_L	O	SS,1D1R	NVAX Suppress	1	3
P%CPU_GRANT_L	I	SS,1D1R	NVAX Grant	1	4
P%CPU_WB_ONLY_L	I	SS,1D1R	Writeback Only	1	5
P%NDAL_H<63:0>	IO	T,4D4R	Data/Address Lines	64	69
P%CMD_H<3:0>	IO	T,4D4R	Command	4	73
P%ID_H<2:0>	IO	T,4D4R	Node Identification Lines	3	76
P%PARITY_H<2:0>	IO	T,4D4R	NDAL Parity	3	79
P%ACK_L	IO	OD,4D4R	Acknowledge	1	80
CLOCKS (15 total)[4]					
P%OSC_H	I	SS,1D1R	Oscillator, High Asserted	1	81
P%OSC_L	I	SS,1D1R	Oscillator, Low Asserted	1	82
P%OSC_TC1_H	I	SS,1D1R	Test Clock/Timeout Clock	1	83
P%OSC_TC2_H	I	SS,1D1R	Test Clock	1	84
P%OSC_TEST_H	I	SS,1D1R	Test Clock Control	1	85
P%PHI12_OUT_H	O	SS,1D4R	NDAL PHI12, Driven	1	86
P%PHI23_OUT_H	O	SS,1D4R	NDAL PHI23, Driven	1	87
P%PHI34_OUT_H	O	SS,1D4R	NDAL PHI34, Driven	1	88
P%PHI41_OUT_H	O	SS,1D4R	NDAL PHI41, Driven	1	89
P%PHI12_IN_H	I	SS,1D4R	NDAL PHI12, Received	1	90
P%PHI23_IN_H	I	SS,1D4R	NDAL PHI23, Received	1	91
P%PHI34_IN_H	I	SS,1D4R	NDAL PHI34, Received	1	92
P%PHI41_IN_H	I	SS,1D4R	NDAL PHI41, Received	1	93
P%ASYNC_RESET_L	I	SS,1D1R	Reset Input to NVAX	1	94
P%SYS_RESET_L	O	SS,1D3R	Reset Output to System	1	95
INTERRUPT AND ERROR SIGNALS (10 total)[5]					
P%MACHINE_CHECK_H	O	SS,1D1R	Machine Check	1	96
P%IRQ_L<3:0>	I	OD,3D1R	Interrupt Request Lines	4	100
P%H_ERR_L	I	OD,3D1R	Hard (unrecoverable) Error	1	101
P%S_ERR_L	I	OD,3D1R	Soft (recoverable) Error	1	102
P%INT_TIM_L	I	SS,1D1R	Interval Timer Request	1	103
P%PWRFL_L	I	SS,1D1R	Power Fail	1	104
P%HALT_L	I	SS,1D1R	Halt	1	105

					Running
Pin	I/O	Type	Function	Number	Total
BACKUP CACHE SIGNALS (133 total)[6]					
P%TS_INDEX_H<20:5>	O	SS,1D6R	Tag Store Index Lines	16	121
P%TS_OE_L	O	SS,1D6R	Tag Store Output Enable	1	122
P%TS_WE_L	O	SS,1D6R	Tag Store Write Enable	1	123
P%TS_TAG_H<31:17>	IO	T,7D7R	Tag Store Tag	15	138
P%TS_ECC_H<5:0>	IO	T,7D7R	Tag Store ECC	6	144
P%TS_OWNED_H	IO	T,7D7R	Tag Store Owned Bit	1	145
P%TS_VALID_H	IO	T,7D7R	Tag Store Valid Bit	1	146
P%DR_INDEX_H<20:3>	O	SS,1D18R	Data RAM Index Lines	18	164
P%DR_OE_L	O	SS,1D18R	Data RAM Output Enable	1	165
P%DR_WE_L	O	SS,1D18R	Data RAM Write Enable	1	166
P%DR_DATA_H<63:0>	IO	T,19D19R	Data RAM Data Lines	64	230
P%DR_ECC_H<7:0>	IO	T,19D19R	Data RAM ECC	8	238
TEST SIGNALS (23 total)[7]					
P%TEST_DATA_H	I	SS,1D1R	Test data input for microcode use.	1	239
P%TEST_STROBE_H	I	SS,1D1R	Test strobe for microcode use.	1	240
P%DISABLE_OUT_L	I	SS,1D1R	Disable NVAX Outputs	1	241
P%TEMP_H	O	SS,1D1R	NVAX Temperature Output	1	242
P%TMS_H	I	SS,1D1R	JTAG Test Mode Select	1	243
P%TCK_H	I	SS,1D1R	JTAG Test Clock	1	244
P%TDI_H	I	SS,1D1R	JTAG Serial Test Data Input	1	245
P%TDO_H	O	SS,1D2R	JTAG Serial Test Data Output	1	246
P%PP_CMD_H<2:0>	I	SS,1D1R	Parallel Test Port Command	3	249
P%PP_DATA_H<11:0>	O	T,2D2R	Parallel Test Port Data	12	261

[1] Indicates whether the pin is an NVAX CPU Input, Output, or Input/Output pin.

[2] Single Source is denoted by SS, Tristate by T, Open Drain by OD; #D indicates the maximum number of drivers and #R indicates the maximum number of receivers expected on the board.

[3] These pins are discussed in detail in this chapter.

[4] These pins are discussed in detail in Chapter 17

[5] These pins are discussed in detail in Chapter 10

[6] These pins are discussed in detail in Chapter 13

[7] These pins are discussed in detail in Chapter 19

3.2.1 NDAL Signals and Timing

The functionality of the NDAL pins is described in detail in Section 3.3. The timing of the pins is shown in Figure 3–1, and the AC specs are given in Table 3–2.

NOTE

The timing of the NDAL signals is given relative to the NDAL clocks which are received by NVAX: **P%PHI12_IN_H**, **P%PHI23_IN_H**, **P%PHI34_IN_H**, and **P%PHI41_IN_H**. NVAX drivers were designed to meet this timing, taking the NDAL clock skew into account. (NDAL clock skew is covered in Chapter 17.) NVAX expects to receive signals which have been designed taking the clock skew into account; NVAX receivers account for no clock skew.

Figure 3–1: NDAL Pin Timing Relative to the NDAL CLOCKS

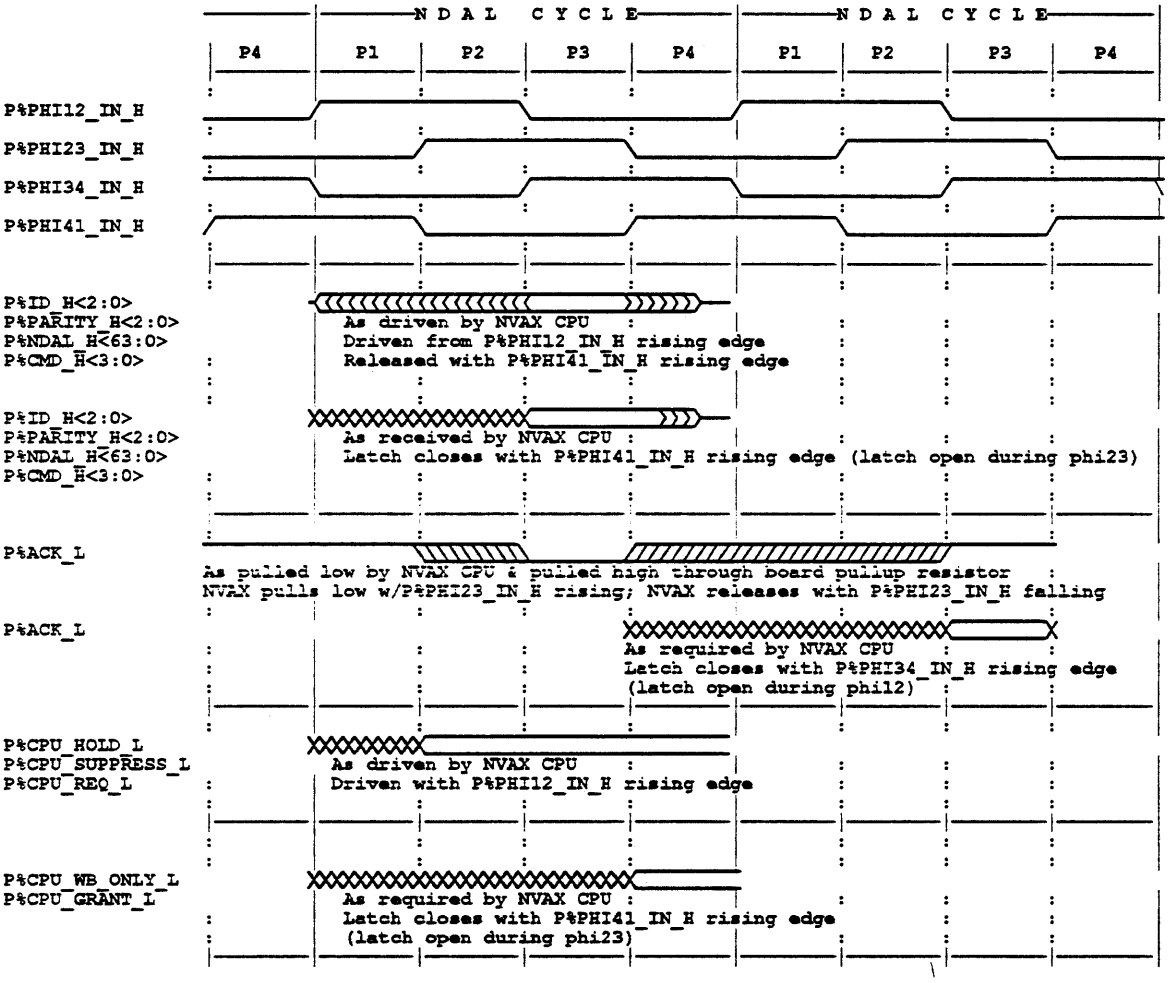

NDAL CYCLE
NDAL CYCLE
P4
P1
P2
P3
P4
P1
P2
P3
P4
P%PHI12_IN_H
P%PHI23_IN_H
P%PHI34_IN_H
P%PHI41_IN_H
P%ID_H<2:0>
P%PARITY_H<2:0>
P%NDAL_H<63:0>
P%CMD_H<3:0>
As driven by NVAX CPU
Driven from P%PHI12_IN_H rising edge
Released with P%PHI41_IN_H rising edge
P%ID_H<2:0>
P%PARITY_H<2:0>
P%NDAL_H<63:0>
P%CMD_H<3:0>
As received by NVAX CPU
Latch closes with P%PHI41_IN_H rising edge (latch open during phi23)
P%ACK_L
As pulled low by NVAX CPU & pulled high through board pullup resistor
NVAX pulls low w/P%PHI23_IN_H rising; NVAX releases with P%PHI23_IN_H falling
P%ACK_L
As required by NVAX CPU
Latch closes with P%PHI34_IN_H rising edge
(latch open during phi12)
P%CPU_HOLD_L
P%CPU_SUPPRESS_L
P%CPU_REQ_L
As driven by NVAX CPU
Driven with P%PHI12_IN_H rising edge
P%CPU_WB_ONLY_L
P%CPU_GRANT_L
As required by NVAX CPU
Latch closes with P%PHI41_IN_H rising edge
(latch open during phi23)

Table 3–2: NDAL AC timing specs

Input Pin	Setup Time[1]	Hold Time
P%NDAL_H<63:0>	1 phase to P%PHI41_IN_H R	P%PHI41_IN_H R + 2ns[2]
P%CMD_H<3:0>	"	"
P%ID_H<2:0>	"	"
P%PARITY_H<2:0>	"	"
P%ACK_L	0 ns to P%PHI34_IN_H R	P%PHI34_IN_H R + 1 phase
P%CPU_WB_ONLY_L	0 ns to P%PHI41_IN_H R	P%PHI41_IN_H R + 1 phase
P%CPU_GRANT_L	"	"
Output Pin	**Drive Time**	**Tristate Time**
P%NDAL_H<63:0>	P%PHI12_IN_H R + 2 phases	P%PHI41_IN_H R + 1 phase
P%CMD_H<3:0>	"	"
P%ID_H<2:0>	"	"
P%PARITY_H<2:0>	"	"
P%ACK_L	P%PHI23_IN_H R + 1 phase (low transition), P%PHI23_IN_H F + 3 phases(high transition)[3]	-
P%CPU_HOLD_L	P%PHI12_IN_H R + 1 phase	-
P%CPU_SUPPRESS_L	"	-
P%CPU_REQ_L	"	-

[1] R means the rising edge of the clock is used; F means the falling edge of the clock is used.

[2] The 2ns hold time requirement on the NDAL is as follows: the data does not have to be actively driven for this amount of time if the driver ensures that the values will be capacitively held on the bus for 2ns past the phi4 rising edge.

[3] P%ACK_L is pulled up through a resistor in the system; the same must be done on the test load board.

3.2.1.1 P%CPU_REQ_L

NVAX asserts **P%CPU_REQ_L** to request the NDAL for the following cycle. **P%CPU_REQ_L** is a unidirectional signal from NVAX to the arbiter.

3.2.1.2 P%CPU_HOLD_L

The NVAX CPU asserts **P%CPU_HOLD_L** in order to drive the NDAL on consecutive cycles.

3.2.1.3 P%CPU_SUPPRESS_L

NVAX asserts **P%CPU_SUPPRESS_L** in order to suppress new NDAL transactions. While **P%CPU_SUPPRESS_L** is asserted, only fills and writebacks are allowed to proceed from non-CPU nodes.

3.2.1.4 P%CPU_GRANT_L

P%CPU_GRANT_L is asserted to notify NVAX that it must drive the NDAL during the following cycle.

3.2.1.5 P%CPU_WB_ONLY_L

When the system asserts **P%CPU_WB_ONLY_L**, NVAX only issues WDISOWN or NOPcommands.

3.2.1.6 P%NDAL_H<63:0>

NVAX uses **P%NDAL_H<63:0>** to transfer address and data information to and from the system.

3.2.1.7 P%CMD_H<3:0>

The **P%CMD_H<3:0>** lines contain the NDAL command during any given cycle. NVAX drives and receives these lines.

3.2.1.8 P%ID_H<2:0>

NVAX drives and receives **P%ID_H<2:0>**, which contain the node identification number for every cycle. These lines identify which node is driving the NDAL or which node is to receive the NDAL, depending upon the current command.

3.2.1.9 P%PARITY_H<2:0>

NVAX drives and receives **P%PARITY_H<2:0>**, which contains parity computed over **P%NDAL_H<63:0>**, **P%CMD_H<3:0>** and **P%ID_H<2:0>** during every NDAL cycle.

3.2.1.10 P%ACK_L

NVAX asserts **P%ACK_L** when it has received a fill data cycle. NVAX receives **P%ACK_L** as an acknowledgement that its outgoing cycle was successfully received. It also receives **P%ACK_L** for cycles which it did not drive on the NDAL, as a way of detecting inconsistent parity errors. An inconsistent parity error is where NVAX detects a parity error on the NDAL and also notices that **P%ACK_L** was asserted for that cycle.

P%ACK_L is an open drain signal which is pulled high (deasserted) by an external resistor on the board.

3.2.2 Clocking signals

The NVAX CPU chip generates four two-phase clocks which are distributed to the system. These clocks are also distributed back to itself, which minimizes skew between NVAX and the other chips on the NDAL. Each NDAL cycle is three CPU cycles long.

The clocking signals are described in detail in Chapter 17.

3.2.2.1 P%OSC_H, P%OSC_L

P%OSC_H and **P%OSC_L** are complementary oscillator inputs to NVAX. They are used to generate on-chip clocks and system clocks. When **P%OSC_TEST_H** is deasserted, **P%OSC_H** and **P%OSC_L** are used to generate NVAX clocks.

3.2.2.2 P%OSC_TC1_H, P%OSC_TC2_H

P%OSC_TC1_H and **P%OSC_TC2_H** are oscillator inputs to NVAX for use during testing only. When **P%OSC_TEST_H** is asserted, **P%OSC_TC1_H** and **P%OSC_TC2_H** are used to generate NVAX clocks.

P%OSC_TC1_H and **P%OSC_TC2_H** are 90 degrees out of phase with each other, and are XOR'd internally to produce an internal clock which runs at twice the speed. This allows NVAX to run at full speed while the input clocks are running at half speed.

P%OSC_TC1_H is also used as an input to the Ebox base timeout counter as an alternate clock for the timeout counter. Normally, the base counter is run from the internal NVAX clock; if the system designer wants to lengthen the timeout values used by NVAX, the base counter may be configured to run from **P%OSC_TC1_H** instead. **P%OSC_TC1_H** is synchronized to the internal NVAX clocks in order to be used for this purpose.

3.2.2.3 P%OSC_TEST_H

P%OSC_TEST_H is a control pin which determines which oscillator inputs are used by the clock generators. When **P%OSC_TEST_H** is deasserted, **P%OSC_H** and **P%OSC_L** are used; when **P%OSC_TEST_H** is asserted, **P%OSC_TC1_H** and **P%OSC_TC2_H** are used.

3.2.2.4 P%PHI12_OUT_H, P%PHI23_OUT_H, P%PHI34_OUT_H, P%PHI41_OUT_H

These two-phase overlapping clocks are driven from the NVAX chip to all nodes on the NDAL, including back to NVAX itself.

3.2.2.5 P%PHI12_IN_H, P%PHI23_IN_H, P%PHI34_IN_H, P%PHI41_IN_H

These NVAX pins are used to receive the NDAL clocks, which are driven from **P%PHI12_OUT_H**, **P%PHI23_OUT_H**, **P%PHI34_OUT_H**, and **P%PHI41_OUT_H**.

3.2.2.6 P%ASYNC_RESET_L

P%ASYNC_RESET_L is an asynchronous input to NVAX which is used to generate an internal reset signal as well as **P%SYS_RESET_L**.

3.2.2.7 P%SYS_RESET_L

NVAX drives **P%SYS_RESET_L** to notify all NDAL receivers to reset. It is deasserted synchronously with the NDAL clocks.

3.2.3 Interrupt and Error Signals

The interrupt and error signals are described in detail in Chapter 10.

3.2.3.1 P%MACHINE_CHECK_H

The assertion of **P%MACHINE_CHECK_H** indicates that the CPU is in a machine check sequence. This signal may be wired to an LED on the board. (The pin is not able to drive the LED directly.) It will flicker during a normal machine check. If the CPU never comes out of machine check, the LED will stay lit and indicates to Field Service that the board needs to be replaced.

3.2.3.2 P%IRQ_L<3:0>

The **P%IRQ_L<3:0>** lines provide a general-purpose interrupt request facility to interrupt the NVAX CPU. These four external interrupt request lines correspond to interrupt requests at IPLs 17, 16, 15, and 14 (hex). **P%IRQ_L<3>** corresponds to IPL 17, **P%IRQ_L<2>** corresponds to IPL 16, **P%IRQ_L<1>** corresponds to IPL 15, and **P%IRQ_L<0>** corresponds to IPL 14. These lines are level-sensitive, NOT edge sensitive. Once a node asserts its interrupt line, it should keep it asserted until NVAX services the request.

P%IRQ_L<3:0> are asynchronous inputs to NVAX and are not expected to operate with any fixed relationship to the NDAL timing.

3.2.3.3 P%H_ERR_L

P%H_ERR_L is used to notify NVAX of an error condition in the system which has corrupted machine state. These errors usually cannot be corrected by any retry mechanism.

If at all possible, NDAL errors should be reported using the transaction level error reporting mechanisms (not asserting **P%ACK_L** or using the Read Data Error command). If this is not possible, **P%H_ERR_L** or **P%S_ERR_L** may be used. When **P%H_ERR_L** is asserted, NVAX will take a Hard Error Interrupt at IPL 1D (hex).

P%H_ERR_L is an asynchronous input to NVAX and is not expected to operate with any fixed relationship to the NDAL timing.

3.2.3.4 P%S_ERR_L

The assertion of **P%S_ERR_L** indicates that an error which did not affect instruction execution has been detected in the system environment. For example, if an NDAL node uses the BADWDATA because of an uncorrectable error in its cache, it would also assert **P%S_ERR_L** to notify NVAX of the event. When it recognizes the assertion of **P%S_ERR_L**, NVAX takes a Soft Error Interrupt at IPL 1A (hex).

P%S_ERR_L is an asynchronous input to NVAX and is not expected to operate with any fixed relationship to the NDAL timing.

3.2.3.5 P%INT_TIM_L

The assertion of **P%INT_TIM_L** indicates that the interval timer period has expired.

P%INT_TIM_L is an asynchronous input to NVAX and is not expected to operate with any fixed relationship to the NDAL timing.

3.2.3.6 P%PWRFL_L

The assertion of **P%PWRFL_L** informs the CPU of an impending power failure.

P%PWRFL_L is an asynchronous input to NVAX and is not expected to operate with any fixed relationship to the NDAL timing.

3.2.3.7 P%HALT_L

The assertion of **P%HALT_L** causes the CPU to enter the console at IPL 1F (hex) at the next macroinstruction boundary.

P%HALT_L is an asynchronous input to NVAX and is not expected to operate with any fixed relationship to the NDAL timing.

3.2.4 Cache interface signals

These pins are described in detail in Chapter 13. The timing of the pins is shown in Figure 3–2.

NOTE

> The timing of the Bcache interface signals is given relative to the INTERNAL NVAX clocks.

3.2.4.1 P%TS_INDEX_H<20:5>

P%TS_INDEX_H<20:5> drive the address lines of the backup cache tag RAMs, thus indexing into one row of the tag store.

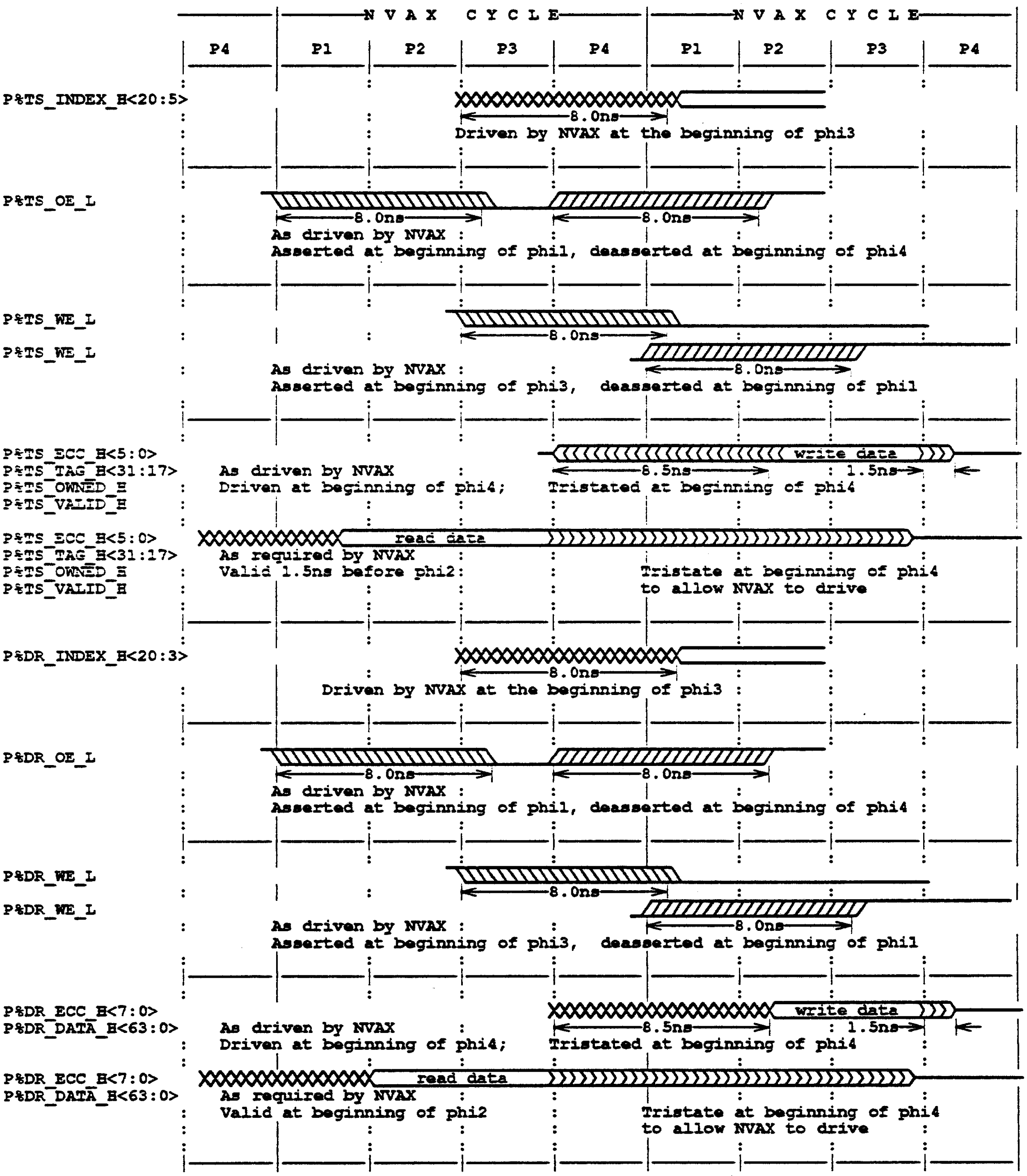
3.2.4.2 P%TS_OE_L

This pin is connected to the output enable pins of the backup cache tag store RAMs. When NVAX asserts **P%TS_OE_L**, the RAMs are enabled to drive **P%TS_TAG_H<31:17>**, **P%TS_VALID_H**, **P%TS_OWNED_H**, and **P%TS_ECC_H<5:0>**.

3.2.4.3 P%TS_WE_L

This pin is connected to the write enable pins of the backup cache tag store RAMs. When NVAX asserts **P%TS_WE_L**, the RAMs are enabled to write the information on **P%TS_TAG_H<31:17>**, **P%TS_VALID_H**, **P%TS_OWNED_H**, and **P%TS_ECC_H<5:0>**, which NVAX drives when **P%TS_WE_L** is asserted.

Figure 3–2: Bcache Pin Timing Relative to INTERNAL NVAX Clocks (14ns system)



NOTE: All drive times are shown as simulated in the XNP board environment with typical (14ns) NVAX parts; drive times in other environments and with non-typical NVAX parts will differ. The diagram assumes a 14-ns NVAX cycle.

3.2.4.4 P%TS_TAG_H<31:17>

P%TS_TAG_H<31:17> carry the tag which is written to and read from the backup cache tag store. Each of these pads is built with an internal resistor so that if the tag bit is not used in a particular system, the pin value as seen by the Cbox is 0. For example, a machine which runs only in 30-bit mode does not need to connect **P%TS_TAG_H<31:29>** to the backup cache.

3.2.4.5 P%TS_ECC_H<5:0>

P%TS_ECC_H<5:0 > carry the error correcting code which is written to and read from the backup cache tag store.

3.2.4.6 P%TS_OWNED_H

P%TS_OWNED_H carries the OWNED bit which is written to and read from the backup cache tag store.

3.2.4.7 P%TS_VALID_H

P%TS_VALID_H carries the VALID bit which is written to and read from the backup cache tag store.

3.2.4.8 P%DR_INDEX_H<20:3>

P%DR_INDEX_H<20:3> drive the address lines of the backup cache data RAMs, thus indexing into one row (one quadword) of the cache.

3.2.4.9 P%DR_OE_L

This pin is connected to the output enable pins of the backup cache data RAMs. When NVAX asserts **P%DR_OE_L**, the RAMs are enabled to drive **P%DR_DATA_H<63:0>** and **P%DR_ECC_H<7:0>**.

3.2.4.10 P%DR_WE_L

This pin is connected to the write enable pins of the backup cache data RAMs. When NVAX asserts **P%DR_WE_L**, the RAMs are enabled to write the information on **P%DR_DATA_H<63:0>** and **P%DR_ECC_H<7:0>**, which NVAX drives when **P%DR_WE_L** is asserted.

3.2.4.11 P%DR_DATA_H<63:0>

P%DR_DATA_H<63:0> carry the cache data which is written to and read from the backup cache.

3.2.4.12 P%DR_ECC_H<7:0>

P%DR_ECC_H<7:0 > carry the error correcting code which is written to and read from the backup cache data RAMs.

3.2.5 Test Pins

These pins are covered in more detail in Chapter 19.

3.2.5.1 P%TEST_DATA_H

TEST_DATA_H is an asynchronous input pin which may be used by microcode. It is pulled high internally so that if it is not used, it does not have to be connected on the board.

3.2.5.2 P%TEST_STROBE_H

TEST_STROBE_H is an asynchronous input pin which may be used by microcode. It is pulled high internally so that if it is not used, it does not have to be connected on the board.

3.2.5.3 P%DISABLE_OUT_L

When P%DISABLE_OUT_L is asserted, NVAX does not drive any of its Input/Output or Output pins, including the NDAL clock outputs (P%PHI12_OUT_H, P%PHI23_OUT_H, P%PHI34_OUT_H and P%PHI41_OUT_H).

This functionality is used only during test.

3.2.5.4 P%TEMP_H

P%TEMP_H is an output pin to be used in test to determine when the NVAX CPU chip is at thermal equilibrium. The voltage on this pin will vary between VDD_I and VSS_I, depending on chip temperature, but the temperature to voltage transfer function will not be specified.

As the chip heats up the voltage on the pin will fall, and once the chip is at thermal equilibrium the voltage will remain at some value below VDD_I. This voltage will be monitored by the tester, and testing will commence only when the voltage stops changing, indicating that the chip is at thermal equilibrium.

3.2.5.5 P%TMS_H

P%TMS_H is the JTAG test mode select input. It is pulled high by an on-chip resistor when it is not being driven externally.

3.2.5.6 P%TCK_H

P%TCK_H is the JTAG test clock. It is pulled low by an on-chip resistor when it is not being driven externally.

3.2.5.7 P%TDI_H

P%TDI_H is the JTAG serial test data input. It is pulled high by an on-chip resistor when it is not being driven externally.

3.2.5.8 P%TDO_H

P%TDO_H is the JTAG serial test data output.

3.2.5.9 P%PP_CMD_H<2:0>

P%PP_CMD_H<2:0> provides the NVAX parallel port a command indicating the current function of the parallel port.

3.2.5.10 P%PP_DATA_H<11:0>

P%PP_DATA_H<11:0> are output pins for reading test data from NVAX.

3.3 The NDAL

The NDAL is a 64-bit limited length, pended, synchronous bus with centralized arbitration. Several transactions can be in progress at a given time, allowing highly efficient use of bus bandwidth. Arbitration and data transfers occur simultaneously. The bus uses multiplexed data and address lines. The NDAL supports quadword, octaword and hexaword reads and writes to memory and I/O space.

The NDAL supports up to four (4) nodes with a maximum of one (1) NVAX CPU. In this spec, these four nodes are referred to as CPU (NVAX), IO1_NODE, IO2_NODE, and the memory interface.

Thirty nanoseconds is the minimum NDAL cycle time being considered for a binned CPU. Operating at 30ns, the NDAL has a raw bandwidth of 267 Mbytes/second. At 42ns, the NDAL has a raw bandwidth of 190 Mbytes/second. The usable bandwidth, which depends on transaction length, is shown in Table 3–3 and Table 3–4.

Table 3–3: NVAX DAL Bandwidth at 30ns

Operation	Bandwidth
Quadword Read	133.0 Mbytes/sec
Octaword Read	178.0 Mbytes/sec
Hexaword Read	213.0 Mbytes/sec
Quadword Write	133.0 Mbytes/sec
Octaword Write	178.0 Mbytes/sec
Hexaword Write	213.0 Mbytes/sec

Table 3–4: NVAX DAL Bandwidth at 42ns

Operation	Bandwidth
Quadword Read	95.0 Mbytes/sec
Octaword Read	127.0 Mbytes/sec
Hexaword Read	152.0 Mbytes/sec
Quadword Write	95.0 Mbytes/sec
Octaword Write	127.0 Mbytes/sec
Hexaword Write	152.0 Mbytes/sec

Table 3–5 details each NDAL signal. Where All is indicated for Drivers and Receivers, all four possible NDAL nodes drive or receive the signal.

Table 3–5: NDAL Signals

Signal	Type[1]	Drivers	Receivers	Function
Arbitration signals				
P%CPU_REQ_L	SS	NVAX	Arbiter	NVAX requests the bus.
IO1_REQ_L	SS	IO1_NODE	Arbiter	IO1_NODE requests the bus.
IO2_REQ_L	SS	IO2_NODE	Arbiter	IO2_NODE requests the bus.
P%CPU_HOLD_L	SS	NVAX	Arbiter	Extends **P%CPU_GRANT_L**.
IO1_HOLD_L	SS	IO1_NODE	Arbiter	Extends IO1_GRANT.
IO2_HOLD_L	SS	IO2_NODE	Arbiter	Extends IO2_GRANT.
P%CPU_GRANT_L	SS	Arbiter	NVAX	Grants NVAX the bus.
IO1_GRANT_L	SS	Arbiter	IO1_NODE	Grants IO1_NODE the bus.
IO2_GRANT_L	SS	Arbiter	IO2_NODE	Grants IO2_NODE the bus.
P%CPU_SUPPRESS_L	SS	NVAX	Arbiter	Suppresses all but writebacks and fills.
P%CPU_WB_ONLY_L	SS	Arbiter	NVAX	Limits NVAX to doing only Disown Writes or NOPs.
IO1_SUPPRESS_L	SS	IO1_NODE	Arbiter	Suppresses all but writebacks and fills.
IO1_WB_ONLY_L	SS	Arbiter	IO1_NODE	IO1_NODE may only do Disown Writes and fills.
IO2_SUPPRESS_L	SS	IO2_NODE	Arbiter	Suppresses all but writebacks and fills.
IO2_WB_ONLY_L	SS	Arbiter	IO2_NODE	IO2_NODE may only do Disown Writes and fills.
Data, address, and command signals				
P%NDAL_H<63:0>	T	All	All	Multiplexed data and address lines.
P%CMD_H<3:0>	T	All	All	Command being performed this cycle.
P%ID_H<2:0>	T	All	All	Commander identification for the transaction.
P%PARITY_H<2:0>	T	All	All	Parity for **P%NDAL_H**, **P%CMD_H**, and **P%ID_H**.
P%ACK_L	OD	All	All	NDAL acknowledgement of receipt.
Clock signals				
P%SYS_RESET_L	SS	NVAX	All but NVAX	Resets all nodes.
PHI12_H	SS	NVAX	All	PHI12 clock for all bus residents.
PHI23_H	SS	NVAX	All	PHI23 clock for all bus residents.
PHI34_H	SS	NVAX	All	PHI34 clock for all bus residents.
PHI41_H	SS	NVAX	All	PHI41 clock for all bus residents.

[1] Indicates whether the pin is Single Source (SS), Tristate (T), or Open Drain (OD)

3.3.1 Terms

In order to clearly describe the transactions which occur on the NDAL, the following terms are used:

- Node - A node is a hardware device that connects to the NDAL. The largest NDAL system configuration will support 4 nodes.
- Transfer - A transfer is the smallest quantum of work that occurs on the NDAL. Typical examples of transfers are the address cycle of a read, the address cycle of a write, and each data cycle of a write.
- Transaction - A transaction is composed of one or more transfers. Transaction is the name given to the logical task being performed (e.g., read); in the case of the read specifically, the transaction consists of a command transfer followed some time later by a return data transfer. See Commander, Responder, Transmitter, and Receiver below.
- Commander - The commander is the node that initiated the transaction in progress. In any write transaction, the commander is the node that requested the write; for reads, the commander is the one who requested the data. The distinction of being the commander in a transaction holds for the duration of the transaction in spite of the fact that in some cases it might appear that the commander changes. A case in point is where the commander initiates a read transaction. It is the responder (data source) that initiates the return data transfer, but the node that requested the data is still the commander.
- Responder - The responder is the complement to the commander in a transaction.
- Transmitter - The transmitter during an NDAL cycle is the node that is driving the information on the NDAL. Using the read transaction as an example, the commander is the transmitter during the command transfer; during the return data transfer the commander is the receiver.
- Receiver - The receiver receives the data being moved during a transfer.
- Naturally Aligned - Refers to a data quantity whose address could be specified as an offset, from the beginning of memory, of an integral number of data elements of the same size. The lower address bits of a piece of naturally aligned data are zero.
- ETM - Error Transition Mode. The backup cache enters Error Transition Mode when an error occurs. While in ETM, the state of the backup cache is preserved as much as possible. It continues to service requests to blocks which it owns, since those contain the only valid copy of data in the system. ETM is described completely in Chapter 13.
- Address cycle - The cycle during which the address of the transaction is transmitted on the NDAL. This is the first cycle of a read or write.
- Data cycle - A cycle during which the NDAL transfers data. These include data cycles of a write and fill data cycles.
- Read Data Return - This is the command used during a cycle in which a responder is returning read data to a commander. These cycles are also referred to as fills.

3.3.2 NDAL Clocking

The NDAL is a four-phase bus. NVAX drives four two-phase overlapping clocks to the other chips on the NDAL as well as back to itself, as shown in Table 3–6.

Table 3–6: NDAL clocks

NVAX output pin	NDAL clock	NVAX input pin
P%PHI12_OUT_H	PHI12_H	P%PHI12_IN_H
P%PHI23_OUT_H	PHI23_H	P%PHI23_IN_H
P%PHI34_OUT_H	PHI34_H	P%PHI34_IN_H
P%PHI41_OUT_H	PHI41_H	P%PHI41_IN_H

See Chapter 17 for more details.

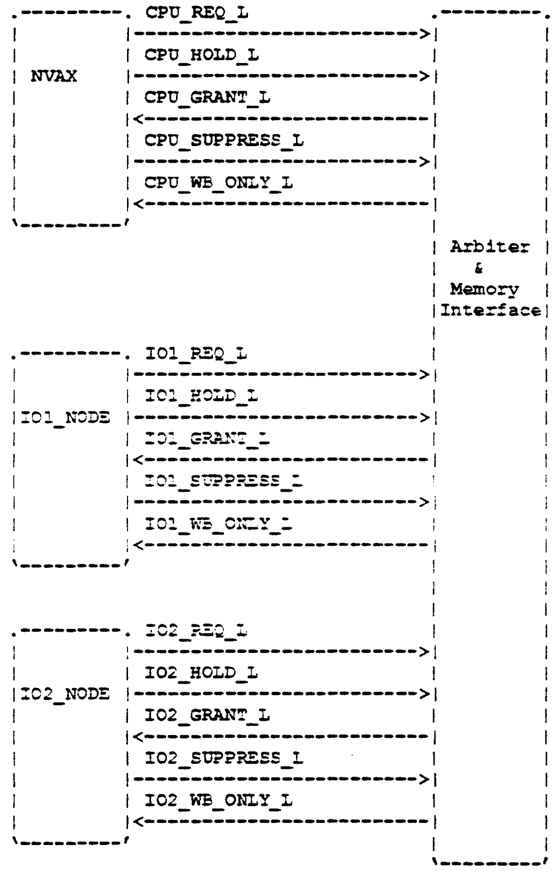
3.3.3 NDAL Arbitration

The NDAL protocol can architecturally support up to 4 nodes, which consist of one NVAX CPU and three interfaces to memory or I/O. This spec assumes one interface to memory and two interfaces to I/O. The I/O interfaces are referred to as IO1_NODE and IO2_NODE. The non-CPU nodes may or may not contain caches.

At a given time, any or all of the nodes may desire the use of the NDAL. Arbitration cycles occur in parallel with data transfer cycles using a set of lines dedicated specifically for arbitration.

Figure 3–3 shows the connection of the arbitration signals on the fully-configured NDAL. This arbitration scheme assumes that the arbiter is built into the memory interface. If the arbiter were built as a separate chip, the memory interface would need its own request, hold, grant, suppress, and wb_only lines. When the arbiter is built into the memory interface, the memory interface can withhold grant if its input queues are filling up.

Figure 3–3: NDAL Arbitration Block Diagram



The following sections describe the NDAL arbitration signals.

3.3.3.1 NDAL Arbitration Signals

3.3.3.1.1 P%CPU_REQ_L

NVAX asserts **P%CPU_REQ_L** to request the NDAL for the following cycle. **P%CPU_REQ_L** is a unidirectional signal from NVAX to the arbiter.

3.3.3.1.2 IO1_REQ_L

IO1_NODE, an interface node, asserts IO1_REQ_L when it wants to drive the NDAL. IO1_REQ_L is a unidirectional signal from IO1_NODE to the arbiter.

3.3.3.1.3 IO2_REQ_L

IO2_NODE, an interface node, asserts IO2_REQ_L when it wants to drive the NDAL. IO2_REQ_L is a unidirectional signal from IO2_NODE to the arbiter.

3.3.3.1.4 P%CPU_HOLD_L

The NVAX CPU asserts **P%CPU_HOLD_L** in order to gain access to the NDAL for consecutive cycles. The NVAX CPU only asserts **P%CPU_HOLD_L** when **P%CPU_GRANT_L** is asserted; it never asserts **P%CPU_HOLD_L** unless **P%CPU_GRANT_L** is asserted. Assertion of **P%CPU_HOLD_L** guarantees that NVAX may retain ownership of the NDAL in the next cycle, independent of the value of any other outstanding requests. The arbiter must grant the bus to the CPU if the CPU asserts **P%CPU_HOLD_L**.

P%CPU_HOLD_L is used for multicycle transfers, allowing NVAX to acquire consecutive cycles. NVAX asserts **P%CPU_HOLD_L** for hexaword Disown Write transactions, in order to transfer the four quadwords of data consecutively and directly after the address cycle; and for quadword Write or Disown Write transactions, in order to transfer the one quadword of data directly after the address cycle. NVAX never asserts **P%CPU_HOLD_L** for more than four contiguous cycles.

3.3.3.1.5 IO1_HOLD_L

IO1_HOLD_L is analogous to **P%CPU_HOLD_L**. It performs HOLD functionality for IO1_NODE.

IO1_NODE may not assert IO1_HOLD_L unless IO1_GRANT_L is asserted during the current NDAL cycle. Assertion of IO1_HOLD_L guarantees that IO1_NODE may retain ownership of the NDAL in the next cycle, independent of the value of any other outstanding requests. The arbiter must grant the bus to IO1_NODE if it asserts IO1_HOLD_L.

IO1_HOLD_L signal is used for multicycle transfers, allowing IO1_NODE to acquire consecutive cycles. In a hexaword write transaction, for instance, IO1_NODE asserts IO1_HOLD_L in order to transfer the four quadwords of data consecutively. IO1_HOLD_L may also be used to transfer Fill data in consecutive cycles. IO1_HOLD_L may be asserted for a maximum of four contiguous cycles.

3.3.3.1.6 IO2_HOLD_L

IO2_HOLD_L is analogous to IO1_HOLD_L. It performs HOLD functionality for IO2_NODE.

3.3.3.1.7 P%CPU_SUPPRESS_L

NVAX asserts **P%CPU_SUPPRESS_L** in order to suppress new NDAL transactions which NVAX treats as cache coherency requests. It does this when its two-entry cache coherency queue (the NDAL_IN_QUEUE) is in danger of overflowing.

During the cycle when **P%CPU_SUPPRESS_L** is asserted, NVAX will accept a new transaction. NVAX requires transactions in the following cycle to be suppressed.

While **P%CPU_SUPPRESS_L** is asserted, only fills and writebacks are allowed to proceed from non-CPU nodes. The CPU may continue to put all transactions onto the bus (as long as **P%CPU_WB_ONLY_L** is not asserted). Because the NDAL_IN_QUEUE is full and takes the highest priority within the Cbox, NVAX is mostly working on cache coherency transactions while **P%CPU_SUPPRESS_L** is asserted, which may cause NVAX to issue WDISOWNs on the NDAL. However, NVAX may and does issue any type of transaction while **P%CPU_SUPPRESS_L** is asserted.

3.3.3.1.8 IO1_SUPPRESS_L

IO1_NODE can suppress new transactions on the NDAL by asserting IO1_SUPPRESS_L. Fills and writebacks will proceed as usual.

3.3.3.1.9 IO2_SUPPRESS_L

IO2_NODE can suppress new transactions on the NDAL by asserting IO2_SUPPRESS_L. Fills and writebacks will proceed as usual.

3.3.3.1.10 P%CPU_GRANT_L

P%CPU_GRANT_L is asserted to notify NVAX that it must drive the NDAL during the following cycle. When **P%CPU_GRANT_L** is asserted, NVAX must drive the bus with a valid command and correct parity. If NVAX did not request the NDAL, it drives the bus with a NOP. It only drives a non-NOP command if it actually requested the NDAL in the previous cycle.

If NVAX asserts **P%CPU_HOLD_L**, **P%CPU_GRANT_L** must be asserted in the next cycle.

3.3.3.1.11 IO1_GRANT_L

The arbiter asserts IO1_GRANT_L when IO1_NODE is permitted to drive the bus. When IO1_GRANT_L is asserted, IO1_NODE must drive the bus with a valid command and correct parity. If IO1_HOLD_L is asserted, IO1_GRANT_L must be asserted in the next cycle.

3.3.3.1.12 IO2_GRANT_L

IO2_GRANT_L is analogous to IO1_GRANT_L. It grants the bus to IO2_NODE.

3.3.3.1.13 P%CPU_WB_ONLY_L

When **P%CPU_WB_ONLY_L** is asserted, NVAX will only issue Write Disown or NOP commands, including Write Disowns due to Write Unlocks when the cache is off or in ETM. Otherwise, NVAX will not issue any new requests. During the cycle in which **P%CPU_WB_ONLY_L** is asserted, the system must be prepared to accept one more non-writeback command from the CPU. Starting with the cycle following the assertion of **P%CPU_WB_ONLY_L**, NVAX will only issue writeback commands.

3.3.3.1.14 IO1_WB_ONLY_L

IO1_WB_ONLY_L is driven by the arbiter and received by IO1_NODE. When IO1_WB_ONLY_L is asserted, IO1_NODE only arbs for the bus in order to return fills or disown writes. It does not initiate any new transactions.

3.3.3.1.15 IO2_WB_ONLY_L

IO2_WB_ONLY_L is driven by the arbiter and received by IO2_NODE. When IO2_WB_ONLY_L is asserted, IO2_NODE only arbs for the bus in order to return fills or disown writes. It does not initiate any new transactions.

3.3.3.2 NDAL Arbitration Timing

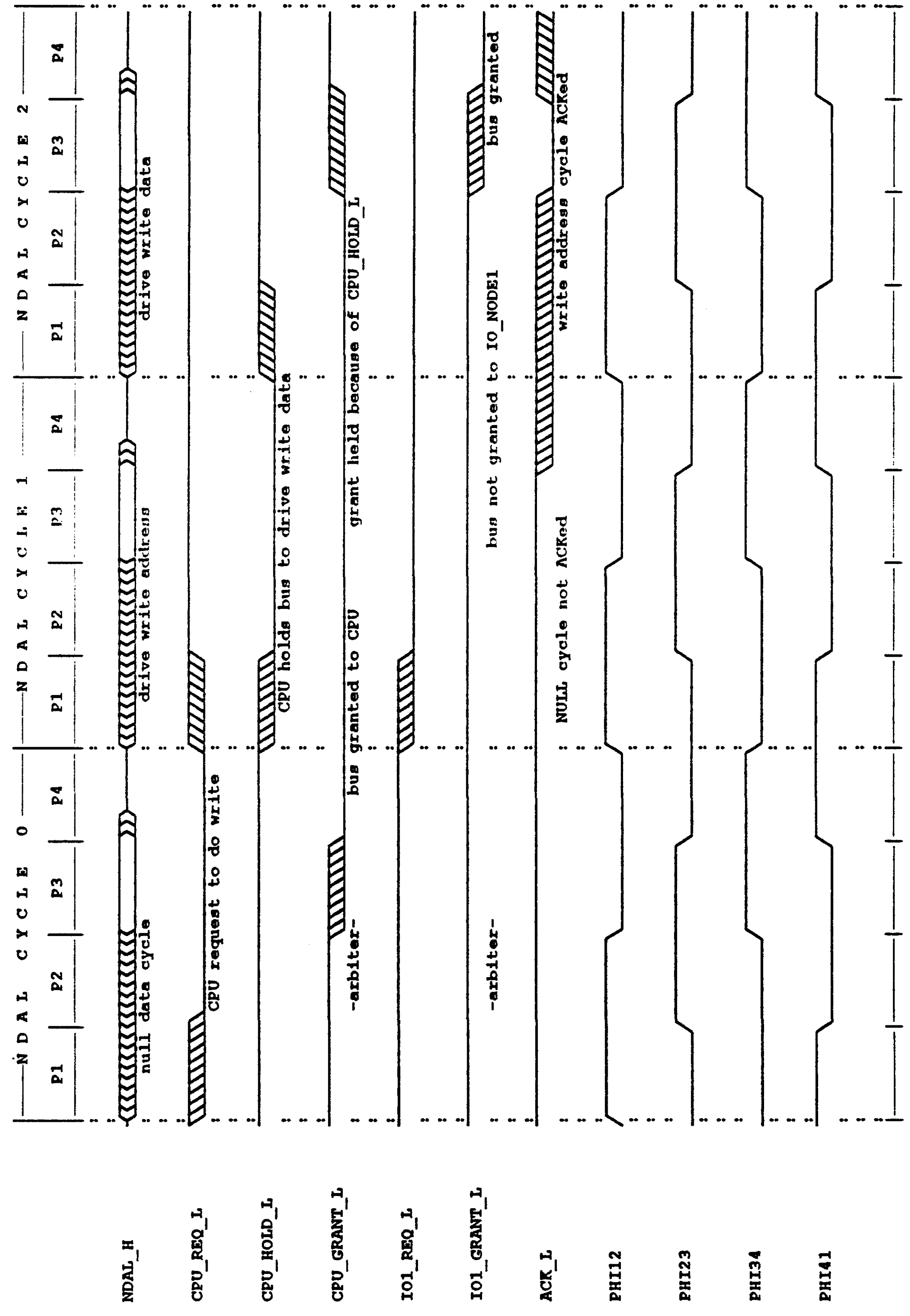
The timing for NDAL arbitration is shown in Figure 3–4. There are several critical spots to note in the diagram. The arbiter receives the request lines by the end of P1. It must drive the grant lines to valid values by the end of P3. It has two phases to calculate arbitration and to drive the grant lines across the board.

In the fastest system (10ns NVAX), the arbiter has 15ns after receiving the request lines to arbitrate and to drive the grant lines. Board simulations for one system show that driving the grant lines will take about half that time.

From the time a bus driver receives its grant line, it has three phases to drive **P%NDAL_H<63:0>**, **P%CMD_H<3:0>**, **P%ID_H<2:0>**, and **P%PARITY_H<2:0>** to valid levels.

From the time the NDAL is valid on its pins, the receiver has four phases to compute parity and to assert **P%ACK_L**.

Figure 3–4: NDAL Arbitration timing



3.3.3.3 NDAL Suppress and Its Timing

When any node asserts its suppress line, no transactions other than writebacks or fills must be driven onto the bus, starting in the following cycle. For example, when **P%CPU_SUPPRESS_L** is asserted, the arbiter can accomplish this in the following way: if **P%CPU_SUPPRESS_L** is asserted during cycle 0, the arbiter does not grant the bus to any node, with the possible exception of the CPU, in cycle 0. At the same time it asserts IO1_WB_ONLY and IO2_WB_ONLY. In cycle 1, the arbiter continues to perform bus arbitration as it normally would, but now IO1_NODE and IO2_NODE recognize the assertion of their respective WB_ONLY lines, and they do not request the bus except for fills and writebacks.

From this, it may be seen that the assertion of **P%CPU_SUPPRESS_L** causes the arbiter to assert IO1_WB_ONLY_L and IO2_WB_ONLY_L; the assertion of IO1_SUPPRESS_L causes the arbiter to assert **P%CPU_WB_ONLY_L** and IO2_WB_ONLY_L; and the assertion of IO2_SUPPRESS_L causes the arbiter to assert **P%CPU_WB_ONLY_L** and IO1_WB_ONLY_L.

The timing for suppression of the bus is shown in Figure 3–5. In this example, the CPU suppresses the bus by asserting **P%CPU_SUPPRESS_L**, which is valid at the end of P1 in NDAL cycle 0. The arbiter immediately asserts IO1_WB_ONLY_L and IO2_WB_ONLY_L, which are valid by the end of P3 in the same cycle. This notifies IO1_NODE and IO2_NODE that they should not arbitrate for the bus for new transactions, only for writebacks and fills. (If the IO chip cannot suppress its request line quickly enough, it may drive NOPs onto the NDAL if it gets GRANT, instead of withdrawing its request in the first cycle.) Accordingly, in NDAL CYCLE 1 as shown in the diagram, IO1_REQ_L is deasserted by IO1_NODE, since it has a read or a write request to do. IO2_REQ_L remains asserted because IO2_NODE has a fill to do.

During the cycle in which **P%CPU_SUPPRESS_L** is asserted, the arbiter does not grant to any node with the exception of the CPU. Since it is the one suppressing the bus it should be allowed to continue issuing transactions on the bus.

If a node had its HOLD line asserted and it had been granted the bus in the cycle before, it WOULD get grant under suppress. The rules for HOLD override the rules for SUPPRESS.

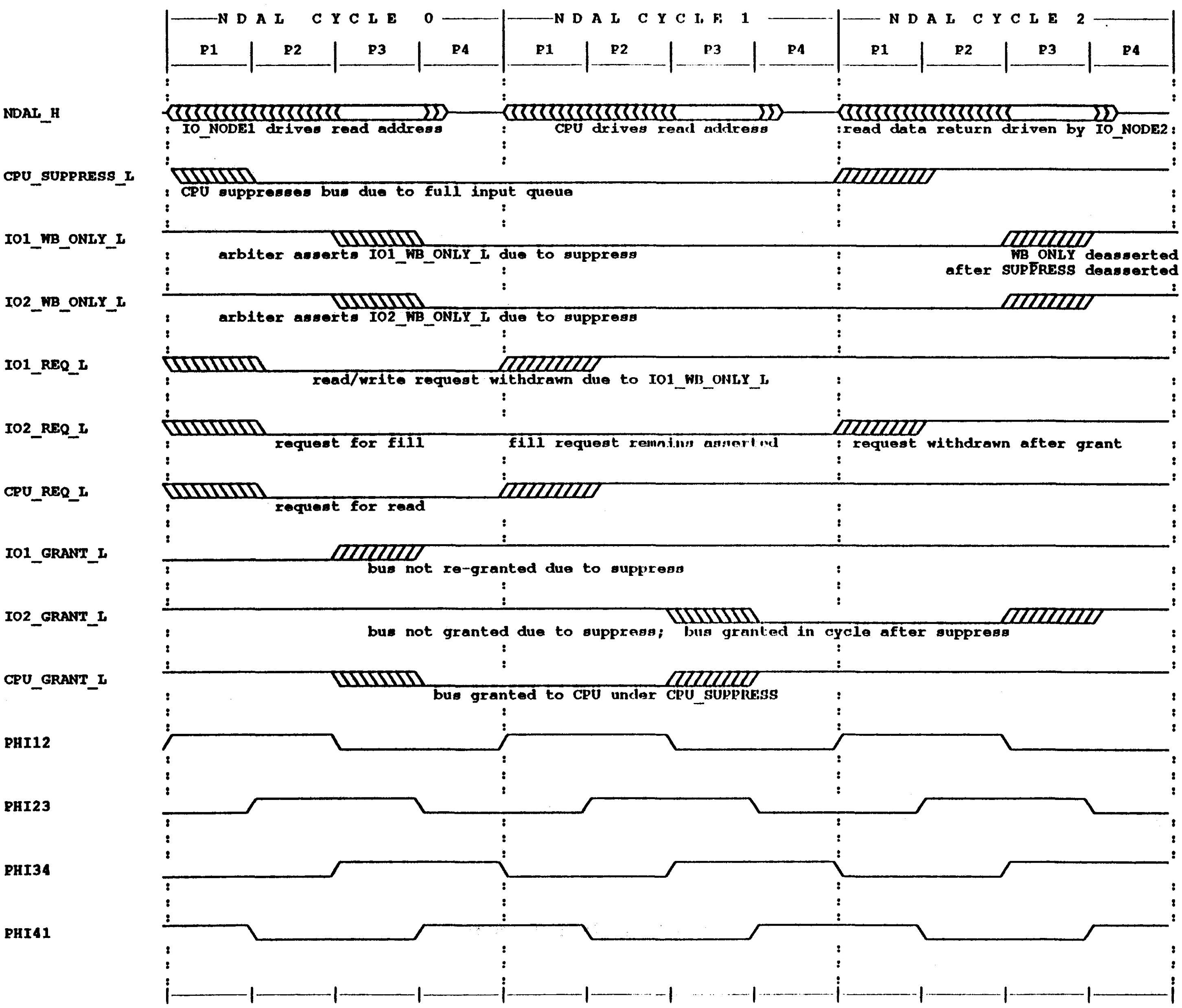
In NDAL CYCLE 1, the bus is granted to IO2_NODE which has arb'd to do its fill. The fill is driven in NDAL cycle 2.

3.3.3.4 NDAL Arbitration Rules

The rules of arbitration are as follows:

1. Any node may assert its request line during any cycle.
2. A node's grant line must be asserted before that node drives the NDAL.
3. An NDAL driver may only assert its HOLD_L line if it has been granted the bus for the current cycle.
4. If a node has been granted the bus, and it asserts HOLD, it is guaranteed to be granted the bus in the following cycle.
5. HOLD may only be used in two cases: (a) to hold the bus for the data cycles of a write; (b) to send consecutive fill cycles.

Figure 3–5: NDAL Suppress timing


NDAL CYCLE 0
NDAL CYCLE 1
NDAL CYCLE 2
P1
P2
P3
P4
NDAL_H
IO_NODE1 drives read address
CPU drives read address
read data return driven by IO_NODE2
CPU_SUPPRESS_L
CPU suppresses bus due to full input queue
IO1_WB_ONLY_L
arbiter asserts IO1_WB_ONLY_L due to suppress
WB_ONLY deasserted after SUPPRESS deasserted
IO2_WB_ONLY_L
arbiter asserts IO2_WB_ONLY_L due to suppress
IO1_REQ_L
read/write request withdrawn due to IO1_WB_ONLY_L
IO2_REQ_L
request for fill
fill request remains asserted
request withdrawn after grant
CPU_REQ_L
request for read
IO1_GRANT_L
bus not re-granted due to suppress
IO2_GRANT_L
bus not granted due to suppress; bus granted in cycle after suppress
CPU_GRANT_L
bus granted to CPU under CPU_SUPPRESS
PHI12
PHI23
PHI34
PHI41

6. HOLD must be used to retain the bus for the data cycles of a write, as the data cycles must be contiguous with the write address cycle.
7. HOLD must not be used to retain the bus for new transactions, as arbitration fairness would not be maintained.
8. If a node requests the bus and is granted the bus, it must drive the NDAL during the granted cycle with a valid command. NOP is a valid command. NVAX takes this a step further and drives NOP if it is granted the bus when it did not request it.
9. Any node which issues a read must be able to accept the corresponding fills as they cannot be suppressed or slowed.
10. If a node's WB_ONLY line is asserted, it may only drive the NDAL with NOP, RDE, RDRn, WDISOWN, WDATA, or BADWDATA.
11. If a node asserts its SUPPRESS line, the arbiter must not grant the bus to any node except that one in the next cycle. At the same time the arbiter must assert the appropriate WB_ONLY lines. In the following cycle, the arbiter must grant the bus normally.
12. The rules for HOLD override the rules for SUPPRESS.
13. The bus must be actively driven during every cycle.

Specifics on arbitration algorithms may be found in the system specs for each NVAX system.

3.3.4 NDAL Information Transfer

3.3.4.1 P%NDAL_H<63:0>

The use of this field is multiplexed between address and data information. On data cycles the lines represent 64 bits of read or write data; on address cycles the lines represent address, byte enable, and length information.

There are four types of data cycles: Write Data, Bad Write Data, Read Data Return, and Read Data Error. During write data cycles the commander drives its Commander ID on **P%ID_H<2:0>** and drives data on **P%NDAL_H<63:0>**. The full 64 bits of data are written during hexaword writes. For octaword and quadword length writes, the data bytes which are written correspond to the byte enable bits which were asserted during the address cycle which initiated the transaction. During Read Data Return and Read Data Error cycles the responder drives the original commander ID.

The NDAL address cycle is used by a commander to initiate an NDAL transaction. On address cycles the address is driven in the lower longword of the bus, and the byte enable and transaction length are in the upper longword, as shown in Figure 3–6.

Figure 3–6: Address Cycle Format

```
 63 62 61 60|59 58 57 56|55 54 53 52|51 50 49 48|47 46 45 44|43 42 41 40|39 38 37 36|35 34 33 32
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| Len |                 |  high qw byte enable  |  low qw byte enable   |        unused         |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  space |                                       address                                        |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
     |
     '-- Mem = 000..110
         I/O = 111
```

Each field shown in the diagram is described in the sections which follow.

3.3.4.1.1 Address Field

The address space supported by the NDAL is divided into memory space and I/O space.

The lower 32 bits of the address cycle **P%NDAL_H<31:0>** define the address of an NDAL read or write transaction. The NDAL supports a 4 Gigabyte (2**32 byte) address space. The most significant bits of this address (corresponding to lines **P%NDAL_H<31:29>**) select 512 Mb I/O space (**P%NDAL_H<31:29>** = 111) or 3.5 Gb memory space (**P%NDAL_H<31:29>** = 000..110).

Figure 3–7 illustrates the division of the address space into memory space and I/O space.

The division of the NDAL address space in the I/O region is further defined to accommodate the need for NDAL node and I/O node address space. More information about the division of I/O space may be found in Chapter 2.

Figure 3–7: Physical Address Space Layout

```
               +--------------------------+
00000000       |                          |
               |                          |
               +-                        -+
               |                          |
               |                          |
               +-                        -+
               |          Memory          | 3.5 Gigabytes
               |           Space          |
               +-                        -+
               |                          |
               |                          |
               +-                        -+
               |                          |
               |                          |
               +-                        -+
               |                          |
               |                          |
               +-                        -+
               |                          |
DFFFFFFF       |                          |
               ---------------------------+
E0000000       |           I/O            |
FFFFFFFF       |          space           | 512 Megabytes
               ----------------------------
```

Address bits <31:0> are all significant bits in an address to I/O space. Although the length field on the NDAL is always quadword for I/O space reads and writes, the actual amount of data read or written may be less than a quadword. The byte enable is used to read or write the requested bytes only. If the byte enable indicates a 1-byte read or write, every bit of the address is significant. The lower bits of the address are provided so that the I/O adapters do not have to deduce the address from the byte enable.

The number of significant bits in an address to MEMORY depends on the transaction type and length as shown in Figure 3–8.

Figure 3–8: NDAL Memory Address Interpretation

```
                                     A<i>, i= 4 3 2 1 0
                                              +-+-+-+-+-+
Read quadword, octaword, hexaword             |s|s|d|d|d|
                                              +-+-+-+-+-+
Write quadword                                |s|s|d|d|d|
                                              +-+-+-+-+-+
Write octaword                                |s|d|d|d|d|
                                              +-+-+-+-+-+
Write hexaword                                |d|d|d|d|d|
                                              +-+-+-+-+-+

                                            s = significant
                                            d = don't care
```

It can be seen from the figure that bits A<4:3> are significant address bits or don't care, depending on the function being requested.

All reads have significant bits down to the quadword. Although fills may be returned in any order, there is a performance advantage if memory returns the requested quadword first. The NDAL protocol identifies each quadword using one of the four Read Data Return commands, so that quadwords can be placed in correct locations regardless of the order in which they are returned.

Quadword, octaword and hexaword writes are always naturally aligned and driven on the NDAL in order from the lowest-addressed quadword to the highest.

3.3.4.1.2 Byte Enable Field

The Byte Enable field is located in **P%NDAL_H<55:40>** during the address cycle. It is used to supply byte-level enable information for quadword-length OREADs, IREADs, DREADs, WRITEs, and WDISOWNs and octaword-length WRITEs and WDISOWNs. Of these transactions, NVAX generates only quadword IREADs and DREADs to I/O space, quadword WRITEs to I/O space, and quadword WRITEs and WDISOWNs to memory space.

If the byte enable is a "1", the byte is to be read or written. If it is a "0", the byte is not read or written.

NOTE

> During quadword-length transactions the high portion of the byte enable field, located in **P%NDAL_H<55:48>**, is ignored. Commanders may drive any data pattern they wish in this field as long as it has correct parity. Responders must not depend on a certain defined pattern (such as all zeros).
>
> During hexaword-length transactions the entire byte enable field is ignored. During hexaword transactions, commanders are permitted to drive any data pattern they wish in this field as long as it has correct parity. Responders must not depend on a certain defined pattern (such as all zeros).
>
> During octaword-length transactions, the byte enable located in **P%NDAL_H<47:40>** always corresponds to the low-order quadword of the octaword. The byte enable located in **P%NDAL_H<55:48>** always corresponds to the high-order quadword of the octaword.

The correspondence between bits in the enable and bytes of the data is shown in Table 3–7 and Table 3–8.

Table 3–7: Byte Enable for Quadword Reads and Writes

Address cycle	Data cycle
Byte Enable	Data Byte
P%NDAL_H<47>	P%NDAL_H<63:56>
P%NDAL_H<46>	P%NDAL_H<55:48>
P%NDAL_H<45>	P%NDAL_H<47:40>
P%NDAL_H<44>	P%NDAL_H<39:32>
P%NDAL_H<43>	P%NDAL_H<31:24>
P%NDAL_H<42>	P%NDAL_H<23:16>
P%NDAL_H<41>	P%NDAL_H<15:08>
P%NDAL_H<40>	P%NDAL_H<07:00>

Table 3–8: Byte Enable for Octaword Writes

Address cycle	First data cycle	Second data cycle
Byte Enable Bit	Quadword 0 Data Byte	Quadword 1 Data Byte
P%NDAL_H<47>	P%NDAL_H<63:56>	
P%NDAL_H<46>	P%NDAL_H<55:48>	
P%NDAL_H<45>	P%NDAL_H<47:40>	
P%NDAL_H<44>	P%NDAL_H<39:32>	
P%NDAL_H<43>	P%NDAL_H<31:24>	
P%NDAL_H<42>	P%NDAL_H<23:16>	
P%NDAL_H<41>	P%NDAL_H<15:08>	
P%NDAL_H<40>	P%NDAL_H<07:00>	
P%NDAL_H<55>		P%NDAL_H<63:56>
P%NDAL_H<54>		P%NDAL_H<55:48>
P%NDAL_H<53>		P%NDAL_H<47:40>
P%NDAL_H<52>		P%NDAL_H<39:32>
P%NDAL_H<51>		P%NDAL_H<31:24>
P%NDAL_H<50>		P%NDAL_H<23:16>
P%NDAL_H<49>		P%NDAL_H<15:08>
P%NDAL_H<48>		P%NDAL_H<07:00>

Table 3–9 illustrates possible bit patterns in the byte enable for transactions which NVAX generates. Only transactions in which the byte enable is valid are listed.

NVAX will generate every possible byte enable for every possible address for quadword WRITEs and WDISOWNs to memory space, as shown by the table. IREADs to I/O space will always request a full quadword of data by asserting all the byte enable bits.

DREADs and WRITEs to I/O space are issued using the quadword length NDAL encoding, but the requests are for only a byte, word, or longword at a time, as indicated by the byte enable given in the command cycle of a transaction. References that are unaligned across a naturally aligned quadword are decomposed into two separate requests for the bytes in each quadword; where this is the case, Table 3–9 shows the byte enable values for both references generated. In the cases where a second request is generated, the address is incremented by 8, which addresses the next quadword in I/O space, but address bits <2:0> are 000(BIN).

When the NVAX CPU does an I/O space read for an interrupt acknowledge (IAK read), it always generates a longword-aligned word-length read request. In other words, the byte enable which NVAX uses for an IAK read is either 0000 0011 (binary) or 0011 0000 (binary).

Table 3–9 reflects what NDAL requests the NVAX CPU will generate, depending on the software written. Software must take care only to generate requests which make sense in the system environment. Specifically, unaligned requests are forbidden by DEC Standard 032.

Table 3–9: Possible Byte Enables for NVAX-generated transactions

				Byte Enable<7:0>			
NDAL Transaction	**Software Addr<2:0>**	**NDAL# Req.#**	**NDAL Addr<2:0>**	**Software Byte Req.**	**Software Word Req.**	**Software LW Req.**	**Software QW Req.**
QW WRITE, WDISOWN (memory space)	any addr	1st	same as SW	unrestricted	unrestricted	unrestricted	unrestricted
QW IREAD (I/O space)	000	1st		— —	— —	— —	1111 1111
QW DREAD, QW WRITE (I/O space)	000	1st	000	0000 0001	0000 0011	0000 1111	0000 1111
		2nd	100	— —	— —	— —	1111 0000
	001	1st	001	0000 0010	0000 0110	0001 1110	0001 1110
		2nd	101	— —	— —	— —	1110 0000
		3rd	000	— —	— —	— —	0000 0001
	010	1st	010	0000 0100	0000 1100	0011 1100	0011 1100
		2nd	110	— —	— —	— —	1100 0000
		3rd	000	— —	— —	— —	0000 0011
	011	1st	011	0000 1000	0001 1000	0111 1000	0111 1000
		2nd	111	— —	— —	— —	1000 0000
		3rd	000	— —	— —	— —	0000 0111
	100	1st	100	0001 0000	0011 0000	1111 0000	1111 0000
		2nd	000	— —	— —	— —	0000 1111
	101	1st	101	0010 0000	0110 0000	1110 0000	1110 0000
		2nd	000	— —	— —	0000 0001	0000 0001
		3rd	001	— —	— —	— —	0001 1110
	110	1st	110	0100 0000	1100 0000	1100 0000	1100 0000
		2nd	000	— —	— —	0000 0011	0000 0011
		3rd	010	— —	— —	— —	0011 1100
	111	1st	111	1000 0000	1000 0000	1000 0000	1000 0000
		2nd	000	— —	0000 0001	0000 0111	0000 0111
		3rd	011	— —	— —	— —	0111 1000

3.3.4.1.2.1 I/O space writes

When the NVAX CPU issues an I/O space write, it always replicates the data identically on the high longword and the low longword of the NDAL, although the byte enable indicates that the data is only valid in one longword or the other. A system device may take advantage of this fact to avoid rotating the data.

3.3.4.1.3 Length Field

The length field is used to indicate the amount of data to be read or written for the current transaction. Table 3–10 shows how the length values correspond to transaction lengths.

Table 3–10: NDAL Length Field

P%NDAL_H<63:62>	length
00	hexaword
01	unused
10	quadword
11	octaword (not used by NVAX CPU)

3.3.4.2 P%CMD_H<3:0>

The **P%CMD_H<3:0>** lines specify the current bus transaction during any given cycle. The interpretation of the four bits is shown in Table 3–11.

Table 3–11: NDAL Command Encodings and Definitions

Levels	Abbrev.	Bus Transaction	Type	Function
0000	NOP	No Operation	Nop	No Operation
0001	Reserved			
0010	WRITE	Write	Addr	Write to memory with byte enable if quadword or octaword
0011	WDISOWN	Write Disown	Addr	Write memory; cache disowns block and returns ownership to memory
0100	IREAD	Instruction Stream Read	Addr	Instruction-stream read
0101	DREAD	Data Stream Read	Addr	Data-stream read (without ownership)
0110	OREAD	D-Stream Read Ownership	Addr	Data-stream read claiming ownership for the cache
0111	Reserved			
1000	Reserved			
1001	RDE	Read Data Error	Data	Used instead of Read Data Return in the case of an error.
1010	WDATA	Write Data Cycle	Data	Write data is being transferred
1011	BADWDATA	Bad Write Data	Data	Write data with errors is being transferred
1100	RDR0	Read Data0 Return (fill)	Data	Read data is returning corresponding to QW 0 of a hexaword.
1101	RDR1	Read Data1 Return (fill)	Data	Read data is returning corresponding to QW 1 of a hexaword.
1110	RDR2	Read Data2 Return (fill)	Data	Read data is returning corresponding to QW 2 of a hexaword.
1111	RDR3	Read Data3 Return (fill)	Data	Read data is returning corresponding to QW 3 of a hexaword.

The NVAX CPU does not implement all transaction lengths with all commands. The commands and lengths which it uses are in the table which follows. If NVAX implements the command in memory space, MEM is indicated in the table; if it implements the command in I/O space, I/O is indicated in the table.

Table 3–12: NDAL Address Cycle Commands as used by the NVAX CPU

COMMAND	QUADWORD	OCTAWORD	HEXAWORD
IREAD	I/O	—	MEM
DREAD	I/O	—	MEM
OREAD	—	—	MEM
WRITE	MEM[1],I/O	—	—
WDISOWN	MEM[1]	—	MEM

[1] NVAX uses these transactions only when the backup cache is disabled or in Error Transition Mode.

When the cache is off, the NVAX CPU issues OREAD commands of hexaword length, and corresponding Disown Write commands of quadword length. These correspond to the CPU-internal commands of Read Lock and Write Unlock. The lock/ownership granularity in memory must not be less than a hexaword. Otherwise, when the CPU did a hexaword OREAD followed by a quadword Disown Write, the other three quadwords would be in limbo. The CPU would assume that it didn't own them, and memory would believe that they were still owned by the cache.

3.3.4.3 P%ID_H<2:0>

During the address cycle and return data cycles, **P%ID_H<2:0>** contain the commander's ID. This ID is used to identify the source of the request on the address cycle and to associate returning data with the commander who issued the request on return data cycles.

The commander ID codes available for use by a node are shown in Table 3–13. **P%ID_H<2:1>** indicate which node originated the transaction, and **P%ID_H<0>** indicates which of two outstanding reads per node.

Table 3–13: Commander P%ID_H Assignments

Node Name	P%ID_H<2:0>
NVAX	00X
memory interface	01X
IO1_NODE	10X
IO2_NODE	11X

During write command and data cycles, **P%ID_H<2:0>** is driven with the ID of the commander. **P%ID_H<2:1>** is driven with the bits identifying the commander, and **P%ID_H<0>** may be driven with any value. **P%ID_H<0>** is not necessarily driven with the same value during the command cycle of a write and the corresponding data cycles of that write.

Each commander node on the NDAL may have two read transactions outstanding.

The memory interface is not a commander node, but it has been assigned a commander ID which may be used in some NVAX systems. For example, in the XMI2 system, the memory interface reflects XMI2 read and write commands into the NDAL for cache coherency reasons. These reads and writes are not taken up by any node on the NDAL except to enforce cache coherency. The memory interface uses its own ID when driving these reads and writes onto the NDAL. If a write is reflected onto the NDAL merely to enforce cache coherency, the WDATA cycles may be omitted.

3.3.4.4 P%PARITY_H<2:0>

P%PARITY_H<2> is computed over **P%CMD_H<3:0>** and **P%ID_H<2:0>**. Even parity is used, where the "exclusive OR" of all bits including the parity bit is a "0". (All bits, including the parity bit, have an even number of "1"'s.)

P%PARITY_H<2> is inverted, forcing an NDAL parity error, when CCTL<FORCE_NDAL_PERR> is set. This is described in Chapter 13.

P%PARITY_H<1> is computed over the high longword of the NDAL, **P%NDAL_H<63:32>**. Odd parity is used, where the "exclusive OR" of all bits including the parity bit is a "1". (All bits, including the parity bit, have an odd number of "1"'s.)

P%PARITY_H<0> is computed over the low longword of the NDAL, **P%NDAL_H<31:0>**. Even parity is used.

Using a combination of odd and even parity means that neither all "1"'s nor all "0"'s is a legal bus pattern.

If a device requests the bus and is granted it, but chooses not to use it during a given cycle, it is responsible for driving the NOP command on **P%CMD_H<3:0>**. It must drive **P%NDAL_H<63:0>**, **P%ID_H<2:0>**, and **P%PARITY_H<2:0>** with correct parity.

If NVAX did not request the bus, and it is granted the NDAL anyway, it will drive the NDAL with a NOP.

When the bus is idle, the arbiter ensures that the NDAL is driven with correct parity. To do this, the arbiter may take advantage of the fact that NVAX will drive the NDAL with NOP if it is unexpectedly granted the bus.

The NVAX BIU checks the NDAL for correct parity in every cycle, regardless of the contents of the bus. It does not distinguish between errors on the command lines or the data lines; it computes the three parity bits, and if any fail, it responds to the error according to Table 3–21.

Table 3–14: NDAL Parity Coverage

Parity bit	protected data	parity type
P%PARITY_H<2>	P%CMD_H<3:0>,P%ID_H<2:0>	even parity
P%PARITY_H<1>	P%NDAL_H<63:32>	odd parity
P%PARITY_H<0>	P%NDAL_H<31:0>	even parity

3.3.4.5 P%ACK_L

P%ACK_L is an open drain signal which is pulled high (deasserted) by an external resistor on the board. The resistor is able to pull the node high during the time allotted without assistance from any other **P%ACK_L** driver. Thus, an **P%ACK_L** driver only has to pull the signal low at the appropriate time.

The receiver for a particular NDAL cycle is responsible for pulling **P%ACK_L** low (asserted) if it receives the cycle without parity errors. If another receiver detects a parity error on the cycle, it reports it by asserting **P%H_ERR_L** or **P%S_ERR_L**.

If **P%ACK_L** is asserted in response to an NDAL cycle, it indicates that the receiving node has accepted an address cycle or a data cycle. **P%ACK_L** being asserted for a read address cycle indicates that the responder will return a read response cycle at a later time. If it is asserted for a write address cycle, the transfer of the write address is assumed successful. If a cycle is accompanied by a NOP command, the cycle may or may not be acknowledged by the assertion of **P%ACK_L**; NOP's do not have to be acknowledged but they may be.

P%ACK_L is always asserted by the NDAL receiver unless there was a parity error on the bus. It is NOT used for flow control.

P%ACK_L is also not asserted when there is no node on the NDAL which recognizes the address space addressed, i.e., transactions to non-existent memory and I/O space will not receive **P%ACK_L** assertion.

See Table 3–21 for NVAX response when **P%ACK_L** is not asserted.

The timing of ACK_L relative to the data or address cycle is shown in Figure 3–9. For a given transfer, ACK_L is asserted one cycle later. In cycle 0 a read is driven, so ACK_L is asserted in cycle 1. In cycle 4 a NOP is driven, and in cycle 5 ACK_L is not asserted because NOP's do not have to be acknowledged.

Figure 3–9: P%ACK_L Timing

```
NDAL cycle|  0  |  1  |  2  |  3  |  4  |  5  |  6  |  7  |
          |-----|-----|-----|-----|-----|-----|-----|-----|
          |     |     |     |     |     |     |     |     |
Cmd       | Read| Writ| Wdat| Read| Nop | Writ| Wdat|     |
          |     |     |     |     |     |     |     |     |
          |     | ACK | ACK | ACK | ACK |     | ACK | ACK |
          |     |cycle|cycle|cycle|cycle|     |cycle|cycle|
ACK_L     |     |  0  |  1  |  2  |  3  |     |  5  |  6  |
          |     |     |     |     |     |     |     |     |
```

3.3.5 NDAL Transactions

The following sections describe the entire set of NDAL transactions.

Table 3–15 shows the entire set of NDAL commands and how they are used by NVAX.

In memory space, NVAX issues all reads with hexaword length. Normal writes to memory space are always quadword length, and Disown Writes are quadword or hexaword. When the cache is operating normally, Disown Writes are only issued in hexaword length. When the cache is in ETM, NVAX issues Disown Writes of both hexaword and quadword length. When the cache is off, NVAX issues only quadword Disown Writes. NVAX issues quadword Disown Writes only as the result of an interlock operation.

In I/O space, the ownership commands (OREAD and Disown Write) are not defined at all. NVAX issues only quadword operations in I/O space. NVAX never uses the BADWDATA command in I/O space.

Table 3–15: NDAL Command Usage by NVAX

Address Space	Command	Used by NVAX	Length QW	Length OW	Length HW
N/A	Nop	yes	-	-	-
N/A	Reserved	no	-	-	-
Memory	WRITE	yes	yes	no	no
Memory	WDISOWN	yes	yes	no	yes
Memory	IREAD	yes	no	no	yes
Memory	DREAD	yes	no	no	yes
Memory	OREAD	yes	no	no	yes
Memory	RDE	no	-	-	-
Memory	WDATA	yes	-	-	-
Memory	BADWDATA	yes	-	-	-
Memory	RDR0	no	-	-	-
Memory	RDR1	no	-	-	-
Memory	RDR2	no	-	-	-
Memory	RDR3	no	-	-	-
I/O	WRITE	yes	yes	no	no
I/O	WDISOWN	no	no	no	no
I/O	IREAD	yes	yes	no	no
I/O	DREAD	yes	yes	no	no
I/O	OREAD	no	no	no	no
I/O	RDE	no	-	-	-
I/O	WDATA	yes	-	-	-
I/O	BADWDATA	no	-	-	-
I/O	RDR0	no	-	-	-
I/O	RDR1	no	-	-	-
I/O	RDR2	no	-	-	-
I/O	RDR3	no	-	-	-

Table 3–16 shows the usage of NDAL commands by NDAL devices other than NVAX. The ownership commands (OREAD and WDISOWN) are not defined at all in I/O space. Although nodes may use OREAD and WDISOWN of lengths other than hexaword, they must be aware of the memory coherency problems connected with using lengths other than hexaword for these operations. Memory defines ownership along hexaword boundaries.

Table 3–16: NDAL Command Usage by NDAL nodes besides NVAX

Address Space	Command	Used by NDAL nodes	Length	Length	Length
			QW	OW	HW
N/A	Nop	yes	-	-	-
N/A	Reserved	no	-	-	-
Memory	WRITE	yes	yes	yes	yes
Memory	WDISOWN	yes	yes	yes	yes
Memory	IREAD	yes	yes	yes	yes
Memory	DREAD	yes	yes	yes	yes
Memory	OREAD	yes	yes	yes	yes
Memory	RDE	yes	-	-	-
Memory	WDATA	yes	-	-	-
Memory	BADWDATA	yes	-	-	-
Memory	RDR0	yes	-	-	-
Memory	RDR1	yes	-	-	-
Memory	RDR2	yes	-	-	-
Memory	RDR3	yes	-	-	-
I/O	WRITE	yes	yes	yes	yes
I/O	WDISOWN	no	no	no	no
I/O	IREAD	yes	yes	yes	yes
I/O	DREAD	yes	yes	yes	yes
I/O	OREAD	no	no	no	no
I/O	RDE	yes	-	-	-
I/O	WDATA	yes	-	-	-
I/O	BADWDATA	yes	-	-	-
I/O	RDR0	yes	-	-	-
I/O	RDR1	yes	-	-	-
I/O	RDR2	yes	-	-	-
I/O	RDR3	yes	-	-	-

3.3.5.1 Reads and Fills

The read address cycle, which is recognized by one of the three read commands (DREAD, IREAD, or OREAD) is decoded by the interfaces in the system, and the one which recognizes the address latches that address and command. This device is the responder. The responder uses Read Data Return or Read Data Error cycles to return the data. Reads and fills are described in the sections which follow.

3.3.5.1.1 Dstream Read Requests (DREAD)

An NDAL commander uses the DREAD command to request Data Stream data from a responder, either memory or an I/O device.

3.3.5.1.2 Istream Read Requests (IREAD)

The IREAD command is used to request Instruction-Stream data from a responder, either memory or an I/O device.

The separate I-stream read command is used in implementing halt protection for the CPU. When a system device which asserts P%HALT_L recognizes an I-stream read in halt-protected space, it prevents P%HALT_L from being asserted to the CPU. In the meantime, DREADs outside of halt-protected space may occur. When an IREAD outside of halt-protected space happens, the system device resumes asserting P%HALT_L to the CPU.

When NVAX issues the IREAD command in I/O space, it expects a full quadword of data in return. The responding device may decode the IREAD command instead of the byte enable field to detect the need to return a full quadword of data.

In addition, the separate IREAD command may be helpful in analysis during system debug or for performance analysis.

3.3.5.1.3 Ownership Read Requests (OREAD)

A node uses the OREAD command to gain ownership of a hexaword block of memory. Whereas previous systems implemented an Interlock read as well, the NDAL defines only the Ownership read. Interlocks can be accomplished using OREADs.

OREADs are only defined for memory space; they are not used in I/O space.

When memory receives an ownership read, an "owned" bit is set in memory and the read data is returned. Each hexaword in memory has an owned bit. The NVAX backup cache is organized by hexawords also, with an owned bit for each hexaword. Memory clears the owned bit when a Disown Write of any length is received to the same block.

3.3.5.1.4 How memory handles reads to Owned blocks

If the ownership bit is already set in memory when the OREAD arrives, data is not returned immediately to the commander. Once the node which owns the data Disown Writes the block, the Ownership bit is set in memory and the data is returned to the commander. The fact that the ownership bit was set at the beginning of the reference is transparent to the commander on the NDAL. Once an OREAD is issued on the NDAL, the data must be returned to the commander without requiring any retry sequence.

The analogous statement is true for an IREAD or a DREAD: If the ownership bit is already set in memory when the IREAD or DREAD arrives, data is not returned immediately to the commander. Once the node which owns the data Disown Writes the block, the the data is returned to the commander. The fact that the ownership bit was set at the beginning of the reference is transparent to the commander on the NDAL. Once an IREAD or DREAD is issued on the NDAL, the data must be returned to the commander without requiring any retry sequence.

In certain error-handling situations, NVAX itself may issue a read to a block which it already owns. In this case the memory controller should handle the read as it normally would: wait until NVAX completes the WDISOWN, then return the read data to NVAX and set the ownership bit if the read was an OREAD.

3.3.5.1.5 Read cycle description and timing

A read command cycle consists of a commander driving an address cycle on **P%NDAL_H<63:0>**, as shown in Section 3.3.4. The commander drives **P%CMD_H<3:0>** with DREAD, IREAD, or OREAD. It drives its own identification code on **P%ID_H<2:0>**, and it drives correct parity on **P%PARITY_H<2:0>**.

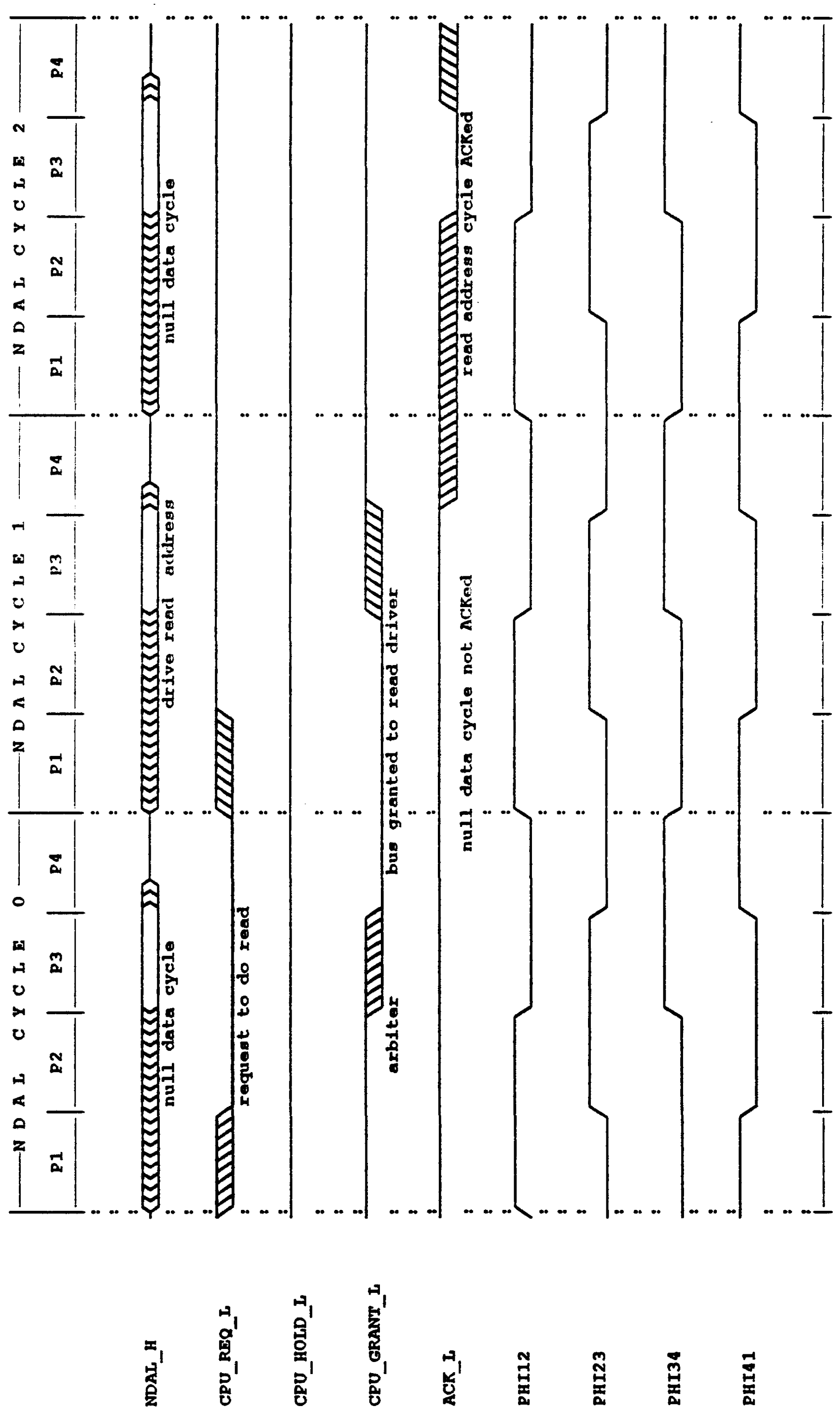
The timing for a read cycle is shown in Figure 3–10. In this example, NVAX is doing a read. In Cycle 0, NVAX asserts **P%CPU_REQ_L** to request the NDAL. It is granted the bus immediately, as shown by the assertion of **P%CPU_GRANT_L** in cycle 0. (This example assumes that no other device was requesting the NDAL during this cycle.)

The assertion of **P%CPU_GRANT_L** in phase 3 of cycle 0 means that NVAX is obligated to drive the NDAL in phase 1 of Cycle 1. It drives the read address out at that time. In this example, it deasserts its request line at the same time as it has no other requests to make. (It is not obligated to deassert request if it does have other requests to make.)

The device receiving the read recognizes it in phase 3 of cycle 1, and computes parity across the data it received. In this example, it recognizes no parity error, and asserts **P%ACK_L** so that it is valid in phase 3 of cycle 2. The CPU receives **P%ACK_L** and knows that the read address cycle completed successfully.

If there had been a parity error and **P%ACK_L** had not been asserted, NVAX would have responded with an error condition as described in Section 3.3.10.3.

Figure 3–10: NDAL Read timing



3.3.5.1.6 Read Data Return cycles (RDR0, RDR1, RDR2, RDR3)

The Read Data Return command is used in response to any read request, whether IREAD, DREAD, or OREAD. Multiple cycles are necessary to transfer all of the quadwords in a given hexaword transaction, and the cycles are not required to be consecutive. The commander, which has been monitoring the bus traffic waiting for its return data, latches the information. The responder returns the commander ID with the returned read data so the commander can recognize the returned read data it requested.

For a hexaword read, the four fill quadwords may be returned in any order. The NDAL Read Data Return command identifies the location of each quadword within the natural boundary as it is returned so that it can be placed in the correct location regardless of the return order. The data which is returned is naturally aligned within each quadword.

In I/O space, only one cycle's worth of data is returned. The actual amount of valid data returned depends upon the byte enable which was issued with the read request, as described in Section 3.3.4.1.2. The Read Data Return command corresponding to the requested I/O space address is used in returning the data.

Read Data Return cycles do not have to occur in adjacent cycles. The requested quadword should be returned as soon as possible, for performance reasons, even if the remaining quadwords are not yet available. The remaining quadwords may be sent as they become available.

Because the NDAL is a pended bus, multiple reads may be outstanding at a time. Because Read Data Return cycles do not have to occur contiguously, it is possible for Read Data Return cycles resulting from different read requests to take place in an interleaved fashion.

Table 3–17 shows the correspondence between address bits <4:3> and the RDR command used in returning data at that address. (Bits <4:3> indicate the alignment of a quadword of data within a hexaword.) The RDR command must correspond to the address of the data being returned for transactions of all lengths, whether quadword, octaword or hexaword. The correct RDR command must be used for both memory space and I/O space.

Table 3–17: RDR usage for ALL fill cycles

Address bits <4:3>	Command used for fill cycle
00	RDR0
01	RDR1
10	RDR2
11	RDR3

3.3.5.1.7 Read data error cycles (RDE)

RDE is used to notify a commander of a problem with read data which is being returned. For example, the memory interface may use this command when it encounters an uncorrectable read error.

Once a Read Data Error cycle is sent for a particular read, no further read responses may be sent for that transaction. The following sequence illustrates the series of events during a return data of hexaword length containing an uncorrectable read error. In this example, HOLD is used to return the data in consecutive cycles.

Figure 3–11: RDE example

```
            0    1    2    3    4    5

Arb       |resp|HOLD|HOLD|    |    |    |
CMD_H     |    |RDR0|RDR1|RDE |    |    |
NDAL_H    |    |data|data|    |    |    |
ID_H      |    |cmdr|cmdr|cmdr|    |    |
ACK_L     |    |    |ACK |ACK |ACK |    |
```

3.3.5.1.8 Read data cycle description and timing

During a read data cycle, P%CMD_H<3:0> is driven to the value representing RDR0, RDR1, RDR2, RDR3, or RDE. P%NDAL_H<63:0> is driven with the quadword of read data being returned. P%ID_H<2:0> is driven with the ID which was issued with the original read request. Correct parity is driven on P%PARITY_H<2:0>.

The timing for a Read Data Return cycle is shown in Figure 3–12. In this example, IO1_NODE has fill data to return. In cycle 0, IO1_REQ_L is asserted to request the bus, and IO1_GRANT_L is asserted in response. Since IO1_GRANT_L was asserted in cycle 0, IO1_NODE is obligated to drive the NDAL in cycle 1. It does so and returns the fill data. The original requestor of the data receives the data at the beginning of phase 3 of cycle 1, and since it detects no parity errors, it asserts P%ACK_L so that it is valid in phi3 of cycle 2, as shown.

3.3.5.1.9 Read Transaction Examples

3.3.5.1.9.1 Quadword Read and Fill

A quadword read consists of a command transfer followed by a return data transfer as shown below:

Figure 3–12: NDAL Fill timing

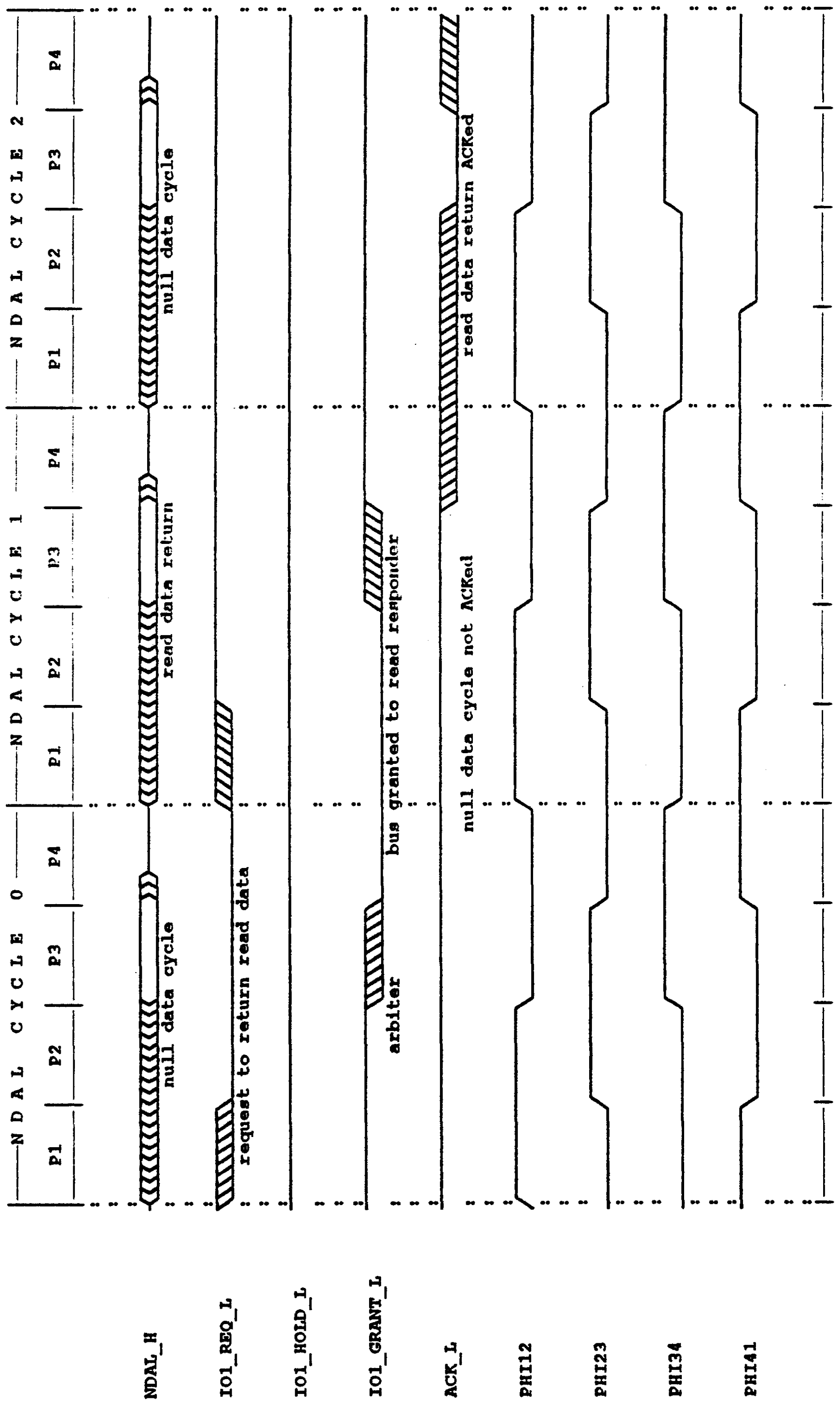


Figure 3–13: Quadword Read and Fill

```
               0     1     2     3                 4     5     6     7

Arb          |cmdr |     |     |     |           |resp |     |     |     |
CMD_H        |     |read |     |     |           |     |RDR2 |     |     |
NDAL_H       |     |addr |     |     |           |     |data |     |     |
ID_H         |     |cmdr |     |     |   ....    |     |cmdr |     |     |
ACK_L        |     |     | ACK |     |           |     |     | ACK |     |
```

The two transfers are the read command and the Read Data Return. The CPU commander arbitrates for the NDAL in cycle 0, and wins. In cycle 1 it drives the command and address of the read, and its own ID (for use later to identify the returning data). In cycle 2 the receiver for that cycle asserts **P%ACK_L** if no parity error was detected on the bus.

Sometime later (call it cycle 4) the return data transfer begins with the responder arbitrating for the NDAL. Having won it, in cycle 5 it drives the command, the data, and the commander's ID. The status of the returning data is specified in the read response code: either Read Data Return or Read Data Error. In this example, the quadword requested was to quadword 2 of a hexaword, so the RDR2 command is used in returning the data.

The commander monitors the NDAL and checks for an ID match during Read Data Return cycles. An ID match indicates that the read data is meant for that commander. In cycle 6, the commander asserts **P%ACK_L** if it detected no parity error during the previous NDAL cycle.

3.3.5.1.9.2 Multiple Quadword Reads

The only type of multiple quadword read which is used by NVAX is the hexaword read. Octaword reads are also supported by the NDAL protocol but are not issued by the NVAX CPU. These read transactions move multiple quadwords of data from the responder to the commander. The command transfer of the transaction is shown below.

Figure 3–14: Read command on the NDAL

```
               0     1     2     3

Arb          |cmdr |     |     |     |
CMD_H        |     |read |     |     |
NDAL_H       |     |addr |     |     |
ID_H         |     |cmdr |     |     |
ACK_L        |     |     | ACK |     |
```

The following sequence illustrates the response to a hexaword read. In this example, quadword 1 of the hexaword was the requested quadword, so Read Data Return 1 is the command accompanying the first data to return. The requested quadword is returned first for performance reasons, although that is not required by NVAX or the NDAL.

Figure 3–15: Read data return without using HOLD

```
               0      1      2      3      4      5      6      7

Arb         |resp  |      |resp  |      |resp  |      |resp  |      |      |      |
CMD_H       |      |RDR1  |      |RDR0  |      |RDR3  |      |RDR2  |      |      |
NDAL_H      |      |data  |      |data  |      |data  |      |data  |      |      |
ID_H        |      |cmdr  |      |cmdr  |      |cmdr  |      |cmdr  |      |      |
ACK_L       |      |      |ACK   |      |ACK   |      |ACK   |      |ACK   |      |
```

The transfer above moves four quadwords of data. The command field of the NDAL in cycle 1, 3, 5, and 7 says Read Data Return with the **P%ID_H** field identifying the intended receiver (the transaction commander). Each cycle provides a new quadword of read data and the **P%ID_H** remains unchanged.

The example shows no transactions interleaved with the Read Data Return cycles, but it is entirely possible for non-related transfers to be taking place in the cycles between the fill cycles for one read.

Read data may be returned in continuous cycles, if desired, through the use of the hold arbitration signals (see example below). The transmitter asserts its hold line in the first cycle to ensure that it maintains use of the NDAL long enough to complete the transfer. The hold lines are the highest priority arbitration lines and thus guarantee access. An interface is constrained to a maximum of four consecutive cycles in which it can assert its hold line.

Figure 3–16: Read data return using HOLD

```
               0      1      2      3      4

Arb         |resp  |hold  |hold  |hold  |      |      |      |
CMD_H       |      |RDR2  |RDR3  |RDR0  |RDR1  |      |      |
NDAL_H      |      |data  |data  |data  |data  |      |      |
ID_H        |      |cmdr  |cmdr  |cmdr  |cmdr  |      |      |
ACK_L       |      |      |ACK   |ACK   |ACK   |ACK   |      |
```

3.3.5.2 Writes

3.3.5.2.1 Normal Write Transactions (WRITE)

These transactions are used to move a pattern of bytes from an NDAL commander to one of the responders. The byte enable functionality is only used for quadword and octaword length transactions. In any hexaword write, all bytes are written regardless of the byte enable values.

Parity must be correct for all bytes sent from any node, as NVAX checks parity across the entire NDAL during every cycle.

If NVAX sees a write on the NDAL, it treats it as an invalidate request. A block invalidate is done if it is valid in the cache. A writeback is done if the block is owned.

3.3.5.2.2 Disown Write Transactions (WDISOWN)

The Disown Write transaction is the complement to the Ownership Read. After NVAX successfully gains ownership of a block in memory, it must relinquish ownership when another node wants ownership of the block or when the Bcache needs to do a deallocate. NVAX accomplishes this by performing a Disown Write to the memory with the latest copy of the data. The memory, which has been monitoring the bus traffic, notices that the transaction requested is a Disown Write. This condition allows it to clear the ownership bit in memory and to write the data as requested.

NVAX uses the Disown Write command of hexaword length to perform writebacks from the backup cache. When the cache is off, it uses quadword Disown Writes to achieve the effect of a Write Unlock.

3.3.5.2.3 Write Data and Bad Write Data (WDATA,BADWDATA)

The Write Data command is used during the data cycles of a write if the data is good. If the data has been corrupted in some way, for instance, there were uncorrectable errors in a cache which was storing the data, the command used is Bad Write Data.

When one quadword of a hexaword Write Disown is bad, the Bad Write Data command is only used for that quadword. The Write Data command is used for the good quadwords. The memory can use this information to distinguish which quadword of a hexaword block is bad. In addition, **P%S_ERR_L** may be asserted when the Bad Write Data command is used, to notify NVAX of the error.

3.3.5.2.4 Write transaction description and timing

In a Write transaction, a commander gains the NDAL and sends an address cycle. In this cycle, **P%CMD_H<3:0>** is driven to the value for WRITE. **P%NDAL_H<63:0>** is driven with the address, the transaction length, and byte enable. **P%ID_H<2:1>** is driven with the commander's identification code, and **P%ID_H<0>** is driven with any value.

The commander immediately follows this cycle with one to four consecutive cycles of write data, depending on the length specified. In these cycles, **P%CMD_H<3:0>** is driven with either the WDATA command or the BADWDATA command. **P%NDAL_H<63:0>** is driven with the write data. **P%ID_H<2:1>** is driven with the commander's identification code, and **P%ID_H<0>** must be driven, but may be driven with any value, as long as the parity is correct.

All interfaces on the NDAL decode the address, and the one that recognizes the address becomes the responder and asserts **P%ACK_L**. The responder accepts the command, address, and data and performs the requested write.

For quadword and octaword length transactions to memory space, the byte enable field that accompanies each command and address is completely unrestricted. Each bit in the 16-bit byte enable field corresponds to a byte of data in the associated quadword or octaword. If the bit is 0, that byte must not be written; if the bit is 1, that byte must be written. For hexaword write transactions, the responder ignores the byte enable and writes all 32 bytes.

For I/O space transactions, the byte enable is used as indicated in Section 3.3.4.1.2.

The timing for a quadword write on the NDAL is shown in Figure 3–17. In cycle 0, NVAX requests the bus for the write by asserting **P%CPU_REQ_L**. In this example, no higher priority request is pending, so NVAX is granted the bus right away, in cycle 0. NVAX then drives the write command and address in cycle 1, and asserts **P%CPU_HOLD_L** at the same time in order to retain the bus. In cycle 2 the write data is driven.

Assuming there are no parity errors, **P%ACK_L** is asserted by the receiver in cycle 2. This is in response to the address cycle of cycle 1. In cycle 3, which is not shown, **P%ACK_L** is asserted for the data cycle, cycle 2.

3.3.5.2.5 Write Transaction Examples

3.3.5.2.5.1 Quadword Writes

Quadword writes move some number of bytes from the commander to the responder as specified by the byte enable field. The commander arbitrates as usual and upon winning the NDAL, drives the appropriate write command, the intended address, the data byte enable, and its own ID and asserts its hold line to signal that it will need the next cycle also. In cycle two, it identifies the cycle as a Write Data Cycle and provides the write data. If an NDAL parity error is detected on cycle 1 or 2, it is signaled in cycle 2 or 3 by withholding the assertion of **P%ACK_L**.

The cycle timing for a quadword write is shown in Figure 3–18.

Figure 3–17: NDAL Write timing

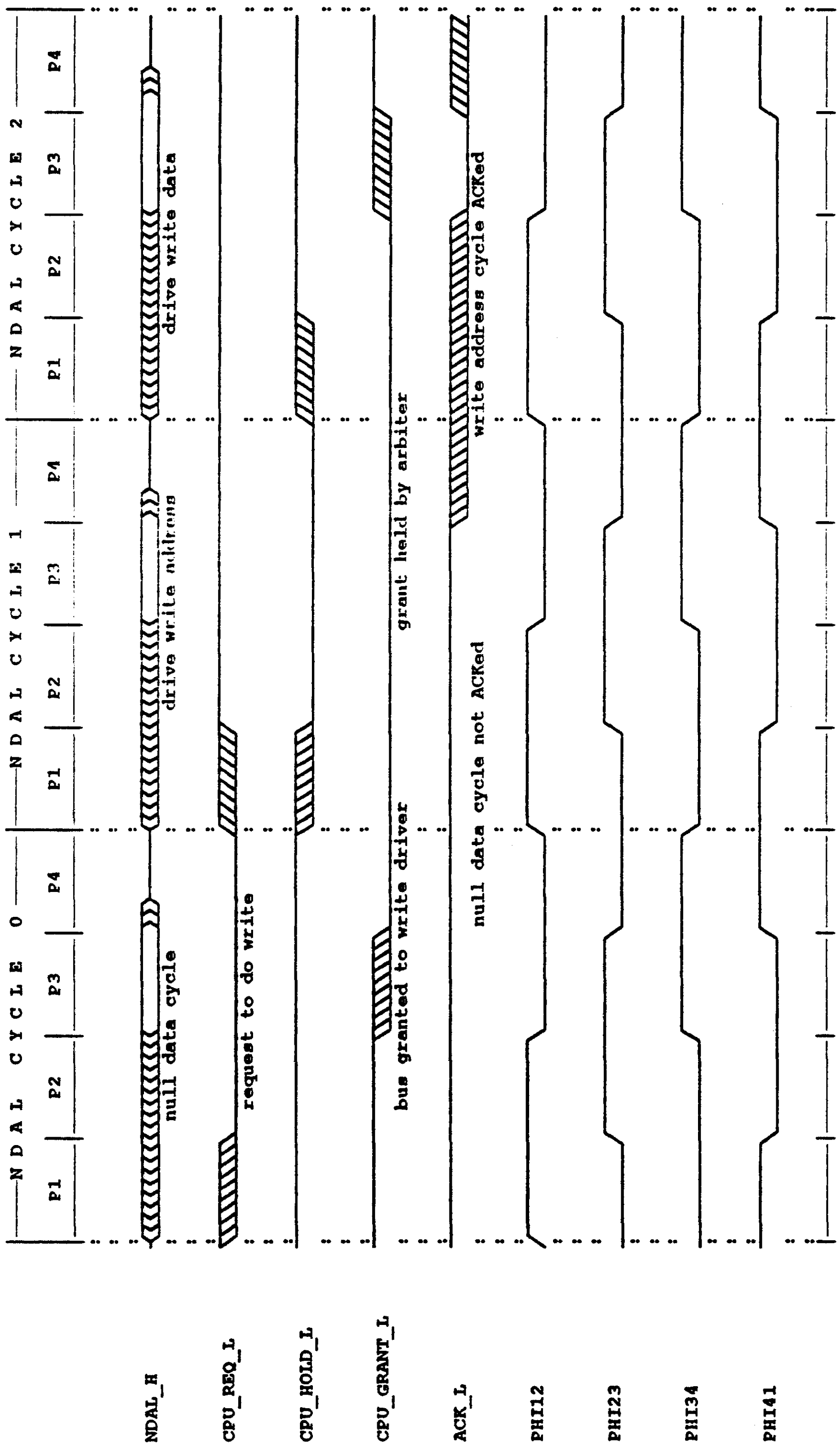


Figure 3–18: Quadword write on the NDAL

```
             0    1    2    3    4

Arb        |cmdr|HOLD|    |    |    |
CMD_H      |    |writ|wdat|    |    |
NDAL_H     |    |addr|data|    |    |
ID_H       |    |cmdr|cmdr|    |    |
ACK_L      |    |    |ACK |ACK |    |
```

3.3.5.2.5.2 Multiple Quadword Writes

The only multiple-data-cycle write issued by the NVAX CPU is the Hexaword Disown Write. Hexaword writes are similar to quadword writes except for the amount of data moved. The byte enable must be ignored in hexaword write transactions and all the bytes of the hexaword must be written.

The first cycle of a hexaword write is identified with the length desired; successive cycles are identified as write data cycles. The hold line remains asserted, maintaining use of the NDAL for the commander.

The four quadwords of data within the hexaword must be issued in order from lowest address to highest address. The order then is quadword 0, quadword 1, quadword 2, quadword 3. (Address bits <4:3> determine the position of a quadword within a hexaword.) Unlike fill data cycles, the same command, WDATA, is issued for every write data cycle, so the order in which the data is issued is essential so that it is written to the correct address in memory.

A hexaword write is shown in Figure 3–19.

Figure 3–19: Hexaword write on the NDAL

```
             0    1    2    3    4    5

Arb        |cmdr|hold|hold|hold|hold|    |    |    |
CMD_H      |    |wrt |wdat|wdat|wdat|wdat|    |    |
NDAL_H     |    |addr|dat0|dat1|dat2|dat3|    |    |
ID_H       |    |cmdr|    |    |    |    |    |    |
ACK_L      |    |    |ACK |ACK |ACK |ACK |ACK |    |
```

NOTE

> The write data must always immediately follow the write address cycle with no NULL cycles in between.

The NDAL protocol also allows for octaword writes. The NVAX CPU does not use these, but they may be used by other nodes.

The two quadwords of data within the octaword must be issued in order from lowest address to highest address. The order then is quadword 0, quadword 1. (Address bit <3> determines the position of a quadword within an octaword.)

An octaword write is shown in Figure 3–20.

Figure 3–20: Octaword write on the NDAL

```
            0    1    2    3    4    5

Arb       |cmdr|hold|hold|    |    |    |    |    |
CMD_H     |    |writ|wdat|wdat|    |    |    |    |
NDAL_H    |    |addr|dat0|dat1|    |    |    |    |
ID_H      |    |cmdr|    |    |    |    |    |    |
ACK_L     |    |    |ACK |ACK |ACK |    |    |    |
```

3.3.5.3 NOPs

For implementation reasons, occasionally NVAX will arbitrate for the NDAL and, if the bus is granted, it will drive a NOP. This only happens when NVAX has just driven out two back-to-back transactions. This happens rarely, and since NVAX has the lowest priority of the NDAL nodes, it is not a performance problem.

3.3.6 Cache Coherency

Ownership Reads and Disown Writes on the NDAL are intended to support writeback caches by attaching an owner status to each block in physical memory. A block in memory is defined as a hexaword, or 32 bytes. A node which owns a block may write it repeatedly without accessing memory. Only one node owns a given block. Ownership is passed from memory to a non-memory node through an Ownership Read command. Ownership is passed from non-memory nodes to memory through a Disown Write command.

The ownership bits in the caches and in memory indicate that a cache owns the block. The ownership bit in the writeback cache is set when the cache owns the block and is clear when the cache does not own the block. The ownership bit in memory is set when some cache owns the block and clear when memory owns the block.

Shared read-only access to a block is permitted only when memory owns it. Otherwise the block can only be read by the node which owns the block.

NVAX nodes with writeback caches can gain ownership and retain it for a very long time. NVAX monitors the bus continuously for memory space read-type and write commands to memory space by other nodes. When NVAX detects a request for a block that it owns, it will perform the disown write to memory, allowing the original command to complete successfully.

Table 3–18 shows what action is performed in the backup cache based upon the state of the block in the cache when a particular command is received.

Table 3–18: NVAX Backup Cache Invalidates and Writebacks

NDAL Command	Invalid block	Valid & Unowned	Valid & Owned
IREAD,DREAD	-	-	Writeback, set Bcache to valid-unowned state
OREAD	-	Invalidate	Writeback, Invalidate
WRITE	-	Invalidate	Writeback, Invalidate
WDISOWN	-	-	-

Some devices other than NVAX will access memory directly over the NDAL. As these commands go to memory, NVAX recognizes the command and performs the appropriate cache coherency action. NVAX does not acknowledge the commands as the memory interface is the receiver for the transaction. NVAX distinguishes cycles driven by devices other than itself by decoding the value driven on **P%ID_H<2:0>** for the cycle, and recognizes those as cache coherency transactions.

In some systems, such as the XMI2 system, there is a system bus to which multiple NVAX CPUs are interfaced. In these systems, memory commands which occur on the system bus must be driven into the NDAL so that NVAX can respond to them as necessary with cache coherency actions.

For example, if an OREAD happens on the XMI2, an OREAD must be driven onto the NDAL to trigger NVAX to write back the block if it owns it. However, there is no node on the NDAL which becomes a responder to a memory access transaction which is driven FROM the memory interface. The result is that **P%ACK_L** is not asserted to acknowledge such a transaction. This is not an error condition.

For more detail on the specific cache coherency requirements in the XMI2 system, refer to Section 3.4.1.

3.3.7 Interrupts

The **P%IRQ_L<3:0>** lines provide a general-purpose interrupt request facility to interrupt the NVAX CPU. These lines are level-sensitive, NOT edge sensitive. Once a node asserts its interrupt line, it should keep it asserted until NVAX services the request.

When NVAX receives an interrupt, it issues a read on the NDAL to one of four specified I/O space addresses. There is one address specified for each Interrupt Priority Level. This mechanism replaces the specific command, Read Interrupt Vector, which was used in previous systems.

Read cycles to these specified I/O space addresses are monitored by all nodes which have an interrupt outstanding. The node which responds first with a Read Data Return transaction will deassert its interrupt request.

Interrupting nodes on the NDAL do not have to deassert and reissue their interrupts after one node is serviced. The remaining nodes monitoring the bus see the return vector cycle and maintain their interrupt requests in anticipation of another NVAX I/O space read for an Interrupt Vector. If the common interrupt line remains asserted, NVAX will initiate another such cycle to be fielded by another first responder.

Chapter 10 describes interrupts in detail.

3.3.8 Clear Write Buffer

Clear Write Buffer is used to force all writes in the processor to be delivered to memory. In previous systems, an explicit Clear Write Buffer command on the pin bus was used. The NDAL uses an I/O space address which may be read or written to indicate that write buffers should be cleared.

The I/O space read is used when the CPU wishes an acknowledgement of the request. The CPU waits for the "read data" to return before continuing operation. The actual read data which is returned is meaningless except to allow the CPU to proceed. The I/O space read does not complete until all previous writes are complete. This mechanism may be used during a process context switch to force any errors associated with previous writes to happen in the context of the current process before the process context switch actually occurs.

The device which responds with read data to the Clear Write Buffer is system dependent. In theory it would be memory, since memory responding would indicate that all buffers before memory had been cleared.

The I/O space write which serves as Clear Write Buffer is used when the process mode changes but the process is not being switched. Here the purpose is to flush the writes as fast as possible when the mode changes, and to flush them ahead of any subsequent reads. Because the mode is changed often, it would be a performance hit to use the CWB read and to have to wait for the read data to return. Therefore the Clear Write Buffer is done as a write.

When the Cbox receives the clear write buffer command from the Mbox, it flushes its write queue. The writes are delivered to the backup cache, since it is writeback, rather than directly to memory. The I/O space clear write buffer command, whether a read or a write, is then issued on the NDAL.

3.3.9 VAX architecturally-defined interlocks

A VAX interlocked instruction causes the generation of a Read-Lock and a Write-Unlock which are guaranteed to happen back-to-back. The NDAL does not explicitly define interlocked transactions. Instead, the Ownership Read command is used in place of Read Lock and the Disown Write command is used in place of Unlock Write.

If the interlocked location is already owned in the backup cache when the Cbox receives the read lock from the Mbox, the command is never seen on the NDAL as it is serviced directly on the cache. Writeback of the block is prevented until the write unlock is issued from the Ebox.

3.3.9.1 Ownership and Interlock transactions

If NVAX has a read lock in progress and **P%CPU_WB_ONLY_L** is asserted, the CBOX issues the write unlock regardless of the assertion of **P%CPU_WB_ONLY_L**. Otherwise, deadlock might occur if **P%CPU_WB_ONLY_L** were asserted and a device in the system was waiting for NVAX to do a Write Unlock before deasserting **P%CPU_WB_ONLY_L**. For example, memory would not return Read Lock data to an I/O device if the ownership bit were set.

The NVAX CPU does not support interlocks to I/O space. If the Cbox receives an interlock to I/O space, it converts it to a normal read on the NDAL.

3.3.10 Errors

The NDAL supports the detection of all single-bit and some multiple-bit transmission related error conditions on the **P%NDAL_H**, **P%CMD_H**, **P%ID_H**, and **P%PARITY_H** lines by implementing parity across those lines. Additionally, the NDAL allows commanders to recover from some memory and I/O-space read/write class errors.

3.3.10.1 Transaction Timeout

Each NDAL node must implement a timeout counter for each read which it may have outstanding. The NVAX Cbox implements two timeout counters, one for each possible outstanding read. If a read request times out, it is aborted by the Cbox. Any missing Read Data Return cycles will eventually cause that read to timeout in the Cbox. See Table 3–21 for details on how timeout is handled.

The NVAX BIU starts its read timeout counter when it receives **P%ACK_L** assertion for the read. The counter is an 8-bit counter which, in normal operation, is clocked with a signal from the Ebox, E%TIMEOUT_ENABLE_H. The base counter in the Ebox is 16 bits wide. This implementation results in the timeout values shown in Table 3–19.

Table 3–19: NVAX Read Timeout Values in Normal Mode

NVAX chip speed	Timeout Granularity	Read timeout
10-ns NVAX	655 microseconds	167 milliseconds
12-ns NVAX	786 microseconds	200 milliseconds
14-ns NVAX	917 microseconds	234 milliseconds

A test mode for the NVAX read timeout counters is provided, and is described in detail in Chapter 13. In test mode, the read timeout counters are run directly from the internal NVAX clock, rather than from **E%TIMEOUT_ENABLE_H**. The test mode timeout values are shown in Table 3–20.

Table 3–20: NVAX Read Timeout Values in Test Mode

NVAX chip speed	Timeout Granularity	Read timeout
10-ns NVAX	10 nanoseconds	2.5 microseconds
12-ns NVAX	12 nanoseconds	3.0 microseconds
14-ns NVAX	14 nanoseconds	3.5 microseconds

The occurrence of transaction timeout is not normal and is expected to happen only when the system is broken.

More information on timeout may be found in Chapter 13.

3.3.10.2 Non-existent memory and I/O

An address which is not implemented in memory on a particular system is known as non-existent memory. An I/O address of a device which is not present on a particular system is known as non-existent I/O.

Devices on the NDAL must acknowledge any transactions to address space which they recognize by asserting **P%ACK_L** (except when there is a parity error). An address which is not recognized by any NDAL device is not acknowledged.

If **P%ACK_L** is not asserted in response to an NVAX request, the Cbox records the error by saving state in its error registers. (This error case is covered in Table 3–21). Software can read the error registers in the other NDAL nodes and find that the absence of **P%ACK_L** was not due to a parity error on the NDAL. From that information it can deduce that the problem was non-existent memory or I/O.

If an interface between the NDAL and another bus recognizes a read to some address and ACK's it, then finds that the address is not implemented on the other bus, the interface must use RDE to terminate the READ on the NDAL. It must not simply let the read time out, as this method of terminating the transaction takes much longer.

If an NDAL device ACK's a write, then determines that it was to non-existent memory or I/O space, it should notify the CPU appropriately. One possibility is to assert **P%H_ERR_L**.

3.3.10.3 Error Handling

This section describes the required behavior of NDAL commanders and responders in reaction to error conditions.

In general, NDAL errors are handled as follows:

- Null cycles have correct parity but are not acknowledged. The absence of **P%ACK_L** assertion for these cycles is not an error condition.
- Any NDAL receiver detecting bad parity in any field on a non-NULL cycle must ignore the cycle. **P%ACK_L** must not be asserted and no action should be taken in response to the NDAL command. The receiver may log the error. The device which drove the NDAL cycle must log the error (the absence of **P%ACK_L** assertion) and notify NVAX in some way, depending on the exact situation.
- If an NDAL responder returns Read Data Error for one quadword of return data, it must not send any further quadwords of data for that request. If any further fills are received, the Cbox treats them as unexpected fills as described in Table 3–21.
- On an ownership read, the memory should set the ownership bit as soon as it starts sending data back to the requestor. The NVAX backup cache does not set its ownership bit until it receives all the data for the block, so if any fill data is lost, the block will appear not to be owned by any element in the system. This simplifies error handling if NVAX did the OREAD because of a write, and the write data has already been written into the cache when the error occurs. No other device can get access to the block while the error is being handled.

- An NDAL memory node may not clear its ownership bit unless all write data cycles associated with the Disown Write transaction are properly received. If write data is sent with the BADWDATA command, it is considered to be properly received.

The NVAX BIU does not retry failed commands on the NDAL.

If the Cbox recognizes that data has been lost, it asserts C%CBOX_H_ERR_H to the Ebox. (In some cases, the data may be recoverable by software.) When C%CBOX_H_ERR_H is asserted, the Cbox always puts the Bcache into Error Transition Mode.

The Cbox asserts C%CBOX_S_ERR_H when it recognizes a soft error. A soft error does not necessarily interfere with code running on the machine. In some cases, the Cbox enters ETM upon recognizing a soft error.

Table 3–21 shows the response of the NVAX CPU for every error situation.

Table 3–21: NDAL Errors and NVAX CPU Error Responses

General Problem	Specific situation and action taken by NVAX CPU	
NVAX detects parity error on any NDAL cycle	P%ACK_L asserted (inconsistent parity error)	Cbox asserts C%CBOX_S_ERR_H, puts backup cache into Error Transition Mode. An invalidate or writeback request may have been missed.
	P%ACK_L not asserted (parity error)	Cbox asserts C%CBOX_S_ERR_H, puts backup cache into Error Transition Mode. An invalidate or writeback request may have been missed. [1]
P%ACK_L not asserted for NVAX-originated command	IREAD, DREAD (to memory or I/O)	Cbox aborts the read in the Cbox and the Mbox[2], asserts C%CBOX_S_ERR_H.
	OREAD	Cbox aborts the read in the Cbox and the Mbox, enters Error Transition Mode, and asserts C%CBOX_S_ERR_H. If the OREAD was done because of a write miss, the write will now be done straight to memory since the cache is in ETM.
	WRITE or WDISOWN, address cycle or data cycle (to memory or I/O)	Cbox asserts C%CBOX_H_ERR_H, enters Error Transition Mode. Data which should have been written to memory has been lost. If the error was on the data cycle and, in the system implementation, memory marks the data bad, software may choose to ignore the hard error response since the error will be detected when/if the data is read. NVAX continues to send the WDATA cycles even if the address cycle or one of the WDATA cycles is NAK'd.
Read timeout or Read Data Error before requested quadword is received	IREAD, DREAD (memory or I/O space)	Cbox aborts the read in the Cbox and the Mbox, asserts C%CBOX_S_ERR_H.
	OREAD	Cbox aborts the read in the Cbox and the Mbox, asserts C%CBOX_S_ERR_H, enters Error Transition Mode. The Cbox does not set the ownership bit in the cache. If memory has set its ownership bit, there is no record of ownership for the block in the system; however, software can analyze and clean up the problem by reading the Cbox error registers. If the OREAD was done because of a write miss, the write will now be done straight to memory since the cache is in ETM.

[1] In some systems, such as the NVAX XMI-2 system, commands may be sent on the NDAL purely to notify NVAX of an invalidate request; these commands are not acknowledged.

[2] The Cbox aborts the read in the Cbox by clearing valid bit in the FILL_CAM; it aborts the read in the Mbox by asserting C%CBOX_HARD_ERR_H with the I_CF or D_CF command.

Table 3–21 (Cont.): NDAL Errors and NVAX CPU Error Responses

General Problem	Specific situation and action taken by NVAX CPU	
Read timeout or Read Data Error after requested quadword successfully received	IREAD, DREAD	Cbox aborts the read in the Cbox and the Mbox, asserts C%CBOX_S_ERR_H, does not validate cache entry.
	OREAD for a read-modify or a read-lock	Cbox aborts the read in the Cbox and the Mbox, asserts C%CBOX_S_ERR_H, enters Error Transition Mode. The block is not validated or marked owned in the backup cache. Depending on system implementation, the ownership bit may be set in memory. If the OREAD was for a read-modify, software can analyze and correct the potential inconsistency in ownership information by reading the Cbox error registers. If the OREAD was for a read-lock, the write-unlock will follow to memory (as a quadword disown write) after the Cbox handles the error. If the memory subsystem has set its ownership bit, this write unlock preserves consistency in ownership in the memory subsystem; if not, the write unlock location appears to be owned by memory and will be handled as an error by memory.
	OREAD for a write	Cbox aborts the read in the Cbox, asserts C%CBOX_H_ERR_H, enters Error Transition Mode. The write was previously done into the cache when the requested quadword returned, since the Cbox merges the write data with the fill data. Since the read did not complete, the ownership bit is not set in the cache even though the new write data is in the cache. Software can recover the write data if it is non-shared data. The backup cache must be flushed of owned data using the deallocate register, then put into force hit mode. The data can then be read and written to memory. If the data is shared, writes to memory may have been done out of order by the Cbox, and system integrity is in question.
Read timeout or RDE on OREAD with pending writeback request		A pending writeback request is entered in the FILL_CAM when a writeback request arrives for an outstanding OREAD. If the OREAD does not complete successfully for any reason, the writeback request is aborted. The Cbox has not received the entire block, so it does not claim ownership for the block. Therefore, it does not write back the block as was requested.

Table 3–21 (Cont.): NDAL Errors and NVAX CPU Error Responses

General Problem	Specific situation and action taken by NVAX CPU
Unexpected fill or unexpected RDE received	If there is no corresponding FILL_CAM entry for a returning fill or RDE, the Cbox ignores the fill data. C%CBOX_H_ERR_H is asserted. The data is not placed in the Bcache and not sent to the Mbox.[3] CEFSTS is loaded and locked; the UNEXPECTED_FILL bit is set since the fill or RDE was unexpected.

[3] It is possible to create a scenario where an unexpected fill is received and is recognized by the Cbox because there is an entry in the FILL_CAM which apparently corresponds to the fill. For example, suppose the Cbox starts READ A. READ A times out, so the Cbox aborts it and the corresponding FILL_CAM entry is cleared. Now the Cbox starts READ B using the same ID as the aborted READ A. Now, if memory returns read data for A, it apparently corresponds to the fill cam entry for READ B. The data is accepted and NVAX is unknowingly operating with incorrect data. This behavior may cascade into READ C, READ D, etc., if the Cbox always has a new read outstanding by the time some unexpected data arrives. Eventually, however, the fill cam entry will be empty when read data is returned, and the Cbox will recognize the error. Before the Cbox recognizes the condition, NVAX may have been behaving very strangely, as it has probably been operating with either wrong Dstream or wrong Istream data.

Each system which uses the NVAX CPU chip will develop its own error strategy. In general, enough information should be logged so that software can understand the problem. Table 3–22 addresses system errors which the system designer should take into account.

Table 3–22: NDAL Errors and Error Responses by System Components

General Problem	Specific situation and considerations to be made	
NDAL parity error and **P%ACK_L** asserted	Node has cache	Assert **P%S_ERR_L** and disable the cache. The node may have missed an invalidate.
	Node has no cache	Assert **P%S_ERR_L**.
NDAL parity error and **P%ACK_L** not asserted		The lack of assertion of **P%ACK_L** is sufficient to notify the transmitter of the cycle; that transmitter is responsible for notifying the CPU of the error. If the transmitter lost a write, it should assert **P%H_ERR_L**.
	Any write or WDATA	The memory interface should not assert **P%H_ERR_L** because it cannot tell who sent the write. It should log the parity error. The transmitter which sent the write asserts **P%H_ERR_L** or takes other actions to initiate error recovery.
	WDATA for a Disown Write	The memory should not clear its OBIT; this way, reads from other CPUs will fail until software corrects the problem.
WDISOWN to memory location which memory owns		Response is system dependent. Memory should probably perform the write and log the error.

3.3.10.4 Error Recovery

In most cases an NDAL commander is permitted to reissue a failing transaction in order to recover from transient bus errors. Should the recovery fail (recovery may involve one or more reattempts of the failed transaction), then the commander logs a hard error. Implementation of error recovery is a system-dependent decision. This section contains guidelines on when a transaction may be retried.

- All transactions which do not receive **P%ACK_L** assertion for the address cycle may be retried.
- Any failing NDAL Write transaction may be retried.
- Any failing Read to memory space may be retried.
- Any failing I/O space Write transaction may be retried.
- It is unsafe to retry any I/O space Read transaction receiving a response timeout since some I/O devices may have read side effects.

The NVAX CPU will not implement retry on any NDAL transactions.

3.3.11 NDAL Initialization

When the NVAX CPU chip enters the reset state, the BIU does the following:

- Tristates **P%NDAL_H<63:0>**, **P%CMD_H<3:0>**, **P%ID_H<2:0>**, and **P%PARITY_H<2:0>**. This occurs when internal reset is asserted, and is not qualified with any clock.
- Releases **P%ACK_L**. This occurs when internal reset is asserted, and is not qualified with any clock.
- Deasserts **P%CPU_REQ_L**, **P%CPU_HOLD_L**, and **P%CPU_SUPPRESS_L**. This occurs when internal reset is asserted, and is not qualified with any clock.
- **P%CPU_GRANT_L** and **P%CPU_WB_ONLY_L** are sampled during reset.

While NVAX is asserting **P%SYS_RESET_L**, the NDAL clocks are running. **P%SYS_RESET_L** is deasserted relative to NDAL PHI12.

During reset, some NDAL node must drive the NDAL so that it is driven with a NOP and good parity by the time **P%SYS_RESET_L** is deasserted. NVAX receives the NDAL during reset. The NDAL must be driven to valid levels with good parity by the time reset is deasserted, to prevent NVAX from detecting a parity error. The following is an example of how to drive the NDAL with a NOP, while putting valid parity on the bus:

- Drive **P%CMD_H<3:0>** low (this is the NOP command).
- Drive **P%NDAL_H<63:0>** low.
- Drive **P%ID_H<2:0>** low.
- Drive **P%PARITY_H<2>** low.
- Drive **P%PARITY_H<1>** high.
- Drive **P%PARITY_H<0>** low.

The NVAX CPU does not assert **P%CPU_REQ_L** until at least 4 NDAL cycles after **P%SYS_RESET_L** is deasserted.

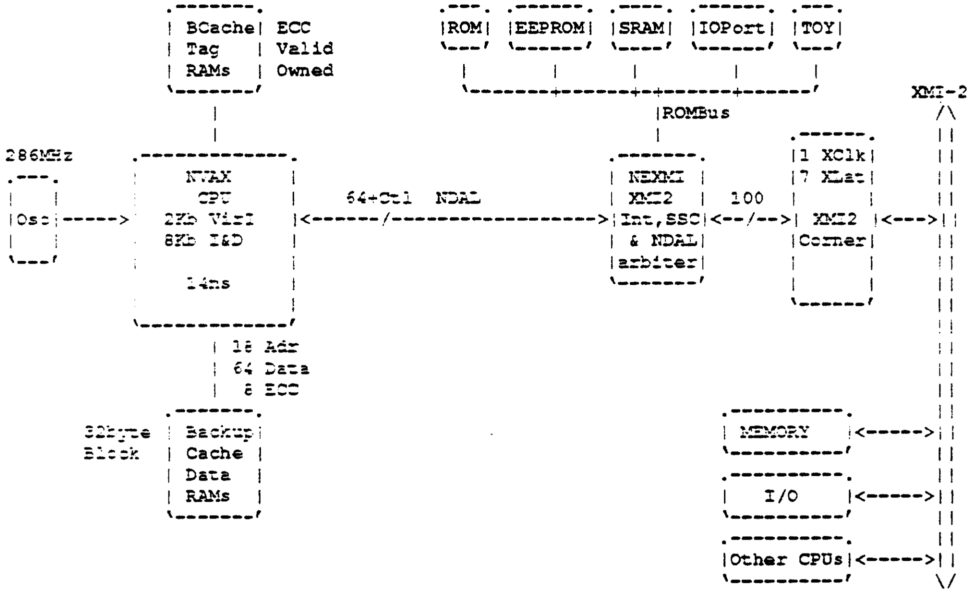
P%CPU_GRANT_L should be deasserted during system reset. NVAX will not drive the NDAL if granted during reset.

3.4 The XMI-2 NVAX System

A block diagram of the XMI-2 system is shown in Figure 3–21. Everything in the picture except memory, I/O, other CPUs, and the XMI-2 is contained on one module.

The XMI-2 system is being developed by MSB and is a follow-on to the Mariah XMI-2 system.

Figure 3–21: NVAX XMI-2 System Block Diagram



3.4.1 Cache coherency in the XMI2 system

Commands on the XMI2 must be forwarded to the CPU in order to maintain cache coherency. Table 3–23 shows the XMI2 commands and the corresponding command which must be forwarded on the NDAL to NVAX. The actions which Nvax takes as a result of the NDAL commands are shown in Section 3.3.6.

Table 3–23: XMI2-NVAX Coherency requirements

XMI2 Command	Resulting NDAL Command
Read	Dstream read
Interlock Read,Ownership Read	Ownership Read
Unlock Write, Write Masked	Write[1]
Disown Write,Tag Bad Data	none

[1] WDATA cycles for the write may be omitted since the write is driven onto the NDAL for cache coherency reasons only.

Unlock Writes must be forwarded to the NDAL for the following case. Assume an I/O device does a Read Lock, Write Unlock to memory location A. Assume that the CPU wants to do a normal read to location A, and that it does not have A in its cache. Assume the following timing on the XMI:

Figure 3–22: XMI2 Unlock Write example

```
time      I/O device          CPU
 |
 |        Interlock Read A
 |
 |                            Read A
 |
 |        Unlock Write A
 V
```

If the CPU reads A between the Read Lock and the Write Unlock the data the CPU caches should be invalidated after the write unlock. Otherwise, the CPU has stale data in its cache. This is because normal reads get data from XMI2 memory even if the location is interlocked. When writes are forwarded from the XMI2 to the NDAL, only the write address cycle must be driven. The write data cycles may be omitted.

3.5 The Lowend NVAX System - OMEGA

A block diagram of the lowend system, called Omega, is shown in Figure 3–23. The lowend system is being developed in Maynard, in ESB, the Entry Systems Business Group (formerly MVB).

Figure 3–23: NVAX Lowend System Block Diagram

```
              .-------.                                 .---------.
              | Bc    | ECC                             | O-bit   |
              | Tag   | Valid                           | memory  |
              | RAMs  | Owned                           \---------'
              \-------'                                      |
                  |                                          |
                  |                                     .---------.    .---.
286MHz     .-------------.                              |   NMC   |    |   |
.---.      |             |                              |memory   |    |   |
|   |      |     CPU     |         NDAL (64+ctl)        |interface|    |   |
|Osc|----->|  2Kb VirI   |<--------------+------------->|& NDAL   |<-->|MEM|
|   |      |  8Kb I&D    |               ^              |arbiter  |    |   |
\---'      |             |               |              |         |    |   |
           |    14ns     |               |              \---------'    \---'
           |             |               V
           \-------------'   .-----------------------.
                  | 18 Adr   |          NCA          |
                  | 64 Data  | (NVAX-CPbus Adapter)  |
                  |  8 ECC   |                       |
                  |          |NDAL-CDAL   NDAL-CDAL  |
              .-------.      |interface   interface  |
     32byte   |Backup |      \-----------------------'
     Block    |Cache  |           ^              ^      .--------.
              |Data   |           |              |      |        |  Qbus
              |Rams   |           |CP-bus        ------>| CQBIC  |<---------->
              \-------'           |              |      |        |
                                  |              |      \--------'
                                  |              |CP-bus
                                  V              |      .--------.
                                                 \----->| SSC    |
                                                        \--------'
```

The Lowend System implements an ownership bit in memory which is used to indicate that the NVAX CPU owns the block in its backup cache. This bit is covered by ECC. If an I/O interface issues a read or a write to a location which is owned by the NVAX backup cache, the memory interface holds the request until the writeback completes. It then completes the original transaction. The same applies to ownership transactions which may arrive from the NCA for an owned block of memory.

The NCA uses the NDAL ownership transactions in order to perform interlocked transactions.

One key problem in the Lowend System is the latency of a Qbus transaction. Once a device successfully issues a transaction on the Qbus, a timeout counter starts which will time out after 8 microseconds. This timing is difficult to meet in an NVAX system because of the writeback cache.

The analysis of the problem may be found in the specs for the Omega system.

3.6 Resolved Issues

1. Issue: Should we implement Force Bad Parity on the NDAL for testing purposes, or can we get away without it? Solution: We are implementing a way to force bad parity on the command field of the ndal.
2. Issue: The arbitration signals are not parity protected. Solution: This is not a problem because they are acknowledged by grant. The commander can always detect a problem by observing grant. If a request line is broken, the CPU will eventually timeout.
3. Issue: Should the Cbox do retry on parity errors? Solution: No. The XMI has never seen a parity error and it is a much longer bus with big connectors which we don't have. Retry would add unnecessary complexity.
4. Issue from Supnik: Allow space for extended addressing by moving byte enable over. Solution: Byte enable moved over.
5. Issue: Should parity be even or odd or a combination of both? Solution: Use even parity across the command, even parity across the lower longword of the NDAL, and odd parity across the upper longword of the NDAL. The combination helps for package reasons - all pins can't drive the same way at once. (Steve Thierauf)
6. Issue : Should the NDAL cycle time equal 2 or 3 CPU cycles? Solution: It will be much easier to design to 3 cycles so we'll do this in the interest of the schedule. 3 cycles may cost us 3% performance but it is worth it for ease of design.
7. Issue: Should NVAX drive the lower three bits of address for I/O space transactions? Solution: Yes. It is in the critical path of I/O devices to deduce the address from the byte enable.
8. Issue: If an Unlock Write transaction is directed to a location not currently locked, should the responder perform the write operation? Solution: This is a system-dependent issue. Recommendation added to the Errors section.
9. Issue: Should we have an acknowledged I/O space write? This would preserve write ordering between memory writes and I/O space writes. Solution: Historically this problem has not been addressed so our solving it is no value added. Software can be written which avoids the problem.
10. Issue: Do the lowend systems need byte parity? Solution: If a system is built without a backup cache, the performance is going to be poor so doing the read-modify-write for masked writes to memory is OK. The Lowend System will need to do read-modify-writes when the cache is in Error Transition Mode, but this is very rare. As long as there is time to compute longword parity it seems sufficient. Adding byte parity would increase the number of pins on the CPU and on all NDAL interfaces by 6.
11. Issue: There was not enough time for the arbiter if HOLD was a single open-drain signal. Solution: Have three hold signals, one for each commander, each of which is point-to-point.
12. Is parity enable necessary? If not, we get rid of a pin. Solution: Parity enable is not necessary. Every planned NVAX system is able to generate parity on every ndal cycle.
13. An additional command is under consideration. It would be called Disown Without Writeback (DISWOWB). It would be driven from the CPU to the memory interface after the CPU received a hexaword write to an owned block. DISWOWB indicates that the backup cache has given up ownership and invalidated the block, but is returning no data to memory. If a hexaword write is done in the system, memory has no use for the old data so it would be a waste of

time for the CPU to return it. Solution: This command does not appear to be useful enough to warrant the complexity.

14. Can the CPU chip remove the internal resistors on the NDAL? If we do, some chip in the system would have to pull the bus to valid levels during reset. Resolution: Yes, NVAX has removed the internal resistors. Another component in every NVAX system will pull the NDAL to valid levels during reset.

3.7 NVAX Chip Interface Signal Name Cross-Reference

All NVAX signal names and pin names referenced in this chapter have appeared in bold and reflect the actual name appearing in the NVAX schematic set. For each signal and pin appearing in this chapter, the table below lists the corresponding name which exists in the behavioral model.

Table 3–24: Cross-reference of all names appearing in the NVAX chip interface chapter

Schematic Name	Behavioral Model Name
C%CBOX_H_ERR_H	C%CBOX_H_ERR_H
C%CBOX_S_ERR_H	C%CBOX_S_ERR_H
C%CBOX_HARD_ERR_H	C%CBOX_HARD_ERR_H
E%TIMEOUT_ENABLE_H	E%TIMEOUT_ENABLE_H
P%ACK_L	P%ACK_L
P%ASYNC_RESET_L	P%ASYNC_RESET_L
P%CMD_H<3:0>	P%CMD_H<3:0>
P%CPU_GRANT_L	P%CPU_GRANT_L
P%CPU_HOLD_L	P%CPU_HOLD_L
P%CPU_REQ_L	P%CPU_REQ_L
P%CPU_SUPPRESS_L	P%CPU_SUPPRESS_L
P%CPU_WB_ONLY_L	P%CPU_WB_ONLY_L
P%DISABLE_OUT_L	P%DISABLE_OUT_L
P%DR_DATA_H<63:0>	P%DR_DATA_H<63:0>
P%DR_ECC_H<7:0>	P%DR_ECC_H<7:0>
P%DR_INDEX_H<20:3>	P%DR_INDEX_H<20:3>
P%DR_OE_L	P%DR_OE_L
P%DR_WE_L	P%DR_WE_L
P%HALT_L	P%HALT_L
P%H_ERR_L	P%H_ERR_L
P%ID_H<2:0>	P%ID_H<2:0>
P%INT_TIM_L	P%INT_TIM_L
P%IRQ_L<3:0>	P%IRQ_L<3:0>
P%MACHINE_CHECK_H	P%MACHINE_CHECK_H
P%NDAL_H<63:0>	P%NDAL_H<63:0>
P%OSC_H	P%OSC_H
P%OSC_L	P%OSC_L
P%OSC_TC1_H	P%OSC_TC1_H
P%OSC_TC2_H	P%OSC_TC2_H
P%OSC_TEST_H	P%OSC_TEST_H

Table 3–24 (Cont.): Cross-reference of all names appearing in the NVAX chip interface chapter

Schematic Name	Behavioral Model Name
P%PARITY_H<2:0>	P%PARITY_H<2:0>
P%PHI12_IN_H	P%PHI12_IN_H
P%PHI12_OUT_H	P%PHI12_OUT_H
P%PHI23_IN_H	P%PHI23_IN_H
P%PHI23_OUT_H	P%PHI23_OUT_H
P%PHI34_IN_H	P%PHI34_IN_H
P%PHI34_OUT_H	P%PHI34_OUT_H
P%PHI41_IN_H	P%PHI41_IN_H
P%PHI41_OUT_H	P%PHI41_OUT_H
P%PP_CMD_H<2:0>	P%PP_CMD_H<2:0>
P%PP_DATA_H<11:0>	P%PP_DATA_H<11:0>
P%PWRFL_L	P%PWRFL_L
P%SYS_RESET_L	P%SYS_RESET_L
P%S_ERR_L	P%S_ERR_L
P%TCK_H	P%TCK_H
P%TDI_H	P%TDI_H
P%TDO_H	P%TDO_H
P%TEMP_H	P%TEMP_H
P%TEST_DATA_H	P%TEST_DATA_H
P%TEST_STROBE_H	P%TEST_STROBE_H
P%TMS_H	P%TMS_H
P%TS_ECC_H<5:0>	P%TS_ECC_H<5:0>
P%TS_INDEX_H<20:5>	P%TS_INDEX_H<20:5>
P%TS_OE_L	P%TS_OE_L
P%TS_OWNED_H	P%TS_OWNED_H
P%TS_TAG_H<31:17>	P%TS_TAG_H<31:17>
P%TS_VALID_H	P%TS_VALID_H
P%TS_WE_L	P%TS_WE_L

3.8 Revision History

Table 3–25: Revision History

Who	When	Description of change
Rebecca Stamm	20-Feb-1991	Update after NVAX first pass. Clarified ACK timing. Added signal name cross-reference. Added NDAL timing AC spec. Corrected Byte Enable table. Updated Bcache pin timing. E%TIMEOUT_ENABLE_H clocks the Cbox timeout counter, not E%TIMEOUT_BASE_H. Added P% prefix to all pin names.
Rebecca Stamm	7-Nov-1990	PP_DATA are output only. Clarify NACK'd write handling. Correction: NVAX DOES receive the NDAL I/O signals during power-up.
Rebecca Stamm	4-Jul-1990	Update initialization description. Assert Herr on unexpected fill. Update ndal pin timing. NVAX may drive NOPs under WB_ONLY. **P%ID_H<0>** not driven with same value during command and data cycles of a write. Close force_bad_parity issue.
Rebecca Stamm	17-May-1990	Take out vector pins, add two new test pins, update description of unexpected fill handling by setting CEFSTS<UNEXPECTED_FILL>.
Rebecca Stamm	20-Feb-1990	Add unexpected RDE handling. Clarified byte enables and octaword-length transactions. Corrected running total for NVAX pins. Add detailed timeout description. Added timeout functionality to **P%OSC_TC1_H**.
Rebecca Stamm	3-Feb-1990	External release. Updates from internal review. Address<2:0> is sent out as zeros for the second half of an unaligned I/O space reference. NVAX does not implement internal resistors to pull the NDAL to valid levels during reset; a system device must drive the bus during reset.
Rebecca Stamm	30-Jan-1990	Reorganized chapter. Clarified byte enable section. NVAX issues identical data on both halves of the bus during I/O space writes. Released for internal review.
Rebecca Stamm	01-Dec-1989	Revision 1.0 release. Clarified byte enable table. Added error handling for unexpected fills. Added error handling for requested writebacks whose OREADs do not complete.
Rebecca Stamm	06-Mar-1989	Release for external review.

Table 3–25 (Cont.): Revision History

Who	When	Description of change
Rebecca Stamm	24-Oct-1989	Several NVAX pins were added, deleted, or changed in either name or functionality. The terminology byte mask is changed to byte enable. IO1_WB_ONLY, IO2_WB_ONLY, IO1_SUPPRESS, and IO2_SUPPRESS were added, and NDAL arbitration was changed, giving the arbiter responsibility for asserted the appropriate WB_ONLY lines when a SUPPRESS line is asserted. Addition of BADWDATA command. New command encodings. Elimination of Read Lock and Write Unlock commands on the NDAL. Add better explanation of Clear Write Buffer. Update error section. Remove PARITY_ENABLE_L pin. Removed Qbus latency problem description. Assigned an ID to the memory interface. Read data may be returned in any order: NVAX does not require the requested quadword first, although it is a performance advantage to return the requested qw first.

Chapter 4

Chip Overview

4.1 NVAX CPU Chip Box and Section Overview

The NVAX CPU Chip is a single-chip CMOS-4 macropipelined implementation of the base instruction group, and the optional vector instruction group of the VAX architecture. Included in the chip are:

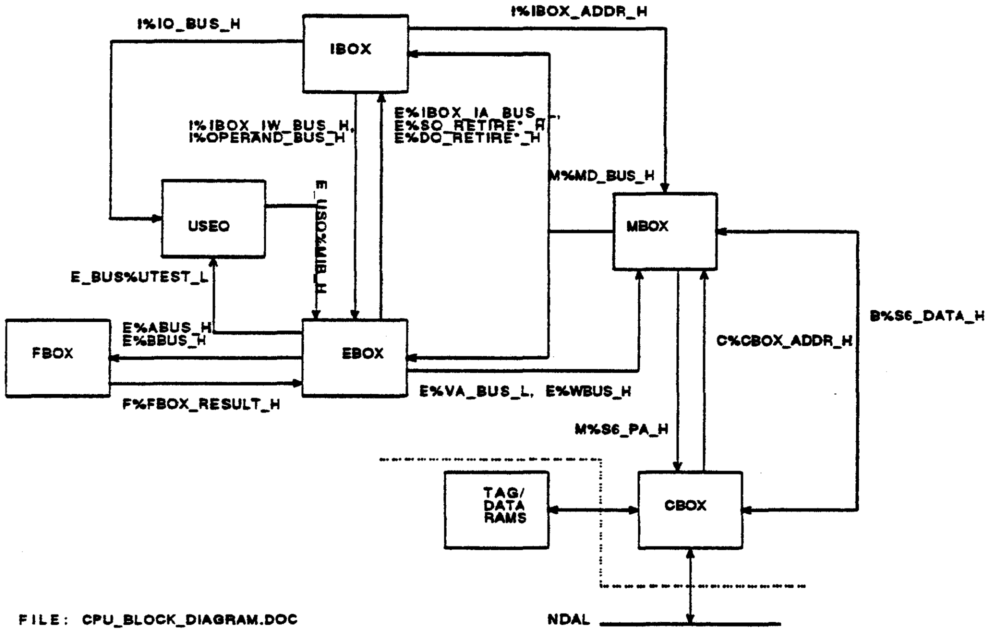
- **CPU:** Instruction fetch and decode, microsequencer, and execution unit
- **Control Store:** 1600, 61-bit microwords
- **Primary Cache:** 8 KB, 2-way set associative, physically-addressed, write through, mixed instruction and data stream
- **Instruction Cache:** 2 KB, direct-mapped , virtually addressed, instruction stream only
- **Translation Buffer:** 96 entries, fully associative
- **Floating Point:** 4 stage, pipelined, integrated floating point unit with selective stage 4 bypass
- **Backup Cache Interface:** Support for four cache sizes (2MB, 512KB, 256KB, 128KB), two tag RAM speeds and three data RAM speeds.
- **NDAL Interface:** Memory subsystem interface. Supports an ownership coherence protocol on the Backup Cache

The NVAX chip is designed in CMOS-4 with a typical cycle time of 14 ns, and with the option of running chips at a slower or faster cycle time. The chip can be incorporated into many different system environments, ranging from the desktop to the midrange, and from single processor to multiprocessor systems.

The NVAX is a macropipelined design: it pipelines macroinstruction decode and operand fetch with macroinstruction execution. Pipeline efficiency is increased by queuing up instruction information and operand values for later use by the execution unit. Thus, when the macropipeline is running smoothly, the Ibox (instruction parser/operand fetcher) is running several macroinstructions ahead of the Ebox (execution unit). Outstanding writes to registers or memory locations are kept in a scoreboard to ensure that data is not read before it has been written. See Chapter 5 for a more in-depth discussion of the macropipeline.

This chapter gives an overview of the different sections, or "boxes", that comprise the NVAX CPU. For more information on any of the boxes, please see the appropriate chapters within this specification. Figure 4–1 is a block diagram of the boxes, and the major buses that run between them.

Figure 4–1: NVAX CPU Block Diagram



4.1.1 The Ibox

The Ibox decodes VAX instructions and parses operand specifiers. Instruction control, such as the control store dispatch address, is then placed in the instruction queue for later use by the Microsequencer and Ebox. The Ibox processes the operand specifiers at a rate of one specifier per cycle and, as necessary, initiates specifier memory read operations. All the information needed to access the specifiers is queued in the source queue and destination queue in the Ebox.

The Ibox prefetches instruction stream data into the prefetch queue (PFQ), which can hold 16 bytes. The Ibox has a dedicated instruction-stream-only cache, called the virtual instruction cache (VIC). The VIC is a 2 KB, direct-mapped cache, with a block and fill size of 32 bytes.

The Ibox has both read and write ports to the GPR and MD portions of the Ebox register file which are used to process the operand specifiers. The Ibox maintains a scoreboard to ensure that reads and writes to the register file are always performed in synchronization with the Ebox. The Ibox stops processing instructions and operands upon issuing certain complex instructions (for example, CALL, RET, and character string instructions). This is done to maintain read/write ordering when the Ebox will be altering large amounts of VAX state.

Since the Ibox is often parsing several macroinstructions ahead of the Ebox, the correct value for the PSL condition codes is not known at the time the Ibox executes a conditional branch instruction. Rather than emptying the pipe, the Ibox predicts which direction the branch will take, and passes this information on to the Ebox via the branch queue. The Ebox later signals if there was a misprediction, and the hardware backs out of the path. The branch prediction algorithm utilizes a 512-entry RAM, which caches four bits of branch history per entry.

4.1.2 The Ebox and Microsequencer

The Ebox and Microsequencer work together to perform the actual "work" of the VAX instructions. Together they implement a four stage micropipelined unit, which has the ability to stall and to microtrap. The Ebox and Microsequencer dequeue instruction and operand information provided by the Ibox via the instruction queue, the source queue, and the destination queue. For literal type operands, the source queue contains the actual operand value. In the case of register, memory, and immediate type operands, the source queue holds a pointer to the data in the Ebox register file. The contents of memory operands are provided by the Mbox based on earlier requests from the Ibox. GPR results are written directly back to the register file. Memory results are sent to the Mbox, where the data will be matched with the appropriate specifier address previously sent by the Ibox. At times, the Ebox initiates its own memory reads and writes using E%VA_BUS_L and E%WBUS_H.

The Microsequencer determines the next microword to be fetched from the control store. It then provides this cycle-by-cycle control to the Ebox. The Microsequencer allows for eight-way microbranches, and for microsubroutines to a depth of six.

The Ebox contains a five-port register file, which holds the VAX GPRs, six Memory Data Registers (MDs), six microcode working registers, and ten miscellaneous CPU state registers. It also contains an ALU, a shifter, and the VAX PSL. The Ebox uses the RMUX, controlled by the retire queue, to order the completion of Ebox and Fbox instructions. As the Ebox and the Fbox are distinct hardware resources, there is some amount of execution overlap allowed between the two units.

The Ebox implements specialized hardware features in order to speed the execution of certain VAX instructions: the population counter (CALLx, PUSHR, POPR), and the mask processing unit (CALLx, RET, FFx, PUSHR, POPR). The Ebox also has logic to gather hardware and software interrupt requests, and to notify the Microsequencer of pending interrupts.

4.1.3 The Fbox

The Fbox implements a four stage pipelined execution unit with selective stage 4 bypass for the floating point and integer multiply instructions. Operands are supplied by the Ebox up to 64 bits per cycle on E%ABUS_H and E%BBUS_H. Results are returned to the Ebox 32 bits per cycle on F%FBOX_RESULT_H. The Ebox is responsible for storing the Fbox result in memory or the GPRs.

4.1.4 The Mbox

The Mbox receives read requests from the Ibox (both instruction stream and data stream) and from the Ebox (data stream only). It receives write/store requests from the Ebox. Also, the Cbox sends the Mbox fill data and invalidates for the Pcache. The Mbox arbitrates between these requesters, and queues requests which cannot currently be handled. Once a request is started, the Mbox performs address translation and cache lookup in two cycles, assuming there are no misses or other delays. The two-cycle Mbox operation is pipelined.

The Mbox uses the translation buffer (96 fully associative entries) to map virtual to physical addresses. In the case of a TB miss, the memory management hardware in the Mbox will read the page table entry and fill the TB. The Mbox is also responsible for all access checks, TNV checks, M-bit checks, and quadword unaligned data processing.

The Mbox houses the Primary Cache (Pcache). The Pcache is 8KB, 2-way set associative and writethrough, with a block and fill size of 32 bytes. The Pcache state is maintained as a subset of the Backup Cache.

The Mbox ensures that Ibox specifier reads are ordered correctly with respect to Ebox specifier stores. This memory "scoreboarding" is accomplished by using the PA queue, a small list of physical addresses which have a pending Ebox store.

4.1.5 The Cbox

The Cbox is the controller for the second level cache (the Backup Cache, or Bcache). Both the tags and data for the Bcache are stored in off-chip RAMs. The size and access time of the Bcache RAMs can be configured as needed by different system environments. The Bcache sizes supported are 2 MB, 512 KB, 256 KB, and 128 KB. In addition, a system with no Bcache RAMs is supported, although significant performance degradation occurs without a Bcache. The Bcache is a direct mapped writeback cache with block and fill sizes of 32 bytes. The Cbox packs sequential writes to the same quadword in order to minimize Bcache write accesses. Multiple write commands are held in the eight-entry WRITE_QUEUE.

The Cbox is also the interface to the NDAL, which is the NVAX connection to the memory subsystem. The NDAL_IN_QUEUE loads fill data and writeback requests from the NDAL to the CPU. The NON_WRITEBACK_QUEUE and WRITEBACK_QUEUE hold read requests and writeback data to be sent to the memory subsystem over the NDAL.

4.1.6 Major Internal Buses

This is a list of the major interbox buses:

- **B%S6_DATA_H:**
  This bidirectional bus between the Cbox and MBox is used to transfer write data to the backup cache, to to transfer fill data to the primary cache.
- **C%CBOX_ADDR_H:**
  This bus is used to transfer the physical address of a Pcache invalidate from the Cbox to the MBox.
- **E%ABUS_H, E%BBUS_H:**
  These two 32-bit buses contain the A- and B-port operands for the Ebox, and are also used to transfer operand data to the Fbox.

- **E%IBOX_IA_BUS_L**:
  This bus is used by the Ibox to read the Ebox Register File in order to perform an operand access. An example is to read a register's contents for a register deferred type specifier.
- **E%DQ_RETIRE_H, E%DQ_RETIRE_RMODE_H, E%DQ_RETIRE_RN_H**:
  This collection of related buses transfers information from the Ebox to the Ibox when a destination queue entry is retired.
- **E%SQ_RETIRE_H, E%SQ_RETIRE_MD_H, E%SQ_RETIRE_RMODE_H, E%SQ_RETIRE_RN1_H, E%SQ_RETIRE_RN2_H**:
  This collection of related buses transfers information from the Ebox to the Ibox when a source queue entry is retired.
- **E%VA_BUS_L**:
  This bus transfers an address from the Ebox to the MBox.
- **E%WBUS_H**:
  This 32-bit bus transfers write data from the RMUX to the register file and the Mbox.
- **E_USQ%MIB_H**:
  This bus carries Control Store data from the Microsequencer to the Ebox.
- **E_BUS%UTEST_L**:
  This 3-bit bus transfers microbranch conditions from the Ebox to the microsequencer.
- **F%FBOX_RESULT_H**:
  This bus is used to transfer results from the Fbox to the Ebox.
- **I%IBOX_ADDR_H**:
  This bus transmits the virtual address of an Ibox memory reference to the Mbox. The address may be for instruction prefetch or an operand access.
- **I%IQ_BUS_H**:
  This bus carries instruction information from the Ibox to the Instruction Queue in the Microsequencer.
- **I%IBOX_IW_BUS_H**:
  This bus is used by the Ibox to write the Ebox Register File for autoincrement/decrement type specifiers and to deliver immediate operands to the Register File.
- **I%OPERAND_BUS_H**:
  This bus transfers information from the Ibox to the source and destination queues in the Ebox.
- **M%MD_BUS_H**:
  The bus returns right-justified memory read data from the Mbox to either the Ibox (64 bits) or the Ebox (32 bits).
- **M%S6_PA_H**:
  This bus transfers the address for a backup cache reference from the MBox to the Cbox.
- NDAL:
  The NDAL are bidirectional off-chip multiplexed address and data lines used by the Cbox to communicate with the memory subsystem. The NDAL carries fill data and writeback requests to the CPU, and writeback data and read requests from the CPU to memory.

4.2 Revision History

Table 4–1: Revision History

Who	When	Description of change
Debra Bernstein	06-Mar-1989	Release for external review.
Mike Uhler	18-Dec-1989	Update for second-pass release.
Mike Uhler	04-Dec-1990	Update after pass 1 PG.

Chapter 5

Macroinstruction and Microinstruction Pipelines

5.1 Introduction

This chapter discusses the architecture of the NVAX CPU macroinstruction and microinstruction pipeline. It includes a section of general pipeline fundamentals to set the stage for the specific NVAX CPU implementation of the pipeline. This is followed by an overview of the NVAX CPU pipeline, an examination of macroinstruction execution, and a discussion of stall and exception handling from the viewpoint of the Ebox.

5.2 Pipeline Fundamentals

This section discusses the fundamentals of instruction pipelining in a general manner that is independent of the NVAX CPU implementation. It is intended as a primer for those readers who do not understand the concept and implications of instruction pipelining. Readers familiar with this material are encouraged to skip (or at most skim) this section.

5.2.1 The Concept of a Pipeline

The execution of a VAX macroinstruction involves a sequence of steps which are carried out in order to complete the macroinstruction operation. Among these steps are: instruction fetch, instruction decode, specifier evaluation and operand fetch, instruction execution, and result store. On the simplest machines, these steps are carried out sequentially, with no overlap of the steps, as shown in Figure 5–1.

Figure 5–1: Non-Pipelined Instruction Execution

```
                --------------- Time ---------------->

                +--------------------+
Instruction 1   |S0|S1|S2|S3|S4|S5|S6|
                +--------------------+
                                     +--------------------+
Instruction 2                        |S0|S1|S2|S3|S4|S5|S6|
                                     +--------------------+
                                                          +--------------------+
Instruction 3                                             |S0|S1|S2|S3|S4|S5|S6|
                                                          +--------------------+
```

In this diagram, "S0", "S2", ..., "S6" denote particular steps in the execution of an instruction. For this simple scheme, all of the steps for one instruction are performed, and the instruction is completed, before any of the steps for the next instruction are started.

In more complex machines, one or more steps of the execution process are carried out in parallel with other steps. For example, consider Figure 5–2.

Figure 5–2: Partially-Pipelined Instruction Execution

```
                --------------- Time ---------------->

                +--------------------+
Instruction 1   |S0|S1|S2|S3|S4|S5|S6|
                +--------------------+
                                  +--------------------+
Instruction 2                     |S0|S1|S2|S3|S4|S5|S6|
                                  +--------------------+
                                                    +--------------------+
Instruction 3                                       |S0|S1|S2|S3|S4|S5|S6|
                                                    +--------------------+
```

In this example, step S6 of each instruction is overlapped in time (or executed in parallel) with step S0 of the next instruction. In doing so, the number of instructions executed per unit time (instruction throughput) goes up because an instruction appears to take less time to complete.

In the most complex machines, most (or all) of the steps are executed in parallel as indicated in Figure 5–3.

Figure 5–3: Fully-Pipelined Instruction Execution

```
                --------------- Time --------------->

                +--------------------+
Instruction 1   |S0|S1|S2|S3|S4|S5|S6|
                +--------------------+
                   +--------------------+
Instruction 2      |S0|S1|S2|S3|S4|S5|S6|
                   +--------------------+
                      +--------------------+
Instruction 3         |S0|S1|S2|S3|S4|S5|S6|
                      +--------------------+
                         +--------------------+
Instruction 4            |S0|S1|S2|S3|S4|S5|S6|
                         +--------------------+
                            +--------------------+
Instruction 5               |S0|S1|S2|S3|S4|S5|S6|
                            +--------------------+
```

In this example every step of instruction execution is performed in parallel with every other step. This means that a new instruction is started as soon as step S0 is completed for the previous instruction. If each step, S0..S6, took the same amount of time, the apparent instruction throughput would be seven times greater than that of Figure 5–1 above, even though each instruction takes the same amount of time to execute in both cases.

Figures 5–2 and 5–3 are examples of the concept of instruction pipelining, in which one or more steps necessary to execute an instruction are performed in parallel with steps for other instructions.

5.2.2 Pipeline Flow

A real-world form of a pipeline is an automobile assembly line. At each station of the assembly line (called segments of the pipeline in our case), a task is performed on the partially completed automobile and the result is passed on to the next station. At the end of the assembly line, the automobile is complete.

In an instruction pipeline, as in an assembly line, each segment is responsible for performing a task and passing the completed result to the next segment. The exact task to be performed in each pipeline segment is a function of the degree of pipelining implemented and the complexity of the instruction set.

One attribute of an automobile assembly line is equally important to an instruction pipeline: smooth and continuous flow. An automobile assembly line works well because the tasks to be performed at each station take about the same amount of time. This keeps the line moving at a constant pace, with no starts and stops which would reduce the number of completed automobiles per unit time.

An analogous situation exists in an instruction pipeline. In order to achieve real efficiency in an instruction pipeline, information must flow smoothly and continuously from the start of the pipeline to the end. If a pipeline segment somewhere in the middle is not able to supply results to the next segment of the pipeline, the entire pipeline after the offending segment must stop, or stall, until the segment can supply a result.

In the general case, a pipeline stall results when a pipeline segment can not supply a result to the next segment, or when it can not accept a new result from a previous segment.

This is a fundamental problem with most instruction pipelines because they occasionally (or not so occasionally) stall. Stalls result in decreased instruction throughput because the smooth flow of the pipeline is broken.

A typical example of a pipeline stall involves memory reads. A simple three-segment pipeline might fetch operands in segment 1, use the operands to compute results in segment 2, and make memory references or store results in segment 3, as shown in Figure 5–4.

Figure 5–4: Simple Three-Segment Pipeline

```
+------------+  +------------+  +------------+
|  Operand   |->|Computation |->|  Memory    |
|   Access   |  |            |  |   Read     |
+------------+  +------------+  +------------+
```

Figure 5–5 illustrates what happens when the pipeline control wants to use the result of the memory read as an operand.

Figure 5–5: Information Flow Against the Pipeline

```
      +------------+  +------------+  +------------+
I1    |  Operand   |->|Computation |->|  Memory    |-----+
      |   Access   |  |            |  |   Read     |     |
      +------------+  +------------+  +------------+     |
               ------------------------------------------+
               |      +------------+  +------------+  +------------+
I2             +----->|  Operand   |->|Computation |->|   Result   |
                      |   Access   |  |            |  |   Store    |
                      +------------+  +------------+  +------------+
```

In this case, the operand access segment of I2 can not supply an operand to the computation segment because the memory read done by I1 has not yet completed. As a result, the pipeline must stall until the memory read has completed. This is shown in Figure 5–6.

Figure 5–6: Stalls Introduced by Backward Pipeline Flow

```
        +------------+  +------------+  +------------+
I1      | Operand    |->|Computation |->|   Memory   |-----+
        |  Access    |  |            |  |    Read    |     |
        +------------+  +------------+  +------------+     |
                  +----------------------------------------+
                  |     +------------+  +------------+  +------------+
I2                +---->|   Stall    |->|   Stall    |->|   Stall    |
                  |     |            |  |            |  |            |
                  |     +------------+  +------------+  +------------+
                  |                     +------------+  +------------+  +------------+
I2                +-------------------->|   Stall    |->|   Stall    |->|   Stall    |
                  |                     |            |  |            |  |            |
                  |                     +------------+  +------------+  +------------+
                  |                                     +------------+  +------------+  +------------+
I2                +------------------------------------>| Operand    |->|Computation |->|   Result   |
                                                        |  Access    |  |            |  |   Store    |
                                                        +------------+  +------------+  +------------+
```

In this diagram, the memory read data from I1 is not available until the read request passes through segment 3 of the pipeline. But the operand access segment for I2 wants the data immediately. The result is that the operand access segment of I2 has to stall twice waiting for the memory read data to become available. This, in turn, stalls the rest of the pipeline segments after the operand access segment.

This situation is an excellent example of an age-old problem with instruction pipelining. The natural and desired direction of information flow in a pipeline is from left to right in the above diagrams. In this case, information must flow from the output of the memory read segment into the operand access segment. This requires a right-to-left movement of information from a later pipeline segment to an earlier one. In general, any information transfer which goes against the normal flow of the pipeline has the potential for causing pipeline stalls.

5.2.3 Stalls and Exceptions in an Instruction Pipeline

Even the best pipeline design must be prepared to deal with stalls and exceptions created in the pipeline. As mentioned above, a stall is a condition in which a pipeline segment can not accept a new result from a previous segment, or can not send a result to a new segment. An exception occurs when a pipeline segment detects an abnormal condition which must stop, and then drain the pipeline. Examples of exceptions are: memory management faults, reserved operand faults, and arithmetic overflows. One of the inherent costs of a pipelined implementation is the extra logic necessary to deal with stalls and exceptions.

There are two primary considerations concerning stalls: what action to take when one occurs, and how to minimize them in the first place. The design of most instruction pipelines assumes that the pipeline will not stall, and handles the stall condition as a special case, rather than the other way around. This means that each segment of the pipeline performs its function and produces a result each cycle. If a stall occurs just before the end of the cycle, the segment must block global state updates and repeat the same operation during the next cycle. The design of the pipeline control must take this into account and be prepared to handle the condition.

A common stall condition occurs when each pipeline segment has the same average speed, but different peak speeds. For example, a pipeline segment whose task is to perform both memory references and register result stores may take longer to perform memory references than result stores. This can cause earlier segments of the pipeline to stall because the segment can not take new inputs as fast if it is doing a memory reference rather than a result store. A common technique to minimize this problem is to place buffers between pipeline segments, as shown in Figure 5–7.

Figure 5–7: Buffers Between Pipeline Segments

```
+------------+ +-------+ +------------+ +-------+ +------------+
|  Operand   |->|Buffer|->|Computation|->|Buffer|->|  Memory   |
|   Access   | |       | |            | |       | |   Read     |
+------------+ +-------+ +------------+ +-------+ +------------+
```

By placing a buffer of sufficient depth between each segment of the pipeline, segments of differing peak speeds can avoid stalls caused if the next segment is unable to accept a new result. Instead, the result goes into the inter-segment buffer and the next segment removes it from the buffer when it needs it. Unfortunately, adding such buffers means that additional logic must also be added to handle the buffer full/buffer empty conditions.

The performance advantage of an instruction pipeline comes from the parallelism built into the pipeline. If the parallelism is defeated by, for example, a stall, the advantage starts to drop. One problem associated with pipelines is that they can provide "lumpy" performance. That is, two similar programs may experience radically different performance if one causes many more stalls (which defeat the parallelism of the pipeline) than the other.

Pipeline exceptions are different from stalls in that exceptions cause the pipeline to empty or drain. Usually, everything that entered the pipeline before the point of error is allowed to complete. Everything that entered the pipeline after the point of error is prevented from completing. This can add considerable complexity to the pipeline control.

A larger problem occurs when the designer wants exceptions to be recoverable. Consider an exception caused by a memory management fault. On the VAX, this condition can occur because of a TB miss. The correct response to this fault is to read a PTE from memory, refill the TB, and restart the request that caused the fault. This can add considerable complexity to the design.

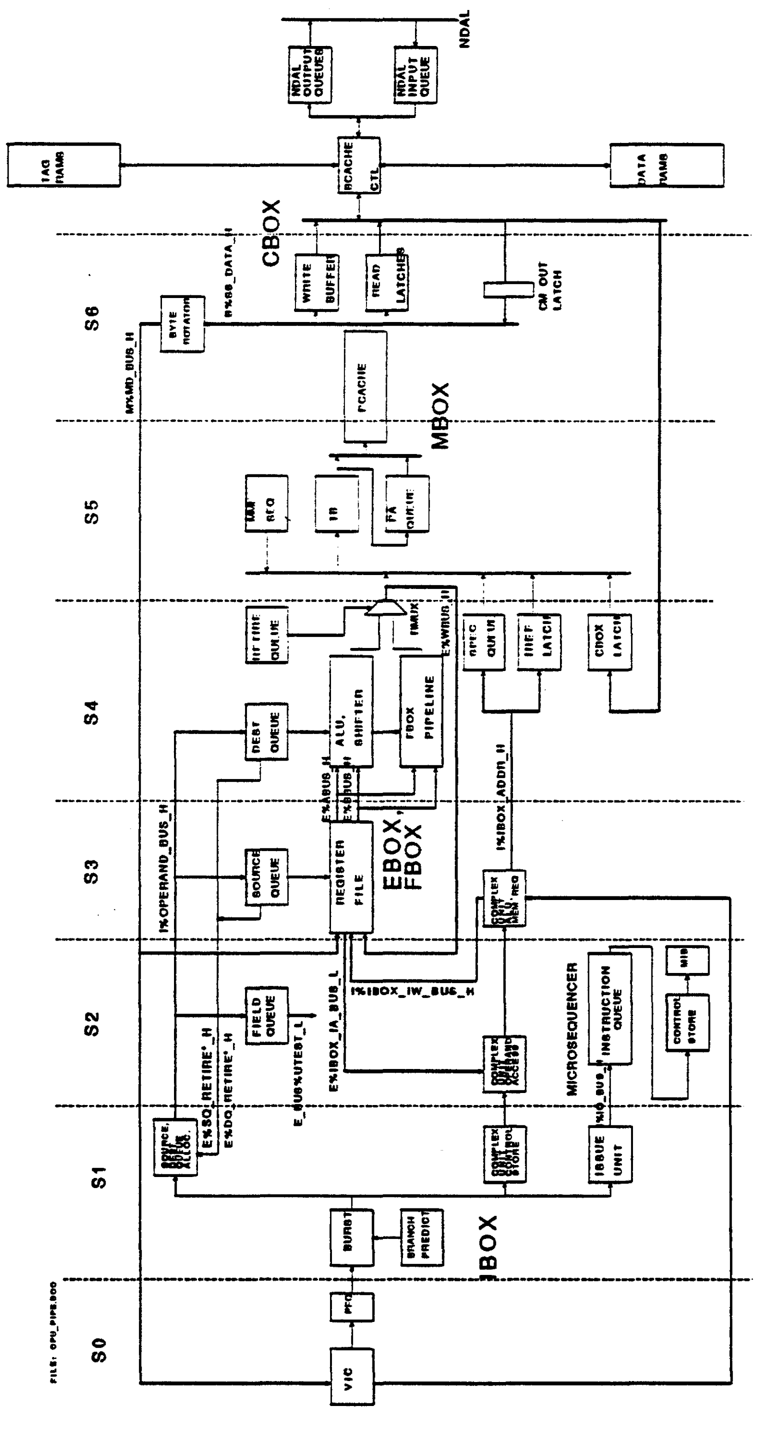
5.3 NVAX CPU Pipeline Overview

The remainder of this chapter discusses the NVAX CPU pipeline, which is shown as a block diagram in Figure 5–8. This is a high-level view of the CPU and abstracts many of the details. For a more detailed view of the pipeline, users are encouraged to refer to the individual box chapters in this specification.

The pipeline is divided into seven segments denoted as "S0" through "S6". In Figure 5–8, the components of each section of the CPU are shown in the segment of the pipeline in which they operate.

The NVAX CPU is fully pipelined and, as such, is most similar to the abstract example shown in Figure 5–3. In addition to the overall macroinstruction pipeline, in which multiple macroinstructions are processed in the various segments of the pipeline, most of the sections also micropipeline operations. That is, if more than one operation is required to process a macroinstruction, the multiple operations are also pipelined within a section.

Figure 5–8: NVAX CPU Pipeline



NVAX Pipe

5.3.1 Normal Macroinstruction Execution

Execution of macroinstructions in the NVAX pipeline is decomposed into many smaller steps which are the distributed responsibility of the various sections of the chip. Because the NVAX CPU implements a macroinstruction pipeline, each section is relatively autonomous, with queues inserted between the sections to normalize the processing rates of each section.

5.3.1.1 The Ibox

The Ibox is responsible for fetching instruction stream data for the next instruction, decomposing the data into opcode and specifiers, and evaluating the specifiers with the goal of prefetching operands to support Ebox execution of the instruction.

The Ibox is distributed across segments S0 through S3 of the pipeline, with most of the work being done in S1. In S0, instruction stream data is fetched from the virtual instruction cache (VIC) using the address contained in the virtual instruction buffer address register (VIBA). The data is written into the prefetch queue (PFQ) and VIBA is incremented to the next location.

In segment S1, the PFQ is read and the burst unit uses internal state and the contents of the IROM to select the next instruction stream component—either an opcode or specifier. This decoding processing is known as *bursting*. Some instruction components take multiple cycles to burst. For example, FD opcodes require two burst cycles: one for the FD byte, and one for the second opcode byte. Similarly, indexed specifiers require at least two burst cycles: one for the index byte, and one or more for the base specifier.

When an opcode is decoded, the information is passed to the issue unit, which consults the IROM for the initial Ebox control store address of the routine which will process the instruction. The issue unit sends the address and other instruction-related information to the instruction queue where it is held until the Ebox reaches the instruction.

When a specifier is decoded, the information is passed to the source and destination queue allocation logic and, potentially, to the complex specifier pipeline. The source and destination queue allocation logic allocates the appropriate number of entries for the specifier in the source and destination queues in the Ebox. These queues contain pointers to operands and results, and are discussed in more detail below.

If the specifier is not a short literal or register specifier, which are collectively known as *simple* specifiers, it is considered to be a *complex* specifier and is processed by the small microcode-controlled complex specifier unit (CSU), which is distributed in segments S1 (control store access), S2 (operand access, including register file read), and S3 (ALU operation, Mbox request, GPR write) of the pipeline. The CSU pipeline computes all specifier memory addresses, and makes the appropriate request to the Mbox for the specifier type. To avoid reading or writing a GPR which is interlocked by a pending Ebox reference, the CSU pipeline includes a register scoreboard which detects data dependencies. The CSU pipeline also provides additional help to the Ebox by supplying operand information that is not an explicit part of the instruction stream. For example, the PC is supplied as an implicit operand for instructions that require it (such as BSBB).

The branch prediction unit (BPU) watches each opcode that is decoded looking for conditional and unconditional branches. For unconditional branches, the BPU calculates the target PC and redirects PC and VIBA to the new path. For conditional branches, the BPU predicts whether the instruction will branch or not based on previous history. If the prediction indicates that the branch will be taken, PC and VIBA are redirected to the new path. The BPU writes the conditional branch prediction flag into the branch queue in the Ebox, to be used by the Ebox in the execution of the instruction. The BPU maintains enough state to restore the correct instruction PC if the prediction turns out to be incorrect.

5.3.1.2 The Microsequencer

The microsequencer operates in segment S2 of the pipeline and is responsible for supplying to the Ebox the next microinstruction to execute. If a macroinstruction requires the execution of more than one microinstruction, the microsequencer supplies each microinstruction in sequence based on directives included in the previous microinstruction.

At macroinstruction boundaries, the microsequencer removes the next entry from the instruction queue, which includes the initial microinstruction address for the macroinstruction. If the instruction queue is empty, the microsequencer supplies the address of a special no-op microinstruction.

The microsequencer is also responsible for evaluating all exception requests, and for providing a pipeline flush control signal to the Ebox. For certain exceptions and interrupts, the microsequencer injects the address of a special microinstruction handler that is used to respond to the event.

5.3.1.3 The Ebox

The Ebox is responsible for executing all of the non-floating point instructions, for delivery of operands to and receipt of results from the Fbox, and for handling non-instruction events such as interrupts and exceptions. The Ebox is distributed through segments S3 (operand access, including register file read), S4 (ALU and shifter operation, Rmux request), and S5 (Rmux completion, register write, completion of Mbox request) of the pipeline.

For the most part, instruction operands are prefetched by the Ibox, and addressed indirectly through the source queue. The source queue contains the operand itself for short literal specifiers, and a pointer to an entry in the register file for other operand types.

An entry in the field queue is made when a field-type specifier entry is made into the source queue. The field queue provides microbranch conditions that allow the Ebox microcode to determine if a field-type specifier addresses either a GPR or memory. A microbranch on a valid field queue entry retires the entry from the queue.

The register file is divided into four parts: the GPRs, memory data (MD) registers, working registers, and CPU state registers. For register-mode specifiers, the source queue points to the appropriate GPR in the register file. For other non-short literal specifier modes, the source queue points to an MD register. The MD register is either written directly by the Ibox, or by the Mbox as the result of a memory read generated by the Ibox.

The S3 segment of the Ebox pipeline is responsible for selecting the appropriate operands for the Ebox and Fbox execution of instructions. Operands are selected onto E%ABUS_H and E%BBUS_H for use in both the Ebox and Fbox. In most instances, these operands come from the register file, although there are other data path sources of non-instruction operands (such as the PSL).

Ebox computation is done by the ALU and the shifter in the S4 segment of the pipeline on operands supplied by the S3 segment. Control for these units is supplied by the microinstruction which was originally supplied to the S3 segment by the microsequencer, and then subsequently moved forward in the pipeline.

The S4 segment also contains the RMUX, whose responsibility is to select results from either the Ebox or Fbox and perform the appropriate register or memory operation. The RMUX inputs come from the ALU, shifter, and **F%FBOX_RESULT_H** at the end of the cycle. The RMUX actually spans the S4/S5 boundary such that its outputs are valid at the beginning of the S5 segment. The RMUX is controlled by the retire queue, which specifies the source (either Ebox or Fbox) of the result to be processed (or retired) next. Non-selected RMUX sources are delayed until the retire queue indicates that they should be processed.

As the source queue points to instruction operands, so the destination queue points to the destination for instruction results. If the result is to be stored in a GPR, the destination queue contains a pointer to the appropriate GPR. If the result is to be stored in memory, the destination queue indicates that a request is to be made to the Mbox, which contains the physical address of the result in the PA queue (which is described below). This information is supplied as a control input to the RMUX logic.

Once the RMUX selects the appropriate source of result information, it either requests Mbox service, or sends the result onto **E%WBUS_H** to be written back to the register file or to other data path registers in the S5 segment of the pipeline. The interface between the Ebox and Mbox for all memory requests is the EM_LATCH, which contains control information and may contain an address, data, or both, depending on the type of request. In addition to operands and results that are prefetched by the Ibox, the Ebox can also make explicit memory requests to the Mbox to read or write data.

5.3.1.4 The Fbox

The Fbox is responsible for executing all of the floating point instructions in the VAX base instruction group, as well as the longword-length integer multiply instructions.

For each instruction that the Fbox is to execute, it receives from the microsequencer the opcode and other instruction-related information. The Fbox receives operand data from the Ebox on **E%ABUS_H** and **E%BBUS_H**.

Execution of instructions is performed in a dedicated Fbox pipeline that appears in segment S4 of Figure 5–8, but is actually a minimum of three cycles in length. Certain instructions, such as integer multiply, may require multiple passes through some segments of the Fbox pipeline. Other instructions, such as divide, are not pipelined at all.

Fbox results and status are returned via **F%FBOX_RESULT_H** to the RMUX in the Ebox for retirement. When the instruction is next to retire, the RMUX hardware, as directed by the destination queue, sends the results to either the GPRs for register destinations, or to the Mbox for memory destinations.

5.3.1.5 The Mbox

The Mbox operates in the S5 and S6 segments of the pipeline, and is responsible for all memory references initiated by the other sections of the chip. Mbox requests can come from the Ibox (for VIC fills and for specifier references), the Ebox or Fbox via the RMUX and the EM_LATCH (for instruction result stores and for explicit Ebox memory requests), from the Mbox itself (for translation buffer fills and PTE reads), and from the Cbox (for invalidates and cache fills).

All virtual references are translated to a physical address by the translation buffer (TB), which operates in the S5 segment of the pipeline. For instruction result references generated by the Ibox, the translated address is stored in the physical address queue (PA queue). These addresses are later matched with data from the Ebox or Fbox, when the result is calculated.

For memory references, the physical address from either the TB or the PA queue is used to address the primary cache (Pcache) starting in the S5 segment of the pipeline and continuing into the S6 segment. Read data is available in the middle of the S6 segment, right-justified and returned to the requester on **M%MD_BUS_H** by the end of the cycle. Writes are also completed by the end of the cycle. Although the Pcache access spans the S5 and S6 segments of the pipeline, a new access can be started each cycle in the absence of a TB or cache miss.

5.3.1.6 The Cbox

The Cbox is responsible for maintaining and accessing the backup cache (Bcache), and for control of the off-chip bus (the NDAL). The Cbox receives input from the Mbox in the S6 segment of the pipeline, and usually takes multiple cycles to complete a request. For this reason, the Cbox is not shown in specific pipeline segments.

If a memory read misses in the Pcache, the request is sent to the Cbox for processing. The Cbox first looks for the data in the Bcache and fills the Pcache from the Bcache if the data is present. If the data is not present in the Bcache, the Cbox requests a cache fill on the NDAL from memory. When memory returns the data, it is written to both the Bcache and to the Pcache (and potentially to the VIC). Although Pcache fills are done by making a request to the Mbox pipeline, data is returned to the original requester as quickly as possible by driving data directly onto **B%S6_DATA_H**, and from there onto **M%MD_BUS_H** as soon as the bus is free.

Because the Pcache operates as a write-through cache, all memory writes are passed to the Cbox. To avoid multiple writes to the same Bcache block, the Cbox contains a write buffer in which multiple writes to the same quadwords are packed together before the Bcache is actually written. To maintain cache coherence with other system components, the Cbox acquires ownership of any data that is written to the cache.

5.3.2 Stalls in the Pipeline

Despite our best attempts at keeping the pipeline flowing smoothly, there are conditions which cause segments of the pipeline to stall. Conceptually, each segment of the pipeline can be considered as a black box which performs three steps every cycle:

1. The task appropriate to the pipeline segment is performed, using control and inputs from the previous pipeline segment. The segment then updates local state (within the segment), but not global state (outside of the segment).
2. Just before the end of the cycle, all segments send stall conditions to the appropriate state sequencer for that segment, which evaluates the conditions and determines which, if any, pipeline segments must stall.
3. If no stall conditions exist for a pipeline segment, the state sequencer allows it to pass results to the next segment and accept results from the previous segment. This is accomplished by updating global state.

This sequence of steps maximizes throughput by allowing each pipeline segment to assume that a stall will not occur (which should be the common case). If a stall does occur at the end of the cycle, global state updates are blocked, and the stalled segment repeats the same task (with potentially different inputs) in the next cycle (and the next, and the next) until the stall condition is removed.

This description is over-simplified in some cases because some global state must be updated by a segment before the stall condition is known. Also, some tasks must be performed by a segment once and only once. These are treated specially on a case-by-case basis in each segment.

Within a particular section of the chip, a stall in one pipeline segment also causes stalls in all upstream segments (those that occur earlier in the pipeline) of the pipeline. Unlike Rigel, stalls in one segment of the pipeline do not cause stalls in downstream segments of the pipeline. For example, a memory data stall in Rigel also caused a stall of the downstream ALU segment. In NVAX, a memory data stall does not stall the ALU segment (a no-op is inserted into the S4 segment when S4 advances to S5).

There are a number of stall conditions in the chip which result in a pipeline stall. Each is discussed briefly below and in much more detail in the appropriate chapter of this specification.

5.3.2.1 S0 Stalls

Stalls that occur in the S0 segment of the pipeline are as follows:

Ibox:

- PFQ full: In normal operation, the VIC is accessed using the address in VIBA, the data is sent to the prefetch queue, and VIBA is incremented. If the PFQ is full, the increment of VIBA is blocked, and the data is re-referenced in the VIC until there is room for it in the PFQ. At that point, prefetch resumes.

5.3.2.2 S1 Stalls

Stalls that occur in the S1 segment of the pipeline are as follows:

Ibox:

- Insufficient PFQ data: The burst unit attempts to decode the next instruction component each cycle. If there are insufficient PFQ bytes valid to decode the entire component, the burst unit stalls until the required bytes are delivered from the VIC.
- Source queue or destination queue full: During specifier decoding, the source and destination queue allocation logic must allocate enough entries in each queue to satisfy the requirements of the specifier being parsed. To guarantee that there will be sufficient resources available, there must be at least 2 free source queue entries and 2 free destination queue entries to complete the burst of the specifier. If there are insufficient free entries in either queue,the burst unit stalls until free entries become available.
- MD file full: When a complex specifier is decoded, the source queue allocation logic must allocate enough memory data registers in the register file to satisfy the requirements of the specifier being parsed. To guarantee that there will be sufficient resources available, there must be at least 2 free memory data registers available to complete the burst of the specifier. If there are insufficient free registers, the burst unit stalls until enough memory data registers becomes available.

- Second conditional branch decoded: The branch prediction unit predicts the path that each conditional branch will take and redirects the instruction stream based on that prediction. It retains sufficient state to restore the alternate path if the prediction was wrong. If a second conditional branch is decoded before the first is resolved by the Ebox, the branch prediction unit has nowhere to store the state, so the burst unit stalls until the Ebox resolves the actual direction of the first branch.
- Instruction queue full: When a new opcode is decoded by the burst unit, the issue unit attempts to add an entry for the instruction to the instruction queue. If there are no free entries in the instruction queue, the burst unit stalls until a free entry becomes available, which occurs when an instruction is retired through the RMUX.
- Complex specifier unit busy: If the burst unit decodes an instruction component that must be processed by the CSU pipeline, it makes a request for service by the CSU through an S1 request latch. If this latch is still valid from a previous request for service (either due to a multi-cycle flow or a CSU stall), the burst unit stalls until the valid bit in the request latch is cleared.
- Immediate data length not available: The length of the specifier extension for immediate specifiers is dependent on the data length of the specifier for that specific instruction. The data length information comes from one of the Ibox instruction PLAs which is accessed based on the opcode of the instruction. If the PLA access is not complete before an immediate specifier is decoded (which would have to be the first specifier of the instruction), the burst unit stalls for one cycle.

5.3.2.3 S2 Stalls

Stalls that occur in the S2 segment of the pipeline are as follows:

Ibox:

- Outstanding Ebox or Fbox GPR write: In order to calculate certain specifier memory addresses, the CSU must read the contents of a GPR from the register file. If there is a pending Ebox or Fbox write to the register, the Ibox GPR scoreboard prevents the GPR read by stalling the S2 segment of the CSU pipeline. The stall continues until the GPR write completes.
- Memory data not valid: For certain operations, the Ibox makes an Mbox request to return data which is used to complete the operation (e.g., the read done for the indirect address of a displacement deferred specifier). The Ibox MD register contains a valid bit which is cleared when a request is made, and set when data returns in response to the request. If the Ibox references the Ibox MD register when the valid bit is off, the S2 segment of the CSU pipeline stalls until the data is returned by the Mbox.

Microsequencer:

- Instruction queue empty: The final microinstruction of a macroinstruction execution flow in the Ebox is indicated when a SEQ.MUX/LAST.CYCLE* microinstruction is decoded by the microsequencer. In response to this event, the Ebox expects to receive the first microinstruction of the next macroinstruction flow based on the initial address in the instruction queue. If the instruction queue is empty, the Microsequencer supplies the instruction queue stall microinstruction in place of the next macroinstruction flow. In effect, this stalls the microsequencer for one cycle.

5.3.2.4 S3 Stalls

Stalls that occur in the S3 segment of the pipeline are as follows:

Ibox:

- Outstanding Ebox GPR read: In order to complete the processing for auto-increment, auto-decrement, and auto-increment deferred specifiers, the CSU must update the GPR with the new value. If there is a pending Ebox read to the register through the source queue, the Ibox scoreboard prevents the GPR write by stalling the S3 segment of the CSU pipeline. The stall continues until the Ebox reads the GPR.
- Specifier queue full: For most complex specifiers, the CSU makes a request for Mbox service for the memory request required by the specifier. If there are no free entries in the specifier queue, the S3 segment of the CSU pipeline stalls until a free entry becomes available.
- RLOG full: Auto-increment, auto-decrement, and auto-increment deferred specifiers require a free RLOG entry in which to log the change to the GPR. If there are no free RLOG entries when such a specifier is decoded, the S3 segment of the CSU pipeline stalls until a free entry becomes available.

Ebox:

- Memory read data not valid: In some instances, the Ebox may make an explicit read request to the Mbox to return data in one of the 6 Ebox working registers in the register file. When the request is made, the valid bit on the register is cleared. When the data is written to the register, the valid bit is set. If the Ebox references the working register when the valid bit is clear, the S3 segment of the Ebox pipeline stalls until the entry becomes valid.
- Field queue not valid: For each macroinstruction that includes a field-type specifier, the microcode microbranches on the first entry in the field queue to determine whether the field specifier addresses a GPR or memory. If the field queue is empty (indicating that the Ibox has not yet parsed the field specifier), the result of the next address calculation repeats the microbranch the next cycle. Although this is not a true stall, the effects are the same in that a microinstruction is repeated until the field queue becomes valid.
- Outstanding Fbox GPR write: Because the Fbox computation pipeline is multiple cycles long, the Ebox may start to process subsequent instructions before the Fbox completes the first. If the Fbox instruction result is destined for a GPR that is referenced by a subsequent Ebox microword, the S3 segment of the Ebox pipeline stalls until the Fbox GPR write occurs.
- Fbox instruction queue full: When an instruction is issued to the Fbox, an entry is added to the Fbox instruction queue. If there are no free entries in the queue, the S3 segment of the Ebox pipeline stalls until a free entry becomes available.

Ebox/Fbox:

- Source queue empty: Most instruction operands are prefetched by the Ibox, which writes a pointer to the operand value into the source queue. The Ebox then references up to two operands per cycle indirectly through the source queue for delivery to the Ebox or Fbox. If either of the source queue entries referenced is not valid, the S3 segment of the Ebox pipeline stalls until the entry becomes valid.

- Memory operand not valid: Memory operands are prefetched by the Ibox, and the data is written by the either the Mbox or Ibox into the memory data registers in the register file. If a referenced source queue entry points to a memory data register which is not valid, the S3 segment of the Ebox pipeline stalls until the entry becomes valid.

5.3.2.5 S4 Stalls

Stalls that occur in the S4 segment of the pipeline are as follows:

Ebox:

- Branch queue empty: When a conditional or unconditional branch is decoded by the Ibox, an entry is added to the branch queue. For conditional branch instructions, the entry indicates the Ibox prediction of the branch direction. The branch queue is referenced by the Ebox to verify that the branch displacement was valid, and to compare the actual branch direction with the prediction. If the branch queue entry has not yet been made by the Ibox, the S4 segment of the Ebox pipeline stalls until the entry is made.
- Fbox GPR operand scoreboard full: The Ebox implements a register scoreboard to prevent the Ebox from reading a GPR to which there is an outstanding write by the Fbox. For each Fbox instruction which will write a GPR result, the Ebox adds an entry to the Fbox GPR scoreboard. If the scoreboard is full when the Ebox attempts to add an entry, the S4 segment of the Ebox pipeline stalls until a free entry becomes available.

Fbox:

- Fbox operand not valid: Instructions are issued to the Fbox when the opcode is removed from the instruction queue by the microsequencer. Operands for the instruction may not arrive until some time later. If the Fbox attempts to start the instruction execution when the operands are not yet valid, the Fbox pipeline stalls until the operands become valid.

Ebox/Fbox:

- Destination queue empty: Destination specifiers for instructions are processed by the Ibox, which writes a pointer to the destination (either GPR or memory) into the destination queue. The destination queue is referenced in two cases: when the Ebox or Fbox store instruction results via the RMUX, and when the Ebox tries to add the destination of Fbox instructions to the Ebox GPR scoreboard. If the destination queue entry is not valid (as would be the case if the Ibox has not completed processing the destination specifier), a stall occurs until the entry becomes valid.
- PA queue empty: For memory destination specifiers, the Ibox sends the virtual address of the destination to the Mbox, which translates it and adds the physical address to the PA queue. If the destination queue indicates that an instruction result is in memory, a store request is made to the Mbox which supplies the data for the result. The Mbox matches the data with the first address in the PA queue and performs the write. If the PA queue is not valid when the Ebox or Fbox has a memory result ready, the RMUX stalls until the entry becomes valid. As a result, the source of the RMUX input (Ebox or Fbox) also stalls.
- EM_LATCH full: All implicit and explicit memory requests made by the Ebox or Fbox pass through the EM_LATCH to the Mbox. If the Mbox is still processing the previous request when a new request is made, the RMUX stalls until the previous request is completed. As a result, the source of the RMUX input (Ebox or Fbox) also stalls.

- RMUX selected to other source: Macroinstructions must be completed in the order in which they appear in the instruction stream. The Ebox retire queue determines whether the next instruction to complete comes from the Ebox or the Fbox. If the next instruction should come from one source and the other makes an RMUX request, the other source stalls until the retire queue indicates that the next instruction should come from that source.

5.3.3 Exception Handling

A pipeline exception occurs when a segment of the pipeline detects an event which requires that the normal flow of the pipeline be stopped in favor of another flow. There are two fundamental types of pipeline exceptions: those that resume the original pipeline flow once the exception is corrected, and those that require the intervention of the operating system. A TB miss on a memory reference is an example of the first type, and an access control violation is an example of the second type. M=0 faults are handled specially, as described below.

Restartable exceptions are handled entirely within the confines of the section that detected the event. Other exceptions must be reported to the Ebox for processing. Because the NVAX CPU is macropipelined, exceptions can be detected by sections of the pipeline long before the instruction which caused the exception is actually executed by the Ebox or Fbox. However, the reporting of the exception is deferred until the instruction is executed by the Ebox or Fbox. At that point, an Ebox handler is invoked to process the event.

Because the Ebox and Fbox are micropipelined, the point at which an exception handler is invoked must be carefully controlled. For example, three macroinstructions may be in execution in segments S3, S4, and S5 of the Ebox pipeline. If an exception is reported for the macroinstruction in the S3 segment, the two macroinstructions that are in the S4 and S5 segments must be allowed to complete before the exception handler is invoked.

To accomplish this, the S4/S5 boundary in the Ebox is defined to be the *commit point* for a microinstruction. Architectural state is not modified before the S5 segment of the pipeline, unless there is some mechanism for restoring the original state if an exception is detected (the Ibox RLOG is an example of such a mechanism). Exception reporting is deferred until the microinstruction to which the event belongs attempts to cross the S4/S5 boundary. At that point, the exception is reported and an exception handler is invoked. By deferring exception reporting to this point, the previous microinstruction (which may belong to the previous macroinstruction) is allowed to complete.

Most exceptions are reported by requesting a *microtrap* from the Microsequencer. When the Microsequencer receives a microtrap request, it causes the Ebox to break all its stalls, aborts the Ebox pipeline (by asserting **E_USQ%PE_ABORT_L**), and injects the address of a handler for the event into the control store address latch. This starts an Ebox microcode routine which will process the exception as appropriate. Certain other kinds of exceptions are reported by simply injecting the appropriate handler address into the control store at the appropriate point.

The VAX architecture categorizes exceptions into two types: faults and traps. For both types, the microcode handler for the exception causes the Ibox to back out all GPR modifications that are in the RLOG, and retrieves the PC from the PC queue. For faults, the PC returned is the PC of the opcode of the instruction which caused the exception. For traps, the PC returned is the PC of the opcode of the next instruction to execute. The microcode then constructs the appropriate exception frame on the stack, and dispatches to the operating system through the appropriate SCB vector.

There are a number of exceptions detected by the NVAX CPU pipeline, each of which is discussed briefly below, and in much more detail in the appropriate chapter of this specification.

5.3.3.1 Interrupts

The CPU services interrupt requests from various sources between macroinstructions, and at selected points within the string instructions. Interrupt requests are received by the interrupt section and compared with the current IPL in the PSL. If the interrupt request is for an IPL that is higher than the current value in the PSL, a request is posted to the microsequencer. At the next macroinstruction boundary, the microsequencer substitutes the address of the microcode interrupt service routine for the instruction execution flow.

The microcode handler then determines if there is actually an interrupt pending. If there is, it is dispatched to the operating system through the appropriate SCB vector.

5.3.3.2 Integer Arithmetic Exceptions

There are three integer arithmetic exceptions detected by the CPU, all of which are categorized as traps by the VAX architecture. This is significant because the event is not reported until after the commit point of the instruction, which allows that instruction to complete.

Integer Overflow Trap

An integer overflow is detected by the RMUX at the end of the S4 segment of the Ebox pipeline. If PSL<IV> is set and overflow traps are enabled by the microcode, the event is reported in segment S5 of the pipeline via a microtrap request.

Integer Divide-By-Zero Trap

An integer divide-by-zero is detected by the Ebox microcode routine for the instruction. It is reported by explicitly retiring the instruction and then jumping directly to the microcode handler for the event.

Subscript Range Trap

A subscript range trap is detected by the Ebox microcode routine for the INDEX instruction. It is reported by explicitly retiring the instruction and then jumping directly to the microcode handler for the event.

5.3.3.3 Floating Point Arithmetic Exceptions

All floating point arithmetic exceptions are detected by the Fbox pipeline during the execution of the instruction. The event is reported by the RMUX when it selects the Fbox as the source of the next instruction to process. At that point, a microtrap is requested.

5.3.3.4 Memory Management Exceptions

Memory management exceptions are detected by the Mbox when it processes a virtual read or write. This section covers actual memory management exceptions such as access control violation, translation not valid, and M=0 faults. Translation buffer misses are discussed separately in the next section. Because the reporting of memory management exceptions is specific to the operation that caused the exception, each case is discussed separately.

- **I-Stream Faults**

  While the Ibox is decoding instructions, it may access a page which is not accessible due to a memory management exception. This may occur on the opcode, a specifier or specifier extension, or on a branch displacement. Should this occur, the Ibox sets a global MME fault flag and stops. Memory management exceptions detected on intermediate operations during specifier evaluation (such as a read for the indirect address of a displacement deferred specifier) are converted by the Ibox into source or destination faults, as described below.

  If the Ebox reaches the instruction which caused the exception (which may not happen due to, for example, interrupt, exception, or branch), it will reference one of the queues, which does not have a valid entry because the Ibox stopped when the error was detected. The particular queue depends on the instruction component on which the error was detected. If the Ibox global MME flag is set when an empty queue entry is referenced, the error is reported in one of four ways.

  If the Ibox global MME flag is set when the microsequencer references an invalid instruction queue entry, it inserts the instruction queue stall into the pipeline and the Ebox qualifies it with the fault flag. When this flag reaches the S4 segment of the pipeline and is selected by the RMUX, a microtrap is requested.

  If the Ibox global MME flag is set when the Ebox references an invalid source queue entry, a fault flag is injected into either the Ebox or Fbox pipelines, depending on the type of instruction. To avoid a deadlock, S3 stalls do not prevent forward prgress of the flag in the pipeline. When the flag reaches the S4 segment of the pipeline and is selected by the RMUX, a microtrap is requested.

  If the Ibox global MME flag is set when the Ebox microcode microbranches on an invalid field queue entry, a fault flag is injected into the Ebox pipeline. When the flag reaches the S4 segment of the pipeline and is selected by the RMUX, a microtrap is requested.

  If the Ibox global MME flag is set when the Ebox references an invalid branch queue entry, and the RMUX selects the Ebox, a microtrap is requested.

  If the Ibox global MME flag is set when the RMUX references an invalid destination queue entry for a store request, a microtrap is requested.

- **Source Operand Faults**

  If the Mbox detects a memory management exception during the translation for a source specifier, it qualifies the data returned to the MD file with a fault flag which is written into the MD file. When this entry is referenced by the Ebox, a fault flag is injected into the pipeline. To avoid a deadlock, S3 stalls do not prevent forward prgress of the flag in the pipeline. When the flag reaches the S4 segment of the pipeline and is selected by the RMUX, a microtrap is requested.

- **Destination Address Faults**

  If the Mbox detects a memory management exception during the translation for a destination specifier, it sets a fault flag in the PA queue entry for the address. When this entry is referenced by the RMUX, a microtrap is requested,.

- **Faults on Explicit Ebox Memory Requests**

  Explicit Ebox reads and writes are, by definition, performed in the context of the instruction which the Ebox is currently executing. If the Mbox detects a memory management exception that was the result of an explicit Ebox read or write, it requests an immediate microtrap to the memory management fault handler.

- **M=0 faults**

  M=0 faults occur when the Mbox finds the M-bit clear in the PTE which is used to translate write-type references. The event is reported to the Ebox in one of the three ways described above: via the MD file or PA queue fault flags, or via an immediate microtrap for explicit Ebox writes.

  Unlike other memory management exceptions, which are dispatched to the operating system, M=0 faults are completely processed by the Ebox microcode handler. For normal instructions, the handler causes the Ibox to back out all GPR modifications that are in the RLOG and retrieves the PC from the PC queue. For string instructions, any RLOG entries that belong to the string instructions are not processed, and PSL<FPD> is set. Using the PTE address supplied by the Mbox, the Ebox microcode reads the PTE, sets the M-bit, and writes the PTE back to memory. The instruction stream is then restarted at the interrupted instruction (which may result in special FPD handling, as described below).

5.3.3.5 Translation Buffer Miss

Translation buffer misses are handled by the Mbox transparently to the rest of the CPU. When a reference misses in the translation buffer, the Mbox aborts the current reference and invokes the services of the memory management exception sequencer in the Mbox, which fetches the appropriate PTE from memory and loads it into the translation buffer. The original reference is then restarted.

5.3.3.6 Reserved Addressing Mode Faults

Reserved addressing mode faults are detected by the Ibox for certain illegal combinations of specifier addressing modes and registers. When one of these combinations is detected, the Ibox sets a global addressing mode fault flag that indicates that the condition was detected and stops.

If the Ibox global addressing mode fault flag is set when the Ebox references an invalid source queue entry, a fault flag is injected into either the Ebox or Fbox pipelines, depending on the type of instruction. To avoid a deadlock, S3 stalls do not prevent forward prgress of the flag in the pipeline. The fault flag is carried along the Ebox or Fbox pipeline and passed to the RMUX, which reports the event by requesting a microtrap when that source is selected.

If the Ibox global addressing mode fault flag is set when the Ebox microcode microbranches on an invalid field queue entry, a fault flag is injected into the Ebox pipeline. When the flag reaches the S4 segment of the pipeline and is selected by the RMUX, a microtrap is requested.

Similarly, if the Ibox global addressing mode fault flag is set when the RMUX, in response to a request by the Ebox or Fbox, references an invalid destination queue entry, a microtrap is requested.

5.3.3.7 Reserved Operand Faults

Reserved operand faults for floating point operands are detected by the Fbox, and reported in the same manner as the floating point arithmetic exceptions described above.

Other reserved operand faults are detected by Ebox microcode as part of macroinstruction execution flows and are reported by jumping directly to the fault handler.

5.3.3.8 Exceptions Occurring as the Consequence of an Instruction

Opcode-specific exceptions such as reserved instruction faults, breakpoint faults, etc., are dispatched directly to handlers by placing the address of the handler in the instruction PLA for each instruction.

Other instruction-related faults, such as privileged instruction faults, are detected in execution flows by the Ebox microcode and are reported by jumping directly to the fault handler.

For testability, the Fbox may be disabled. If this is the case, integer multiply instructions are executed by the Ebox microcode and floating point instructions are converted into reserved instruction faults for emulation by software. When the first Ebox microinstruction of an Fbox operand flow for a floating point macroinstruction reaches the S4 segment of the pipeline, a microtrap is requested. The handler for this microtrap then jumps directly to the reserved instruction fault handler.

5.3.3.9 Trace Fault

Trace faults are detected by the microsequencer with some help from the Ebox. The microsequencer maintains a duplicate copy of PSL<TP>, which it updates as required to track the state of the PSL copy as it would exist when the instruction is executed by the Ebox. At the end of a macroinstruction, the microsequencer logically ORs its local copy of the TP bit with PSL<TP>. If either is set, the microsequencer substitutes the address of the microcode trace fault handler for the address of the next macroinstruction.

5.3.3.10 Conditional Branch Mispredict

When the Ibox decodes a conditional branch, it predicts the path that the branch will take and places its prediction into the branch queue. When the Ebox reaches the instruction, it evaluates the actual path that the branch took and compares it in the S5 segment of the Ebox pipeline with the Ibox prediction. If the two are different, the Ibox is notified that the branch was mispredicted and a microtrap request is made to abort the Ebox and Fbox pipelines. The Ibox flushes itself, backs out any GPR modifications that are in the RLOG, and redirects the instruction stream to the alternate path. The Ebox microcode handler for this event cleans up certain machine state and waits for the first instruction from the alternate path.

5.3.3.11 First Part Done Handling

During the execution of one of the 8 string instructions that are implemented by the CPU, an exception or an interrupt may be detected. In that event, the Ebox microcode saves all state necessary to resume the instruction in the GPRs, backs up PC to point to the opcode of the string instruction, sets PSL<FPD> in the saved PSL, and dispatches to the handler for the interrupt or exception.

When the interrupt or exception is resolved, the software handler terminates with an REI back to the instruction. When the Ibox decodes an instruction with PSL<FPD> set, it stops parsing the instruction immediately after the opcode. In particular, it does not parse the specifiers. When the microsequencer finds PSL<FPD> set at a macroinstruction boundary, it substitutes the address of a special FPD handler for the instruction execution flow.

The FPD handler determines which instruction is being resumed from the opcode, unpacks the state saved in the GPRs, clears PSL<FPD>, advances PC to the end of the string instruction (by adding the opcode PC to the length of the instruction, which was part of the saved state), and jumps back to the middle of the interrupted instruction.

5.3.3.12 Cache and Memory Hardware Errors

Cache and memory hardware errors are detected by the Mbox or Cbox, depending on the type of error. If the error is recoverable (e.g., a Pcache tag parity error on a write simply disables the Pcache), it is reported via a soft error interrupt request and is dispatched to the operating system.

In some instances, write errors that are not recoverable by hardware are reported via a hard error interrupt request, which results in the invocation of the operating system.

Read errors that are not recoverable by hardware are reported via the assertion of a soft error interrupt, and also in a manner that is similar to that used for memory management exceptions, as described above. In fact, the MD file, PA queue, and the Ibox all contain a hardware error flag in parallel with the memory management fault flag. With the exception of TB parity errors, which cause an immediate microtrap request, the event is reported to the Ebox in exactly the same way as the equivalent memory management exception would be, but the microcode exception handler is different. For example, an unrecoverable error on a specifier read would set the hardware error flag in the MD file. When the flag is referenced, the error flag is injected into the pipeline. When the flag advances to the S4 segment and is selected by the RMUX, it causes a microtrap request which invokes a hardware error handler rather than a memory management handler.

Note that certain other errors are reported in the same way. For example, if the memory management sequencer in the Mbox receives an unrecoverable error trying to read a PTE necessary to translate a destination specifier, it sets the hardware error flag in the PA queue for the entry corresponding to the specifier. This results in a microtrap to the hardware error handler when the entry is referenced. PTE read errors for read references are also reported via the original reference.

5.4 Revision History

Table 5–1: Revision History

Who	When	Description of change
Mike Uhler	06-Mar-1989	Release for external review.
Mike Uhler	19-Dec-1989	Update for second-pass release.
Mike Uhler	02-Feb-1991	Update after pass 1 PG.

Chapter 6

Microinstruction Formats

6.1 Ebox Microcode

The NVAX microword consists of 61 bits divided into two major sections. Bits <60:15> control the Ebox Data Path and are encoded into two formats. Bits <14:0> control the Microsequencer and are also encoded into two formats.

6.1.1 Data Path Control

The Data Path Control Microword specifies all the information needed to control the Ebox Data Path. The two formats, Standard and Special, are selected by bit <60>, the FORMAT bit. In addition, bit <45>, the LIT bit, selects the constant generation format of the microword, which may be either an 8-bit constant or a 10-bit constant, depending on a decode in the MISC field. Pictures of the microword formats are in Figure 6–1 and Figure 6–2. A brief description of each field is given in Table 6–1 and Table 6–2.

Figure 6–1: Ebox Data Path Control, Standard Format

```
 6|5 5 5 5|5 5 5 5|5 5 4 4|4 4 4 4|4 4 4 4|3 3 3 3|3 3 3 3|3 3 2 2|2 2 2 2|2 2 2 2|1 1 1 1|1
 0|9 8 7 6|5 4 3 2|1 0 9 8|7 6 5 4|3 2 1 0|9 8 7 6|5 4 3 2|1 0 9 8|7 6 5 4|3 2 1 0|9 8 7 6|5
+-+---------+---------+-+-----+-+---------+---------+-+-+-+---------+---------+-------------+
|0|   ALU   |   MRQ   |Q| SHF |0|   VAL   |    B    |L|W|V|   DST   |    A    |    MISC     |
+-+---------+---------+-+-----+-+---------+---------+-+-+-+---------+---------+-------------+
                              |1| POS |    CONST    | MISC not equal CONST.10
                              +-+-----+-------------+
                              |1|     CONST.10      | MISC equal CONST.10
                              +-+-------------------+
```

Table 6–1: EBOX Data Path Control Microword Fields, Standard Format

Bit Position	Microword Field	Microword Format	Description
60	FORMAT	—	Microword format–Standard or Special
59:55	ALU	Both	ALU function select

Table 6–1 (Cont.): EBOX Data Path Control Microword Fields, Standard Format

Bit Position	Microword Field	Microword Format	Description
54:50	MRQ	Both	Mbox request select
49	Q	Standard	Q register load control
48:46	SHF	Standard	Shifter function select
45	LIT	Both	ALU/shifter B port control–register or literal
44:40	VAL	Standard[1]	Constant shift amount
39:35	B	Both[1]	ALU/shifter B port select
44:43	POS	Both[2]	Constant position
42:35	CONST	Both[2]	8-bit constant value
44:35	CONST.10	Both[3]	10-bit constant value
34	L	Both	Length control
33	W	Both	Wbus driver control
32	V	Both	VA write enable
31:26	DST	Both	WBUS destination select
25:20	A	Both	ALU/shifter A port select
19:15	MISC	Both	Miscellaneious function select, group 0

[1] NOT Constant generation microword variant

[2] 8-Bit Constant generation microword variant, when MISC field not equal CONST.10

[3] 10-Bit Constant generation microword variant, when MISC field equal CONST.10

Figure 6–2: Ebox Data Path Control, Special Format

```
 6|5 5 5 5|5 5 5 5|5 5 4 4|4 4 4 4|4 4 4 4|3 3 3 3|3 3 3 3|3 3 2 2|2 2 2 2|2 2 2 2|1 1 1 1|1
 0|9 8 7 6|5 4 3 2|1 0 9 8|7 6 5 4|3 2 1 0|9 8 7 6|5 4 3 2|1 0 9 8|7 6 5 4|3 2 1 0|9 8 7 6|5
+-+-------+-------+-------+-+-----+-+---------------+-+-+-+-----------+-----------+---------+
|1|  ALU  |  MRQ  | MISC1 |0| MISC2|D|       B       |L|W|V|    DST    |     A     |  MISC   |
+-+-------+-------+-------+-+-----+-+---------------+-+-+-+-----------+-----------+---------+
                          |1|POS|     CONST         | MISC not equal CONST.10
                          +-+---+-------------------+
                          |1|      CONST.10         | MISC equal CONST.10
                          +-+-----------------------+
```

Table 6–2: EBOX Data Path Control Microword Fields, Special Format

Bit Position	Microword Field	Microword Format	Description
60	FORMAT	—	Microword format–Standard or Special

Table 6–2 (Cont.): EBOX Data Path Control Microword Fields, Special Format

Bit Position	Microword Field	Microword Format	Description
59:55	ALU	Both	ALU function select
54:50	MRQ	Both	Mbox request select
49:46	MISC1	Special	Miscellaneous function select, group 1
45	LIT	Both	ALU/shifter B port control–register or literal
44:41	MISC2	Special[1]	Miscellaneous function select, group 2
40	DISABLE.RETIRE	Special[1]	Instruction retire disable
39:35	B	Both[1]	ALU/shifter B port select
44:43	POS	Both[2]	Constant position
42:35	CONST	Both[2]	8-bit constant value
44:35	CONST.10	Both[3]	10-bit constant value
34	L	Both	Length control
33	W	Both	Wbus driver control
32	V	Both	VA write enable
31:26	DST	Both	WBUS destination select
25:20	A	Both	ALU/shifter A port select
19:15	MISC	Both	Miscellaneious function select, group 0

[1] NOT Constant generation microword variant

[2] 8-Bit Constant generation microword variant, when MISC field not equal CONST.10

[3] 10-Bit Constant generation microword variant, when MISC field equal CONST.10

6.1.2 Microsequencer Control

The Microsequencer Control Microword supplies the information necessary for the Microsequencer to calculate the address of the next microinstruction. The basic computation done by the Microsequencer involves selecting a base address from one of several sources, and then optionally modifying three bits of the base address to get the final next address.

Bit <14>, SEQ.FMT, selects between Jump and Branch formats. Figure 6–3 and Figure 6–4 show the two formats. Table 6–3 and Table 6–4 describe each of the fields.

Figure 6–3: Ebox Microsequencer Control, Jump Format

```
 1 1 1|1 1     |       |
 4 3 2|1 0 9 8|7 6 5 4|3 2 1 0
+-+-+---+---------------------+
|0|S|MUX|          J          |
+-+-+---+---------------------+
```

Table 6–3: Ebox Microsequencer Control Microword Fields, Jump Format

Bit Position	Microword Field	Microword Format	Description
14	SEQ.FMT	—	Microsequencer format–Jump or Branch
13	SEQ.CALL	Both	Subroutine call
12:11	SEQ.MUX	Jump	Next address select
10:0	J	Jump	Next address

Figure 6–4: Ebox Microsequencer Control, Branch Format

```
 1 1 1|1 1     |       |
 4 3 2|1 0 9 8|7 6 5 4|3 2 1 0
+-+-+---------+---------------+
|1|S|SEQ.COND |    BR.OFF     |
+-+-+---------+---------------+
```

Table 6–4: Ebox Microsequencer Control Microword Fields, Branch Format

Bit Position	Microword Field	Microword Format	Description
14	SEQ.FMT	—	Microsequencer format–Jump or Branch
13	SEQ.CALL	Both	Subroutine call
12:8	SEQ.COND	Branch	Microbranch condition select
7:0	BR.OFF	Branch	Page offset of next address

6.2 Ibox CSU Microcode

The Ibox complex specifier unit is controlled by a 29-bit microword, as shown in Figure 6–5. A brief description of each field is given in Table 6–5.

Figure 6–5: Ibox CSU Format

```
 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|  ALU      |DL|    A      |     B     |    DST    |   MISC    |      MREQ     |MUX    |          NXT              |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Table 6–5: Ibox CSU Microword Fields

Bit Position	Microword Field	Description
28:26	ALU	ALU function select
25	DL	Data length control
24:22	A	ALU A port select
21:19	B	ALU B port select
18:16	DST	Wbus destination
15:13	MISC	Miscellaneous function select
12:9	MREQ	Mbox request select
8:7	MUX_CNT	Next address mux select
6:0	NXT	Next address

6.3 Ibox Instruction ROM and Control PLAs

The Ibox instruction decode is controlled by several ROMs and PLAs that are generated from a single source file whose format is shown in Figure 6–6. A brief description of each field is given in Table 6–6. A more detailed description of the control information as it is actually found in the hardware is given in Table 7–12.

Figure 6–6: Ibox Instruction ROM Format

```
 64 63 62 61 60 59 58 57 56 55  54  53 52 51 50 49 48 47 46   45
+--+--+--+--+--+--+--+--+--+--+----+--+--+--+--+--+--+--+--+-----+
|        EXEC_DISP           |VS|ST_SPCQ|DS|B | V|FB| SP_CNT |A_CNT|
+--+--+--+--+--+--+--+--+--+--+----+--+--+--+--+--+--+--+--+-----+

 44 43 42 41 40 39 38 37 36 35 34 33 32
+--+--+--+--+--+--+--+--+--+--+--+--+--+
| A1_REG    |A1_DL| A1_AT  |  ASSIST1  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+

 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|E_DL |  AT 6   | DL 6|  AT 5   | DL 5|  AT 4   | DL 4|  AT 3   | DL 3|  AT 2   | DL 2|  AT 1   | DL 1|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Table 6–6: Ibox Instruction ROM Fields

Bit Position	Microword Field	Description
64:56	EXEC_DISP	Bits <9:1> of the instruction entry point address in the Ebox control store
55	VS	Determines whether a Vfield specifier occupies 1 or 2 source queue entries
54:53	ST_SPCQ	Determines whether the parser is stopped at the end of the instruction, when the next PC queue entry is made, and when the parser is restarted
52	DS	Specifies the length (byte or word) of a branch displacement for the instruction
51	B	Specifies whether the instruction has a branch displacement
50	V	Not currently used
49	FB	Specifies whether this instruction is implemented in the Fbox
48:46	SP_CNT	Specifies the number of real specifiers for the instruction
45	A_CNT	Specifies whether the instruction has an assist
44:41	A1_REG	Specifies the register to use for instructions with an assist
40:39	A1_DL	Specifies the data length to use for instructions with an assist
38:36	A1_AT	Specifies the access type to use for instructions with an assist
35:32	ASSIST1	Specifies the type of assist for instructions with an assist
31:30	E_DL	Specifies the initial Ebox data length to be used for the instruction
29:27	AT6	Supplies the encoded access type of the sixth specifier, if any
26:25	DL6	Supplies the encoded data length of the sixth specifier, if any
24:22	AT5	Supplies the encoded access type of the fifth specifier, if any
21:20	DL5	Supplies the encoded data length of the fifth specifier, if any
19:17	AT4	Supplies the encoded access type of the fourth specifier, if any
16:15	DL4	Supplies the encoded data length of the fourth specifier, if any

Table 6–6 (Cont.): Ibox Instruction ROM Fields

Bit Position	Microword Field	Description
14:12	AT3	Supplies the encoded access type of the third specifier, if any
11:10	DL3	Supplies the encoded data length of the third specifier, if any
9:7	AT2	Supplies the encoded access type of the second specifier, if any
6:5	DL2	Supplies the encoded data length of the second specifier, if any
4:2	AT1	Supplies the encoded access type of the first specifier, if any
1:0	DL1	Supplies the encoded data length of the first specifier, if any

6.4 Revision History

Table 6–7: Revision History

Who	When	Description of change
Debra Bernstein	06-Mar-1989	Release for external review.
Mike Uhler	13-Dec-1989	Update for second-pass release.
Mike Uhler	04-Feb-1991	Update after pass 1 PG.

Chapter 7

The Ibox

7.1 Overview

7.1.1 Introduction

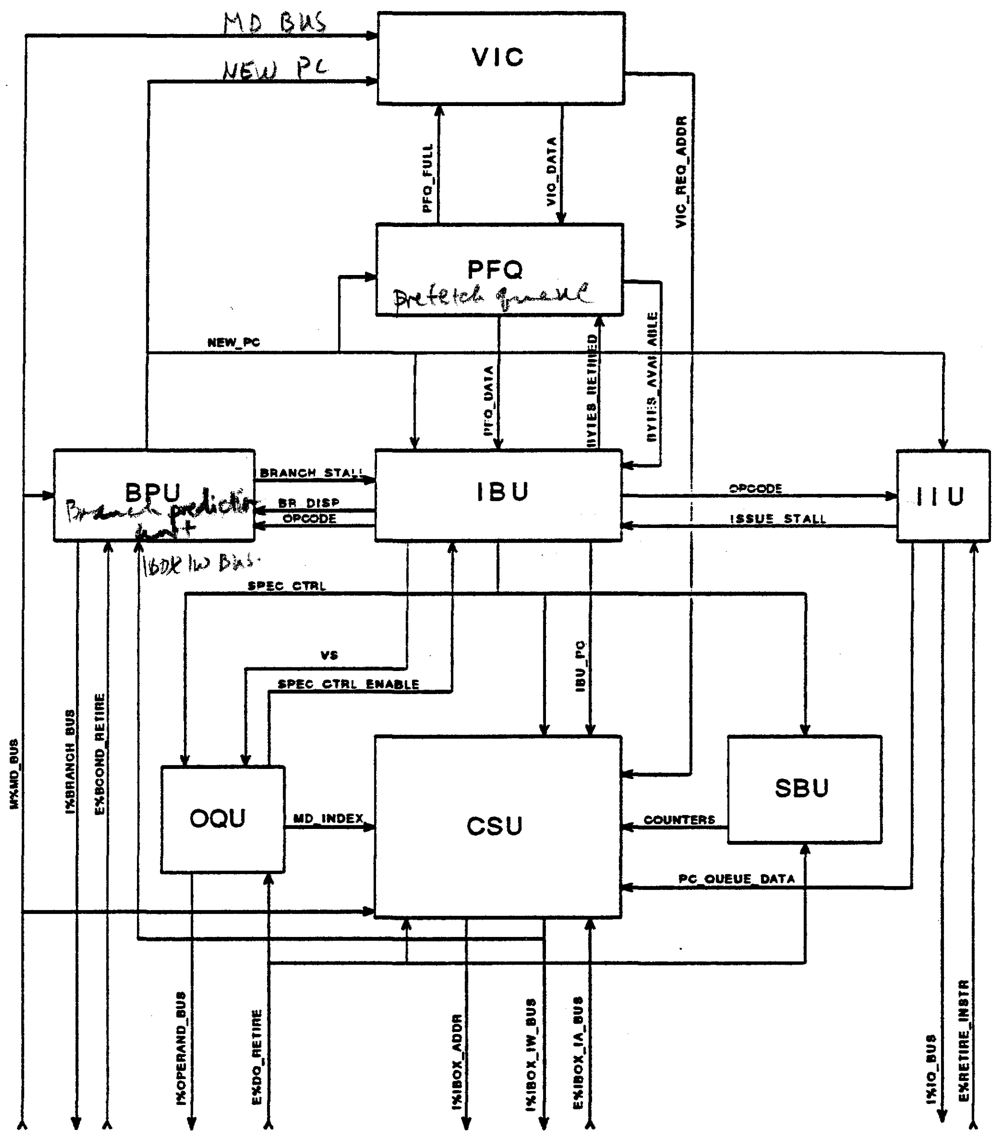
This chapter describes the Ibox section of the NVAX CPU chip. The 4-stage Ibox pipeline (S0..S3) runs semi-autonomously to the rest of the NVAX CPU and supports the following functions:

- **Instruction Stream Prefetching**
  The Ibox attempts to maintain sufficient instruction stream data to decode the next instruction or operand specifier.
- **Instruction Parsing**
  The Ibox identifies the instruction opcodes and operand specifiers, and extracts the information necessary for further processing.
- **Operand Specifier Processing**
  The Ibox processes the operand specifiers, initiates the required memory references, and provides the Ebox with the information necessary to access the instruction's operands.
- **Branch Prediction**
  Upon identification of a branch opcode, the Ibox hardware predicts the direction of the branch (taken vs. not taken). For branch taken predictions, the Ibox redirects the instruction prefetching and parsing logic to the branch destination, where instruction processing resumes.

Figure 7–1 is a top level block diagram of the Ibox showing the major Ibox sub-sections and their inter-connections.

This chapter presents a high-level description of the Ibox functions, then provides details of the Ibox sub-sections which support each function.

Figure 7–1: Ibox Block Diagram



7.1.2 Functional Overview

The Ibox fetches, parses, and processes the instruction stream, attempting to maintain a constant supply of parsed VAX instructions available to the Ebox for execution. The pipelined nature of the NVAX CPU allows for multiple macroinstructions to reside within the CPU at various stages of execution. The Ibox, running semi-autonomously to the Ebox, parses the macroinstructions following the instruction that is currently in Ebox execution. Performance gains are realized when the time required for instruction parsing in the Ibox is hidden during the Ebox execution of an earlier instruction. The Ibox places the information generated while parsing ahead into Ebox queues.

The Instruction Queue contains instruction specific information which includes the instruction opcode, a floating point instruction flag, and an entry point for the Ebox microcode.

The Source Queue contains information about the source operands for the instructions in the instruction queue. Source queue entries contain either the actual operand (as in a short literal), or a pointer to the location of the operand.

The Destination Queue contains information required for the Ebox to select the location for execution results storage. The two possible locations are the VAX General Purpose Registers (GPRs) and memory.

These queues allow the Ibox to work in parallel with the Ebox. As the Ebox consumes the entries in the queues, the Ibox parses ahead adding more. In the ideal case, the Ibox would stay far enough ahead of the Ebox such that the Ebox would never have to stall because of an empty queue.

The Ibox needs access to memory for instruction and operand data. Instruction and operand data requests are made through a common port to the Mbox. All data for both the Ibox and the Ebox is returned on a shared **M%MD_BUS_H**<63:0>

The Ibox port feeds operand data requests to the Mbox Specifier Request Latch and instruction data requests to the Mbox Instruction Request Latch. These 2 latches allow the Ibox to issue memory requests for both instruction and operand data even though the Mbox may be processing other requests.

The Ibox supports 4 main functions:

1. Instruction Stream Prefetching
2. Instruction Parsing
3. Operand Specifier Processing
4. Branch Prediction

Instruction Stream Prefetching works to provides a steady source of instruction stream data for instruction parsing. While the instruction parsing logic works on one instruction, the instruction prefetching logic fetches several instructions ahead.

The Instruction Parsing logic parses the incoming instruction stream, identifying and initial processing each of the instruction's components. The instruction opcodes and associated information are passed directly into the Ebox instruction queue. Operand specifier information is passed on to the operand specifier processing logic.

The Operand Specifier Processing logic locates the operands in registers, in memory, or in the Instruction Stream. This logic places operand information in the Ebox source and destination queues, and makes the required operand memory requests.

The Ibox does not have prior knowledge of branch direction for branches which rely on Ebox condition codes. The Branch prediction logic makes a prediction on which way the branch will go and forces the Ibox to take that path. This logic saves the alternate branch path target, so that in the event that Ebox branch execution shows that the prediction was wrong, the Ibox can be redirected to the correct branch direction.

7.1.3 The Pipeline

The Ibox logic spans the first 4 segments of the NVAX CPU pipeline (S0..S3). The following table lists the major Ibox sub-sections and which pipe segments they occupy.

Table 7–1: Ibox Pipeline

Sub-Section Name	Description
	S0 Pipe Stage
VIC	The Virtual Instruction Cache is a 2KB direct mapped Istream-only cache with 32 byte blocks, a valid bit per quadword, and an access size of 8 bytes.
PFQ	The Prefetch Queue is a queue of instruction stream data supplied by the VIC. It is 4 bytes wide by 4 elements deep.
	S1 Pipe Stage
IBU	The Instruction Burst Unit breaks up the incoming instruction data into opcodes, operand specifiers, specifier extensions, and branch displacements and passes the results to other parts of the Ibox for further processing.
IIU	The Instruction Issue Unit takes the opcodes provided by the IBU and generates an Ebox microcode dispatch addresses and other context for instruction execution.
BPU	The Branch Prediction Unit predicts whether or not branches will be taken and redirects the Ibox instruction processing as necessary.
OQU	The Operand Queue Unit is the interface to the Ebox source and destination queues.
SBU	The Scoreboard Unit tracks outstanding read and write references to the GPRs.
CSU (S1)	This segment of the Complex Specifier Unit contains the microsequencer and control store.
	S2 Pipe Stage
CSU (S2)	This is the register READ segment of the complex specifier unit. It accesses the necessary registers and provides the data to the ALU in the next pipe stage.
	S3 Pipe Stage
CSU (S3)	This is the ALU and WRITE segment of the complex specifier unit. This segment performs the necessary ALU operations and writes the results either to the Ebox register file or to local temporary registers. This segment also contains the Mbox interface.

Pipe segment S0 is dedicated to supplying a steady stream of instruction data for use by the IBU. When prefetching is enabled, the VIC attempts to fill the PFQ with up to 8 bytes of instruction stream data.

The IBU parses in S1, the Ebox receives information about the instruction and its operands in the instruction, source, and destination queues. The IIU is the Ibox interface to the Ebox instruction queue, and the OQU is the interface to the source and destination queues. When the IBU has identified a new opcode, this opcode is passed to the IIU which places the necessary opcode-specific information in the Ebox instruction queue. When operand specifiers are identified, the OQU places the necessary operand specific information in the source and destination queues.

The CSU is a 3 stage (S1..S3) microcoded pipeline dedicated to handling operand specifiers which require complex processing and/or access to memory. It has read and write access to the Ebox register file and a port to the Mbox. Memory requests from the VIC are received at the CSU and forwarded to the Mbox when there is a cycle free of specifier memory requests.

7.2 Instruction Stream Prefetching

The Instruction Stream Prefetching mechanism provides a buffer of Istream data 4 bytes wide and 4 elements deep for use by the instruction parser. This buffer insulates the instruction parser from the bursty behavior of the cache and memory sub-systems, and allows for the parallel operation of the instruction fetching and instruction parsing functions.

The two Ibox sub-sections which support the instruction prefetching function are the Virtual Instruction Cache (VIC) and the Prefetch Queue (PFQ) both of which reside in the S0 pipe stage.

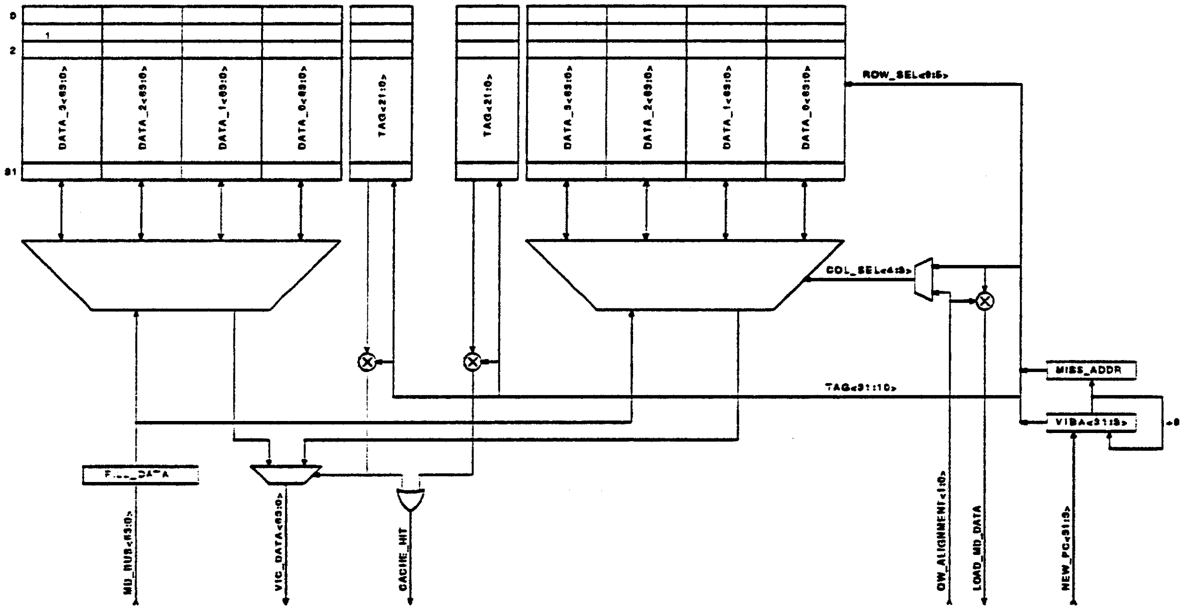
7.2.1 The VIC

The VIC is a 2KB, direct-mapped, Istream cache which acts as the primary source of instruction stream data for the Ibox. The VIC attributes are summarized in Table 7–2.

Table 7–2: VIC Attributes

Cache size	2K Bytes
Access Type	Direct Mapped
Block Size	32 Bytes
Sub-block Size	8 Bytes
Valid Bits	4 Valid bits/Cache Block = 1 Per Sub-block
Data Parity Bits	4 Even Parity bits/Cache Block = 1 Per Sub-block
# of Tags	64 Tags
Tag Parity Bit	1 Even Parity Bit Per Tag
Fill Algorithm	Fill Forward
Access Size	8 Bytes
Bus Size	8 Bytes
Prefetching	NONE
Data stored	Istream Only
Virtual/Physical	Virtual

Figure 7–2: VIC Block Diagram



The VIC is a virtual cache because the addresses that are used to index into the cache are untranslated VAX Virtual addresses. See Section 12.5 for more on VAX Memory Management and Address Translation. The VIC maintains a local prefetch pointer called VIBA<31:3> (Virtual Instruction Buffer Address). This address is quadword aligned and always points to the next quadword of Istream data to be sent to the PFQ. Table 7–3 shows the fields in VIBA<>.

Table 7–3: VIBA bit fields

Bit field	Field name	Description
<4:3>	SUBBLK_INDEX	Sub-block index (or column select) bits indicate which sub-block to select from cache block.
<10:5>	ROW_INDEX	Row select bits determine which cache row to access
<31:11>	VIBA_TAG	Bits to be compared against cache tag

Whenever the BPU issues a new PC, the VIC latches the NEW_PC<31:3> in VIBA<31:3>. VIBA<10:5> are used to select which cache row to access. Each cache row, shown in Figure 7–3, stores a 21-bit tag with even parity for the tag, and four quadword sub-blocks each with a valid bit and an even parity bit which covers the data only. When a cache row of the VIC is accessed, The 21-bit tag is compared with VIBA<31:11> to determine cache hit or miss. VIBA<4:3> selects the cache sub-block.

Figure 7–3: VIC Cache Row Format

```
   2             6                      6                      6                      6
   0       0     3                0     3                0     3                0     3                0
--+---------+-+-+------------------+-+-+------------------+-+-+------------------+-+-+------------------+
|P|   TAG   |V|P| Sub-block 3 data |V|P| Sub-block 2 data |V|P| Sub-block 1 data |V|P| Sub-block 0 data |
--+---------+-+-+------------------+-+-+------------------+-+-+------------------+-+-+------------------+
\                                                                                                       /
 --------------------------------------------- 287 bits -----------------------------------------------
```

Whenever space exists in the PFQ, the VIC attempts to supply the next quadword of instruction stream data by doing a VIC_READ using the current value of VIBA<31:3>. If the VIC_READ results in a miss, the VIC begins a VIC_FILL sequence by sending a request through the CSU for a cache fill operation from the Mbox.

7.2.1.1 VIC Control

The VIC control evaluates the status flags summarized in Table 7–4 every cycle to determine the proper type of cache sequence for the next cycle. **VIC_ENABLE** enables the cache itself, specifically **VIC_READ**s and **VIC_WRITE**s. **PREFETCH_ENABLE** is the enable bit for the Istream prefetch sequencer. **VIC_ERROR** indicates that there was a VIC parity error. **MBOX_ERROR** indicates that error status was reported by the Mbox. **WRITE_PENDING** indicates that the Mbox drove valid Istream data on the **M%MD_BUS_H**<63:0> last cycle, and a cache write cycle should begin next. The **MISS_PENDING** flag is set when a **VIC_READ** misses in cache, and remains set until the cache fill sequence terminates. **LOAD_VIC_DATA** indicates that VIC data is ready for the PFQ. **LOAD_MD_DATA** indicates that the data on the **M%MD_BUS_H**<63:0> during a VIC fill should be loaded into the PFQ.

Table 7–4: VIC Status Flags

VIC Flag	Meaning
VIC_ENABLE	The VIC enable bit
PREFETCH_ENABLE	The prefetch enable bit
VIC_ERROR	There was a parity error in the VIC
MBOX_ERROR	There was an error in the Mbox fetching Istream data
WRITE_PENDING	Valid data latched from M%MD_BUS_H<63:0>, ready to be written to the VIC
MISS_PENDING	A VIC cache fill from the Mbox is in progress
VIC_READ	A cache read from the VIC is in progress

7.2.1.2 VIC_Reads

The VIC starts a **VIC_READ** sequence when **PREFETCH_ENABLE** is set and **WRITE_PENDING** is clear. If **VIC_ENABLE** is set, the **VIC_READ** sequence accesses the cache using the address in **VIBA**<31:3>. The decode of **VIBA**<10:5> selects one of 64 cache rows. If **TAG**<20:0> matches **VIBA**<31:11> and the valid (V) bit for the sub-block selected by **VIBA**<4:3> is set, then there is a cache hit. The data from the sub-block selected by **VIBA**<4:3> is driven onto **VIC_DATA_BUS**<63:0>, **LOAD_VIC_DATA** is asserted if the PFQ is not full, and the data is loaded into the PFQ.

If **VIBA**<31:11> does not match **TAG**<20:0>, or the tag matches but the V bit for the selected sub-block is not set, then a cache miss has occurred. In this case, **VIBA**<31:3> is saved in **MISS_ADDRESS**<31:3> and the **MISS_PENDING** flag is set. The four data parity bits for the accessed cache block are latched in **MISS_PARITY**<3:0>. The four valid bits for the same cache block are latched in **MISS_VALID**<3:0> if the cache miss is caused by a clear sub-block valid bit. If the cache miss is caused by a tag miscompare then **MISS_VALID**<3:0> is cleared. **VIC_WRITE**s make use of **MISS_ADDRESS**<31:3>, **MISS_PARITY**<3:0>, and **MISS_VALID**<3:0>. A cache fill operation begins as described in Section 7.2.1.3.

If **VIC_ENABLE** is clear or the **LOCK** bit in the **ICSR** register is set, indicating a VIC parity error has occurred, then all **VIC_READ**s are forced to miss.

7.2.1.3 VIC Fills

Upon detection of a cache miss during a **VIC_READ**, the VIC issues a fill request to the CSU. The miss address, stored in **MISS_ADDRESS**<31:3>, is driven onto **VIC_REQ_ADDR**<31:3> and **VIC_REQ** is asserted. The CSU forwards the **VIC_REQ** to the Mbox during the next free cycle on the **I%IBOX_ADDR_H**<> bus and associated control lines.

The Mbox returns quadwords of instruction data starting with the requested quadword and continuing to the end of the block. This cache fill algorithm is called fill forward. If the Mbox goes off-chip to get the requested data, then a full cache block of instruction data is returned, but not necessarily in any particular order. If the Mbox processes the fill request and finds that the request resides in I/O space, the request is also sent off-chip. In this case only the single requested quadword of data returns to the VIC. In all cases, the VIC is unaware of the number of data blocks being returned. When the last block of data is being returned by either the Cbox or Mbox, a **M%LAST_FILL_H** is signaled allowing **MISS_PENDING** to be cleared and a new read begun.

7.2.1.4 VIC Writes

The assertion of **M%VIC_DATA_L** indicates the presence of Istream data on **M%MD_BUS_H**<63:0>. The VIC latches **M%MD_BUS_H**<63:0> in **FILL_DATA**<63:0>, **M%MD_BUS_QW_PARITY_L**<0> in **FILL_DATA_PARITY**<0>, **M%QW_ALIGNMENT_H**<1:0> in **MISS_ADDRESS**<4:3>, and sets **WRITE_PENDING**.

If **VIC_ENABLE** is set, then a **VIC_WRITE** commences the next cycle using the address stored in **MISS_ADDRESS**<31:3> and the data stored in **FILL_DATA**<63:0>. **MISS_ADDRESS**<10:5> selects the cache row to write and **MISS_ADDRESS**<4:3> selects the sub-block to write. **TAG**<20:0> and its parity bit for the selected row are written with **MISS_ADDRESS**<31:11> and the even parity calculated for these bits. The selected sub-block is written with **FILL_DATA**<63:0>. **MISS_PARITY**<3:0> and **MISS_VALID**<3:0> contain the four data parity bits and four valid bits for the cache block being filled. The parity bit in **MISS_PARITY**<3:0> indexed by **MISS_ADDRESS**<4:3> is associated with the sub-block being written. This parity bit is written with **M%MD_BUS_QW_PARITY_L**<0>. The valid bit in **MISS_VALID**<3:0> indexed by **MISS_ADDRESS**<4:3> is associated with the sub-block being written. This valid bit is set. Both **MISS_PARITY**<3:0> and **MISS_VALID**<3:0> are written into the cache array.

There may be up to four **VIC_WRITE**s for each **VIC_FILL** depending upon sub-block alignment and fill sequence. However, the cache block tag and tag parity, all four data parity bits, all four data valid bits, and one sub-block of data are all written with every **VIC_WRITE**.

If **VIC_ENABLE** is clear, **VIC_WRITE**s are disabled, but the cache fill sequence completes normally.

See section Section 7.2.1.7 for information on **M%HARD_ERR_H** and **M%MME_FAULT_H**.

7.2.1.5 VIC Bypass

When fill data arrives at the VIC on the **M%MD_BUS_H**<63:0>, an evaluation is done to determine if the incoming data should be loaded directly into the PFQ. If so, then the PFQ latches the data directly from the **M%MD_BUS_H**<63:0> and **VIBA** is incremented by 8. This action is referred to as a VIC bypass and is signaled to the PFQ by **LOAD_MD_DATA**. Note that a **VIC_WRITE** occurs regardless of the outcome of the evaluation and whether or not the VIC bypass is enabled. If **PFQ_FULL** from the PFQ is asserted, indicating the PFQ is full, then **LOAD_MD_DATA** is not asserted and **VIBA** is not incremented.

The evaluation consists of checking to make sure that the incoming data is for the same cache block and sub-block to which **VIBA** points. The only time **VIBA** can be pointing to a different block than the block for which data is returning, is if a previous VIC bypass or Hit-Under-Miss incremented **VIBA** across a cache block boundary. This circumstance is indicated by a **VIBA_NEW_BLOCK** flag.

In order to facilitate VIC bypass, the Mbox returns **M%QW_ALIGNMENT_H**<1:0> with each piece of fill data. These two bits represent the quadword index for this data within the hexaword cache block. If **VIBA_NEW_BLOCK** is clear and **M%QW_ALIGNMENT_H**<1:0> match **VIBA**<4:3> then the incoming data can be loaded into the PFQ. When **VIBA_NEW_BLOCK** is set, indicating that the data the PFQ is waiting for is not in the block being filled by the Mbox, then VIC bypass is blocked and **LOAD_MD_DATA** is not asserted.

7.2.1.6 VIC Hits Under Miss

If the last **VIC_WRITE** was also a VIC bypass condition, then **VIBA** increments and potentially points to valid data in the current or next cache block. A subsequent **VIC_READ** is permitted even when **MISS_PENDING** is still set. This is referred to as a VIC Hit-Under-Miss. If the **VIC_READ** during **MISS_PENDING** also misses, no cache fill request is started. **MISS_ADDRESS**, **MISS_PARITY**<3:0>, and **MISS_VALID**<3:0> are not updated on a second miss. Note that **VIC_READS** may start and stop during a fill sequence based on **VIC_WRITE**s, but they always restart at the termination of a fill sequence when **M%LAST_FILL_H** is signaled.

7.2.1.7 VIC Exceptions and Errors

The VIC interprets the Mbox exception and error signals during the **VIC_WRITE** sequence. The **M%MME_FAULT_H** signal indicates that the Mbox encountered a memory management exception during the processing of an instruction stream reference. The Mbox produces the **M%HARD_ERR_H** signal when a hardware error is detected during the processing of an instruction stream reference. When **M%VIC_DATA_L** indicates the presence of data from the Mbox on the **M%MD_BUS_H**<63:0>, the assertion of either **M%MME_FAULT_H** or **M%HARD_ERR_H** blocks the setting of the **WRITE_PENDING** flag. **M%MME_FAULT_H** and **M%HARD_ERR_H** set the error flags **IMMGT_EXC** and **MHARD_ERR**, respectively. These flags are sent directly to the IBU. They are also used to disable prefetching and block VIC bypass until they are cleared either by a **E%STOP_IBOX_H** from the Ebox or a **LOAD_NEW_PC** from the BPU. They are also cleared by **E%IBOX_LOAD_PC_L** which indicates an impending **LOAD_NEW_PC**.

The VIC checks tag and data during **VIC_READ**s. Parity is calculated for the data sub-blocks selected by **VIBA**<4:3>. The even parity value for the quadword of data is then compared to the parity (P) bit associated with the sub-block read from cache. Data parity miscompares are reported as parity errors only on valid data. The even parity value for **VIC_TAG**<20:0> is calculated on **VIC_READ**s and compared to the parity (P) bit from the array that is associated with the tag read. Tag parity miscompares are always reported as parity errors. When the VIC detects either parity error, it clears **PREFETCH_ENABLE**, disabling VIC prefetching, and sets the **LOCK** bit in the **ICSR** register, preventing further cache reads and writes. The VIC asserts **IHARD_ERR** to forward the error condition to the IBU. **IHARD_ERR** remains asserted until it is cleared by a **E%STOP_IBOX_H**. The error status bits are set appropriately in the **ICSR** IPR register and the address of the error is latched in the **VMAR** register, as explained in Section 7.2.1.16. In addition, the VIC requests a system soft error interrupt by asserting the **I%IBOX_S_ERR_L**.

VIC tag and data parity checking are done specifically to protect the data in the VIC arrays.

Refer to section Section 7.9.2 for details on the IBU handling of Istream errors.

7.2.1.8 PC Load Effects

The assertion of **LOAD_NEW_PC** by the BPU has the following effects:

1. **PREFETCH_ENABLE** is set.
2. **VIBA** is loaded.
   VIBA<31:3> is loaded from the global Ibox bus **NEW_PC**<31:3>
3. **MHARD_ERR** is cleared.
4. **IMMGT_EXC** is cleared.
5. **MISS_PENDING** is cleared.

6. **WRITE_PENDING** is cleared.
7. **VIC_READ** is set.
8. **I%FLUSH_IREF_LAT_H** is asserted by the BPU to the Mbox.

The VIC reacts to any **LOAD_NEW_PC** from the BPU on a cycle by cycle basis as follows:

Cycle N :

- The Ibox may make an Istream request this cycle.
- Fill data returning from the Mbox to the Ibox is ignored.

Cycle N+1 :

- **LOAD_NEW_PC** is asserted to redirect instruction flow.
- **I%FLUSH_IREF_LAT_H** is asserted to clear outstanding Istream references.
- The Ibox may make an Istream request this cycle which is ignored by the Mbox.
- Fill data returning from the Mbox to the Ibox is ignored. This is the last cycle in which fill data for the Istream being flushed can be sent.
- Prefetching is enabled if previously disabled.
- **MISS_PENDING** is cleared and **VIC_READ** is set.
- New VIC hit or miss is determined.

Cycle N+2 :

- The Ibox may make a new Istream request based on whether the VIC hit or missed.
- **MISS_PENDING** may be set and **VIC_READ** cleared if a VIC miss was determined.
- The Mbox may not send Istream data for the old Istream request to the Ibox.

Section 7.6 and Section 7.5.1.7 explain more about PC loads.

7.2.1.9 E%STOP_IBOX_H Effects

The assertion of **E%STOP_IBOX_H** by the Ebox has the following effects:

1. **PREFETCH_ENABLE** is cleared.
2. **MHARD_ERR** is cleared.
3. **IMMGT_EXC** is cleared.
4. **IHARD_ERR** is cleared.
5. **MISS_PENDING** is cleared.
6. **WRITE_PENDING** is cleared.
7. **VIC_READ** is cleared.
8. **I%FLUSH_IREF_LAT_H** is asserted by the BPU.

The VIC reacts to a **E%STOP_IBOX_H** on a cycle by cycle basis as follows:

Cycle N :

- **E%STOP_IBOX_H** is asserted.
- The Ibox may make an Istream request this cycle.

- Fill data returning from the Mbox to the Ibox is ignored.

Cycle N+1 :

- I%FLUSH_IREF_LAT_H is asserted to clear outstanding Istream references.
- The Ibox will not make an Istream request this cycle.
- Fill data returning from the Mbox to the Ibox is ignored. This is the last cycle in which fill data for the Istream being flushed can be sent.
- Prefetching is disabled.
- MISS_PENDING and VIC_READ are cleared, VIC is put into an idle state, waiting for an E%IBOX_LOAD_PC_L from the Ebox.

7.2.1.10 Prefetch Stop Conditions

PREFETCH_ENABLE is cleared in the following cases:

1. **Any VIC, Mbox error, or Mbox exception**
   when a VIC error is detected or Mbox error is reported.
2. **E%STOP_IBOX_H signaled by Ebox**
   when the Ebox microcode performs a MISC/RESET_CPU which asserts E%STOP_IBOX_H.
3. **STOP_VIC_PREFETCH, STOP_PARSER bit from the IROM**
   stops Ibox prefetching for those instructions expected to redirect the instruction flow or access the IPRs.

7.2.1.11 Prefetch Start Conditions

PREFETCH_ENABLE is set in the following cases:

1. **PC load**
   on all PC loads.
2. **E%RESTART_IBOX_H signaled by Ebox**
   when the Ebox microcode performs a E%RESTART_IBOX_H, unless there is an outstanding VIC or Mbox error, or a PC load by the Ebox is pending, as signaled by E%IBOX_LOAD_PC_L.

7.2.1.12 Prioritized List of Prefetch Start/stop Conditions

The following priority is followed when multiple prefetch start/stop conditions occur simultaneously:

1. E%STOP_IBOX_H - stops prefetching
2. **PC Load** - starts prefetching
3. E%IBOX_LOAD_PC_L - stops prefetching (a PC load is pending)
4. **Any VIC or Mbox Error or Exception** - stops prefetching
5. E%RESTART_IBOX_H - starts prefetching
6. STOP_VIC_PREFETCH - stops prefetching

7.2.1.13 VIC Enable

The VIC powers up with **VIC_ENABLE** clear. **VIC_ENABLE** can be set and cleared during normal operation through the IPR register described in Section 7.2.1.16. **VIC_ENABLE** is cleared by hardware when any VIC parity error is detected.

MACROCODE RESTRICTION

> In functional operation, an REI must precede the MTPR which enables the VIC in order to flush all of the valid bits. However, if all the valid bits are guaranteed to have been written with a known value (such as in diagnostics or in macrocode that initializes the entire VIC), then this REI may be omitted.

7.2.1.14 VIC Flushing

The Ebox asserts **E%FLUSH_VIC_H** under microcode control to flush the VIC (clear all data valid bits). VIC flushes occur in such instances as the REI instruction, machine checks, and certain exceptions and interrupts.

MICROCODE RESTRICTION

> The Ebox microcode guarantees that prefetching is disabled whenever **E%FLUSH_VIC_H** is asserted, either implicitly in the context of an instruction with a **STOP_PARSER** assist or by performing an explicit **E%STOP_IBOX_H**.

The VIC reacts to a **E%FLUSH_VIC_H** on a cycle by cycle basis as follows:

Cycle N :

- Prefetching has already been disabled.
- **E%FLUSH_VIC_H** is asserted.
- The Ibox may make an Istream request this cycle.
- Fill data returning from the Mbox to the Ibox is ignored.

Cycle N+1 :

- **I%FLUSH_IREF_LAT_H** is asserted to clear outstanding Istream references.
- The Ibox will not make an Istream request this cycle.
- Fill data returning from the Mbox to the Ibox is ignored. This is the last cycle in which fill data for the Istream being flushed can be sent.

7.2.1.15 Flushing IREFs

The signal **I%FLUSH_IREF_LAT_H** is asserted by the BPU whenever a new PC is loaded indicating a redirection of the Istream. It is also asserted whenever there is a **E%STOP_IBOX_H** or a **E%FLUSH_VIC_H** from the Ebox. In all cases, the Mbox may continue to return VIC fill data in the same cycle as the **I%FLUSH_IREF_LAT_H**, but not the following cycle. The VIC will ignore any fill data received in the same cycle or the one cycle previous to the cycle in which **I%FLUSH_IREF_LAT_H** is signaled.

7.2.1.16 VIC Control and Error Registers

The VIC contains 4 internal processor registers (IPRs) which provide VIC control and read/write access to the arrays.

MACROCODE RESTRICTION

VIC_ENABLE must be cleared before writing to the VIC IPRs: VMAR, VDATA, or VTAG. VIC_ENABLE must be cleared before reading from VIC IPRs: VDATA, VTAG. In functional operation, an REI must preceed the MTPR which enables the VIC.

See Section 7.4.2.8 for details of the IPR mechanism.

7F0
7F4

Figure 7–4: IPR D0 (hex), VMAR

C0

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10  9  8| 7  6  5  4| 3  2  1  0
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                               ADDR                            |                 |     |  | 0| 0| :VMAR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                                           ^           ^     ^
                                                                           |           |     |
                                                                           |           |     |
                                                             ROW_INDEX ---+           |     |
                                                                      SUB_BLOCK ---+     |
                                                                                  LW ---+
```

Table 7–5: VMAR Field Descriptions

Name	Extent	Type	Description
LW	2	WO	Longword select bit. Selects longword of sub-block for cache access
SUB_BLOCK	4:3	RW	Sub-block select. Selects data sub-block for cache access, also latches VIBA<4:3> on VIC parity errors
ROW_INDEX	10:5	RW	Row select. Row index for read and write access to cache array, also latches VIBA<10:5> on VIC parity errors
ADDR	31:11	RO	Error address field. Latches tag portion of VIBA on VIC parity errors

When the VIC is disabled, the VIC Memory Address Register (VMAR) may be used as an index for direct IPR access to the cache arrays. VMAR<10:5> supply the cache row index, VMAR<4:3> supply the cache sub-block, and VMAR<2> indicates the longword within a quadword address.

VMAR also latches and holds the VIBA<31:3> on VIC array parity errors.

Figure 7–5: IPR D1 (hex), VTAG

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10  9  8| 7  6  5  4| 3  2  1  0
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                             TAG                              | 1| 1|TP|    DP     |     V     | :VTAG
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Table 7–6: VTAG Field Descriptions

Name	Extent	Type	Description
V	3:0	RW	Data valid bits. Supply data valid bits on array read/writes
DP	7:4	RW	Data parity bits. Supply data parity on array read/writes
TP	8	RW	Tag parity bit. Supplies tag parity on tag array read/writes
TAG	31:11	RW	Tag. Supplies tag on tag array read/writes

The VTAG IPR provides read and write access to the cache tag array. An IPR write to VTAG will write the contents of the M%MD_BUS_H<63:0> to the tag, parity, and valid bits for the row indexed by VMAR<10:5>. VTAG<31:11> are written to the cache tag. VTAG<8> is written to the associated tag parity bit. VTAG<7:4> are used to write the four data parity bits associated with the indexed cache row. Similarly VTAG<3:0> write the four data valid bits associated with the cache row. DP<3:0> and V<3:0> are the data parity and data valid bits, respectively, for the 4 quadwords of data in the same row. DP<0> and V<0> correspond to the quadword of data addressed when address bits 4:3 = 00, DP<1> and V<1> correspond to the quadword of data addressed when address bits 4:3 = 01, etc.

Figure 7–6: IPR D2 (hex), VDATA

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10  9  8| 7  6  5  4| 3  2  1  0
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                            DATA                                               | :VDATA
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Table 7–7: VDATA Field Descriptions

Name	Extent	Type	Description
DATA	31:0	RW	Data for data array reads and writes

The VDATA IPR provides read and write access to the cache data array. When VDATA is written, the cache data array entry indexed by VMAR is written with the IPR data. Since the IPR data is a longword, two accesses to VDATA are required to read or write a quadword cache sub-block.

Writes to VDATA with VMAR<2> = 0 simply accumulate the IPR data destined for the low longword of a sub-block in FILL_DATA<31:0>. A subsequent write to VDATA with VMAR<2> = 1 directs the the IPR data to FILL_DATA<63:32>, and triggers a cache write sequence to the sub-block indexed by VMAR.

Reads to VDATA with VMAR<2> = 0 trigger a cache read sequence to the sub-block indexed by VMAR<>. The low longword of the a sub-block is returned as IPR read data. A read of VDATA with VMAR<2> = 1 returns the high longword of the sub-block as IPR data.

Figure 7–7: IPR D3 (hex), ICSR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10  9  8| 7  6  5  4| 3  2  1  0
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                       0                                        |  |  |  | 0|  | :ICSR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                                                   ^  ^  ^     ^
                                                                                   |  |  |     |
                                                                                   |  |  |     |
                                                                                   |  |  |     |
                                                                          TPERR ---+  |  |     |
                                                                             DPERR ---+  |     |
                                                                                 LOCK ---+     |
                                                                                     ENABLE ---+
```

Table 7–8: ICSR Field Descriptions

Name	Extent	Type	Description
ENABLE	0	RW,0	Enable Bit. When set, allows cache access to the VIC. Initializes to 0 on RESET.
LOCK	2	WC	Lock Bit. When set, validates and prevents further modification of the error status bits in the ICSR and the error address in the VMAR register. When clear, indicates no VIC parity error has been recorded and allows ICSR and VMAR to be updated.
DPERR	3	RO	Data Error Bit. When set, indicates data parity error occurred in data array if the Lock Bit is also set.
TPERR	4	RO	Tag Error Bit. When set, indicates tag parity error occurred in tag array if the Lock Bit is also set.

The ICSR IPR provides control and status functions for the Ibox. VIC tag and data parity errors are latched in the read-only ICSR<4:3>, respectively. ICSR<2> is set when a tag or data parity error occurs and keeps the error status bits and the VMAR register from being modified further. Writing a logic one to ICSR<2> clears the LOCK bit and allows the error status to be updated. When ICSR<2> is clear, the values in ICSR<4:3> are meaningless. When ICSR<2> is set, a VIC parity error has occurred, and either ICSR<4> or ICSR<3> will be set indicating that the parity error was either a tag parity error or a data parity error, respectively. ICSR<4:3> cannot be cleared from software. ICSR<0> provides IPR control of the VIC enable. It is cleared on RESET.

7.2.1.17 VIC Performance Monitoring Hardware

Hardware exists in the Ibox VIC to support the NVAX Performance Monitoring Facility. See Chapter 18 for a global description of this facility.

The VIC hardware generates two signals I%PMUX0_H and I%PMUX1_H which are driven to the central performance monitoring hardware residing in the Ebox. These two signals are used to supply VIC hit rate data to the performance monitoring counters.

I%PMUX0_H is asserted the cycle when a VIC read reference is first attempted while the prefetch queue is not full. **I%PMUX1_H** signals the hit status for this event in the same cycle.

The data is captured only on the first read reference that could be used by the PFQ to avoid skewed hit ratios caused by multiple hits or misses to the same reference while the prefetch queue is full or the VIC is waiting for a cache fill.

7.2.2 The Prefetch Queue

The PFQ is a 4-longword-deep queue for Istream data. When prefetching is enabled, the VIC controls the supply of data to the PFQ. The PFQ can accept one quadword of data each cycle. When the PFQ contains insufficient available space to load another quadword of data it asserts **PFQ_FULL** which prevents the VIC from loading additional data into the PFQ. When the PFQ contains no unused Istream data it asserts **PFQ_EMPTY** and sends it to the IBU.

The PFQ loads data from the **M%MD_BUS_H<63:0>** or **VIC_DATA_BUS** as directed by the load signals **LOAD_MD_DATA** and **LOAD_VIC_DATA** from the VIC. **LOAD_MD_DATA** is asserted by the VIC only when there are no errors associated with the data. Data loaded from the **VIC_DATA_BUS** must be conditioned with the error signal **IHARD_ERR**. If **LOAD_VIC_DATA** and **IHARD_ERR** are both asserted, corrupted data is loaded into the PFQ from the **VIC_DATA_BUS**. To prevent this data from being used, the IBU reports the error immediately and stops parsing data.

The PFQ determines the number of valid unused bytes of Istream data available for parsing and sends this information to the IBU on **AVAIL_LE***. When the IBU retires Istream data it signals the PFQ on **I_IBU%RETIRE_SPECB_H<5:0>** and **I_IBU%RETIRE_OPCODE** the number of Istream bytes retired. These two signals are used to update the pointers in the PFQ.

The output of the PFQ is directed through a MUX which aligns the data for use by the IBU. The alignment MUX takes the first and second longwords and the first byte from the third longword as inputs. The alignment MUX outputs 6 contiguous bytes starting from any byte in the first longword, based on the PFQ pointers.

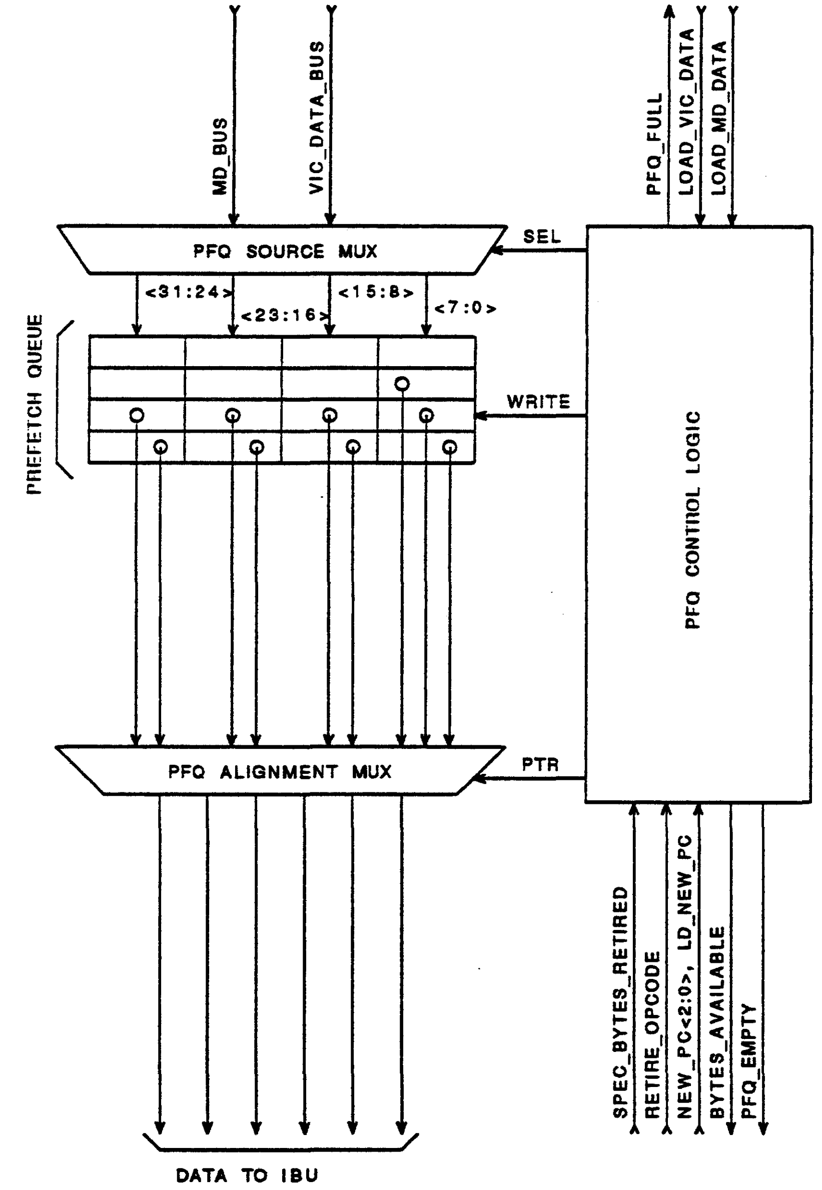
7.2.2.1 PC load effects

The PFQ is flushed when the BPU broadcasts a new PC load as indicated by **I_BPU%LOAD_NEW_PC** and when the Ebox asserts **E%IBOX_LOAD_PC_L**. In addition, when the BPU loads the PC, bits <2:0> of the new PC are decoded and used to set the PFQ pointer.

7.3 Instruction Parsing

The instruction parser identifies the different components of incoming VAX instructions and forwards those components to other parts of the Ibox for further processing. The instruction parser contains two logic sub-sections - the Instruction Burst Unit (IBU) and the Instruction Issue Unit (IIU).

Figure 7–8: Prefetch Queue Block Diagram



The IBU parses incoming instruction data into Opcodes, Operand Specifiers and Specifier Extensions and Branch Displacements. This information is then passed on to the operand specifier processing logic. The opcode is also sent to the IIU which generates an Ebox microcode entry point for this opcode and places it and other needed information in the instruction queue in the Ebox. See Table 7–15 for more information on the format of the Ebox instruction queue.

Instruction parsing is logically divided into 2 distinct activities: Instruction issue and specifier identification, and branch displacement and Ebox assist processing. The instruction issue and specifier identification activity starts when a new opcode is loaded by the IBU. The IBU sends the opcode to the IIU for issuing to the Ebox. The instruction opcode is also used to determine the number of operand specifiers and branch displacements associated with the instruction. In parallel with instruction issue, the IBU identifies the operand specifiers. When all the operand specifiers are processed, the IBU begins the branch displacement and Ebox assist processing activity. The branch displacement (if present) is sent to the BPU, and Ebox assist specifiers (if present) are processed. See Section 7.3.2.7 for more on Ebox assists.

7.3.1 VAX Instruction Format

There are 3 components in VAX instructions: opcodes, operand specifiers and specifier extensions, and branch displacements. The 1 or 2 byte opcode specifies the function to be performed. Operand specifiers with potential extensions range from 1 to 9 bytes and specify an instruction operand or operand location. The 1 or 2 byte branch displacements are signed offsets used to compute the destination PC in branch instructions. A VAX instruction is composed of an opcode and optionally up to 6 operand specifiers and one branch displacement. For a given opcode. the number of operand specifiers and branch displacements is fixed.

The instruction opcode is the first one or two bytes in the instruction followed by the operand specifiers, followed by the branch displacement, all at successively increasing addresses. All references to opcodes in this section refer to one-byte opcodes unless specified otherwise. For more information on VAX instruction formats, opcodes, and operand specifiers, see DEC STD 032, VAX Architecture Standard.

7.3.2 The Instruction Burst Unit

The IBU bursts apart Istream data into its component parts: opcodes, operand specifiers, and branch displacements. The IBU is capable of identifying an opcode and one operand specifier each cycle. Operand specifiers are categorized according to the their Addressing Mode as being either simple or complex. Simple specifiers are register mode (Addressing Mode 5) and short literal (Addressing Modes 0..3). All other specifier types, including assists, are considered complex.

The IBU retires up to 6 bytes of data from the PFQ each cycle. New data is available from the PFQ at the beginning of a cycle. The IBU sends the number of specifier bytes being retired back to the PFQ so that new data is available for processing by the next cycle.

Instruction components extracted from the Istream data are sent to other parts of the Ibox for further processing. The opcode is sent to the IIU and the BPU on **OPCODE<8:0>**. The specifiers, except for branch displacements, are sent to the CSU, the SBU and the OQU via **SPEC_CTRL<21:0>**. Branch displacements are sent to the BPU on **B_BRANCH_DISP<7:0>** and **SPEC_DATA<7:0>**.

The specifier control field **SPEC_CTRL<21:0>** contains information about the specifier being retired each cycle. **SPEC_CTRL<21:14>** and **SPEC_DATA<31:0>** contain information used in processing complex specifiers. Table 7–9 describes the information contained on these busses.

Table 7–9: Specifier Control Fields

Bit Field	Field Name	Description
<0>	SHLIT	This bit is set if the specifier is a short literal.
<6:1>	RN/SHORT LITERAL	Contains a 6-bit short literal if the shlit flag is set. <4:1> contains the general purpose register number associated with the specifier if the shlit flag is not set, in which case <6:5> are not used.
<9:7>	AT	Access Type of the instruction operand with which this operand specifier is associated.
<11:10>	DL	Data length of the instruction operand with which this operand specifier is associated.
<12>	VALID	Flags data valid on the bus.
<13>	COMPLEX	This bit is set if this is a complex specifier.

If the IBU is retiring a specifier, SPEC_CTRL<21:0> and SPEC_DATA<31:0> contain information about the specifier being retired. SPEC_CTRL<21:14> and SPEC_DATA<31:0> contain valid data used by the CSU only when the specifier is complex. If a simple specifier is being retired, the information on SPEC_CTRL<21:14> is invalid and not used by the CSU and the complex flag SPEC_CTRL<13> is not set. Table 7–10 describes the fields in SPEC_CTRL<21:14> used for complex specifiers. Table 7–11 describes the fields in SPEC_DATA<31:0> used by the CSU and BPU. When displacement and displacement deferred mode specifiers are processed, byte and word data length specifiers are sign extended to longword data length on SPEC_DATA<31:0>.

Table 7–10: Complex Specifier Control Fields

Bit Field	Field Name	Description
<16:14>	DISPATCH	Dispatch address for Complex Specifier Unit Control Store.
<17>	AT_RMW	1 if access type of operand is R, M or W.
<18>	INDEXED	This bit is set if mode of previous specifier is index.
<19>	ASSIST	This bit is set if this is an Ebox assist specifier.
<20>	PC_MODE	This flag is set if the bits <3:0> of the specifier point to GPR 15 = PC.
<21>	JMP_OR_JSB	This bit is set if this instruction is a JMP or JSB.

Table 7–11: Specifier Data Fields

Bit Field	Field Name	Description
<7:0>	WORD_DISP	Upper order byte of word displacement if branch displacement is being processed. Otherwise, the lower order byte of data for immediate and displacement mode specifiers.
<31:8>	SP_DATA	Upper 3 bytes of data for immediate and displacement mode specifiers.

7.3.2.1 Specifier Identification

In the instruction issue and specifier identification phase of instruction parsing, operand specifiers are parsed, and the necessary information about each specifier is sent to the specifier processing logic. The information needed by the Ebox to process the instruction is also identified and sent to the IIU. Each time a new opcode is loaded in the IBU, instruction context for that opcode is extracted from PLAs, complimentary logic, and the Instruction ROM (IROM). This information is summarized in Table 7–12.

As each specifier is identified, the current SPEC_COUNT is decremented. When this counter reaches 0, the IBU enters the next phase of instruction parsing, Ebox assists and branch displacements processing.

Table 7–12: Instruction Context Summary

Field Name	# bits	Description
		Instruction Context stored in the IROM
SPEC_COUNT	3	Number of specifiers for this instruction
STOP_PARSER	2	STP_SUPPRESS_PCQ/STP_RESTART_IBOX:
		0/0 Do not stop parser,make a PC queue entry for the next instruction.
		0/1 Stop parser at the end of the instruction, make a PC queue entry for the next instruction, and restart parser on E%RESTART_IBOX_H.
		1/0 Stop parser at the end of the instruction, suppress PC entry for next instruction until LOAD_NEW_PC is received, and restart parser on LOAD PC. See Table 7–14.
		1/1 Stop parser at the end of the instruction, suppress PC queue entry for next instruction until LOAD_NEW_PC is received, restart parser on E%RESTART_IBOX_H.
ASSIST_COUNT	1	Number of Ebox assists for this instruction
ASSIST	3	Assist dispatch
A_AT	2	Access type for Ebox Assist
A_DL	1	Data Length for Ebox Assist

Table 7–12 (Cont.): Instruction Context Summary

Field Name	# bits	Description
		Instruction Context stored in the IROM
A_REG	1	Register for Ebox Assist
AT1	3	Access type for specifier # 1
AT2	3	Access type for specifier # 2
DL1	2	Data length for specifier # 1
DL2	2	Data length for specifier # 2
FB	1	1 when this is an Fbox instruction
DISPATCH	9	Ebox microcode dispatch address
E_DL	2	Data length for instruction execution
		Instruction Context stored in the PLAs
AT3	3	Access type for specifier # 3
AT4	3	Access type for specifier # 4
AT5	1	Access type for specifier # 5
AT6	1	Access type for specifier # 6
DL3	2	Data length for specifier # 3
DL4	2	Data length for specifier # 4
DL5	2	Data length for specifier # 5
DL6	1	Data length for specifier # 6
B	1	Indicates that there is a branch displacement.
DISP_SIZE	1	Size of the branch displacement. 0 = byte displacement, 1 = word.
		Instruction Context decoded by logic
VFIELD_SPEC	1	Indicates how many source queue entries to allocate for RMODE (Mode 5) specifiers with variable bit field access type. 0 = 1 entry, 1 = 2 entries.

Each cycle, the IBU evaluates the following information to determine if an operand specifier is available and how many PFQ bytes should be retired to get to the next opcode or specifier:

- The number of PFQ bytes available. Each cycle, the PFQ provides the IBU with the number of instruction stream bytes available on AVAIL_LE<5:0>. This can be as little as 0 and as many as 6.
- The number of specifiers left to be parsed in the instruction stream. IBU keeps a running count of the number of specifiers left to be parsed for the current instruction.
- The data length of the next specifier.
- The COMPLEX_UNIT_BUSY flag S1_VALID. When the CSU is busy and cannot accept another complex specifier, S1_VALID is asserted. If the IBU identifies a complex specifier while this signal is asserted, it stalls until the flag is cleared by the CSU.

- DATA_LENGTH_VALID flag. This flag is asserted when the instruction PLAs have valid data length information ready. This flag is cleared when a new opcode is loaded and set when the access type and data length information is available for use.
- Specifier bus enable flag, SPEC_CTRL_ENABLE, from the OQU. This flag enables the loading of specifier information onto the specifier control bus. If SPEC_CTRL_ENABLE is 1 then the specifier control bus is enabled, and one specifier can be processed. If SPEC_CTRL_ENABLE is 0 then no specifiers can be processed, and the IBU stalls.
- The parser stopped flag PARSER_STOPPED. There are many times when the parser must be stopped to prevent it from interfering with Ebox activity. When this is necessary, PARSER_STOPPED is asserted and all parser activity stops.
- The next 2 bytes of the instruction stream.

If the specifier byte is a simple specifier (Addressing Modes 0..3, or 5), and the following conditions are met, then the information for this specifier is driven onto SPEC_CTRL<12:0>, and the specifier byte is retired from the PFQ at the end of the cycle:

1. There are at least 2 bytes of valid PFQ data. (At least one byte in the specifier field and one byte in the opcode field.)
2. The parser is not stopped.
3. There is at least one specifier remaining for this instruction.
4. SPEC_CTRL_ENABLE = 1.

If the first specifier byte is a complex specifier, and the following conditions are met, then the information for this specifier is driven onto SPEC_CTRL<21:0> and SPEC_DATA<31:0>, and the appropriate number of PFQ bytes for this specifier are retired from the PFQ at the end of the cycle:

1. The number of bytes required according to the Addressing Mode and Data Length of the specifier (plus one for the opcode field) are available from the PFQ.
2. The parser is not stopped.
3. There is at least one specifier remaining for this instruction.
4. SPEC_CTRL_ENABLE = 1.
5. COMPLEX_UNIT_BUSY flag is not asserted.

7.3.2.2 Operand Access Types

There are 6 different access types for operands. The access type information determines whether the operand is a source or destination operand, and whether the operand, or the address of the operand is needed by the Ebox. These access types are modeled after, but are not identical to, the operand access types specified in the architectural summary.

- **A (Address)**
  An operand with access type = A is a source operand. The Ebox gets the address of the operand, not the actual operand.
- **R (Read)**
  An operand with access type = R is a source operand. The Ebox gets the actual operand.
- **M (Modify)**
  An operand with access type = M is both a source and a destination. The Ebox gets the actual operand and a pointer to the destination.

- **W (Write)**
  An operand with access type = W is a destination operand. The Ebox gets a pointer to the destination.
- **VR (Variable bit field read-access)**
  An operand with access type = VR is a source operand. The Ebox gets the actual operand if the addressing mode of the specifier for the operand is RMODE (Mode 5). Otherwise the Ebox gets the address of the operand.
- **VM (Variable bit field modify-access)**
  An operand with access type = VM is both a source and a destination. The Ebox gets the actual operand if the addressing mode of the specifier for the operand is RMODE (Mode 5). Otherwise the Ebox gets the address of the operand. If the operand specifier is RMODE, the Ebox gets a pointer to the destination. Otherwise no destination pointer is supplied.

7.3.2.3 DL stall

For all but one addressing mode, the number of bytes to retire for a specifier is determined entirely by the addressing mode. Immediate mode (8F) addressing, however, requires the data length information for the operand to determine how many PFQ bytes to retire. In the event that a new opcode is loaded and the first specifier is an immediate mode specifier, the absence of DATA_LENGTH_VALID causes the IBU to stall because there is no way to determine the number of PFQ bytes to retire for this specifier. DATA_LENGTH_VALID is asserted the following cycle after the opcode has passed through the instruction PLAs and IROM to generate the required data length information. The immediate mode specifier can be retired the following cycle if the conditions described above are met.

7.3.2.4 Driving SPEC_CTRL

The data on SPEC_CTRL<13:0> is used by the OQU to generate Ebox source queue and destination queue entries that may be needed in the next cycle. The data on SPEC_CTRL<21:14> is used by the CSU to generate the microcode dispatch addresses. SPEC_DATA<31:0> contains instruction stream data for Immediate and Displacement mode specifiers.

7.3.2.5 PC and Delta_PC

The IBU keeps a local copy of the PC called the IBU_PC which points to the next byte of I stream data that will be processed by the IBU.

When the IBU retires instruction stream data, the IBU_PC is incremented by the number of operand and operand specifier bytes retired as signaled by SPEC_BYTES_RETIRED and LOAD_NEW_OPCODE. The IBU_PC can be loaded from the NEW_PC<31:0> when the signal LOAD_NEW_PC is asserted and all operand specifier, Ebox assist, and branch displacement processing is completed by the IBU. The IBU_PC is sent to the CSU, IIU and BPU on IBU_PC<31:0>.

7.3.2.6 Branch Displacement Processing

Some instructions have branch displacements as indicated by B. If B is set, the instruction has a branch displacement and the branch size is determined by DISP_SIZE. Both B and DISP_SIZE are outputs of the instruction PLAs. A DISP_SIZE of 0 indicates a byte branch displacement and a DISP_SIZE of 1 indicates a word displacement.

The branch displacement is always the last piece of data for an instruction and is used by the BPU to compute the branch destination. Branch displacements are not sent to the specifier parsing logic. They are sent only to the BPU on SPEC_DATA<7:0> and B_BRANCH_DISP<7:0>. Branch displacement processing begins after all the non-displacement specifiers are parsed and retired from the PFQ. A branch displacement is processed when the following conditions are met:

1. There are no specifiers left to be processed (Ebox assists excluded).
2. The branch flag B<0> is set in the instruction PLAs and the branch displacement has not been processed.
3. The required number of bytes is available from the PFQ according to DISP_SIZE.
4. The parser is not stopped.
5. BRANCH_STALL is not asserted. BRANCH_STALL occurs on the load opcode of the next instruction after a second conditional branch is received.

BRANCH_STALL is described in the Section 7.5.1.6 section.

If all these conditions are met, then the branch displacement is placed on SPEC_DATA<7:0> and B_BRANCH_DISP<7:0> and DISP_VALID is asserted. SPEC_DATA<7:0> contains the high byte of a word branch displacement and B_BRANCH_DISP<7:0> contains the low byte of a word branch displacement or the byte branch displacement. If these conditions are not met, the IBU stalls.

If an instruction contains no operand specifier, the branch displacement can be processed during the same cycle that the opcode is processed provided that there is sufficient data in the PFQ.

7.3.2.7 Ebox Assist Processing

Ebox assist processing can go on in parallel with branch displacement processing since they require no common resources. Ebox assists are implicit specifiers which help the Ebox speed up some of the time critical instructions. To the CSU, these assists look very similar to normal complex specifiers and have associated with them all the normal access type, data length and register information. The only real difference is where this data comes from. Since these specifiers are not a part of the instruction stream, information about them must be stored in the IROM. The 7 Ebox assists are summarized in the following table:

Table 7–13: Ebox Assist Summary

Assist Name	Access Type	Data Length	Register	Description
RET_DEST	Read	Quad	FP	Read register mask for Ebox. Read return PC for Ebox and BPU

Table 7–13 (Cont.): Ebox Assist Summary

Assist	Access	Data	Register	Description
Name	Type	Length		
RSB_DEST	Read	Long	SP	Read return PC for Ebox and BPU
(SP)+.RQ	Read	Quad	SP	Quadword stack pop
–(SP).WL	Write	Long	SP	Longword stack push
PC.RL	Read	Long	NONE	Current PC is sent to Ebox
PC.-(SP).ML	Modify	Long	SP	Combines effects of PC.RL and -(SP).WL assists
STOP.MBOX.QUEUE	NONE	NONE	NONE	Mbox specifier queue is stopped

All of the Ebox assists generate dispatches to the CSU.

When all the normal specifiers for an instruction have been identified and retired from the PFQ, the Ebox assist (if any) is processed. The maximum number of assists for any instruction is 1.

An Ebox assist is processed and its associated data driven onto SPEC_CTRL<21:0> when the following conditions are met:

1. There is an Ebox assist.
2. The parser is not stopped.
3. It is not the same cycle as the opcode load.
4. If the instruction is BSBW or BSBB, the branch displacement has been parsed.
5. SPEC_CTRL_ENABLE = 1.
6. COMPLEX_UNIT_BUSY flag is not asserted.

BSBW and BSBB instructions have PC.RL Ebox assists. For these instructions, the branch displacement must be retired and the IBU_PC must be updated to point to the byte following the branch displacement before the PC.RL assist can be processed.

7.3.2.8 Reserved Addressing Modes

Some combinations of specifier mode, specifier register, and access type cause reserved addressing mode faults in the VAX architecture. Refer to Table 7–33 for more details on reserved address mode detection.

7.3.2.9 Quadword Immediate Specifiers

Immediate mode specifiers with quadword data length take two or more cycles to process. When a quadword immediate specifier is detected by the IBU parse logic, the first longword is processed (like a longword immediate specifier) and QUAD_FLAG, is set.

QUAD_FLAG is used by the IBU retire logic to properly retire the next four bytes when they become available in the PFQ. When the second longword is retired, QUAD_FLAG is cleared and the specifier count is decremented. QUAD_FLAG is also cleared by E%BRANCH_MISPREDICT_H, E%STOP_IBOX_H, I%IMEM_MEXC_H, and I%IMEM_HERR_H.

The first longword of the quadword immediate data is sent to the CSU in the normal fashion. The second longword of the quadword immediate data from the instruction stream is discarded. The CSU then uses the specifier PC and generates a memory request to fetch the next four bytes of the immediate data.

7.3.2.10 Index Mode Specifiers

Index mode specifiers are two-part specifiers which take two or more cycles to process. The first byte of an index mode specifier specifies the index register; it is treated like any other complex specifier with the exception that a flag, index_wait is set, and the specifier counter is NOT decremented. Additionally, **SPEC_CTRL**<21:17> is ignored by the CSU.

When the second byte of an index mode specifier is processed, the specifier counter is decremented and **SPEC_CTRL**<21:17> contains the appropriate data. **SPEC_CTRL**<18> is set and index_wait is cleared.

The reserved addressing mode fault PLA in the IBU checks the mode of the second specifier byte. If the index_wait is set, and if the second byte is short literal, register mode, or index mode, a reserved addressing mode fault is detected and sent to the Ebox on **I%RSVD_ADDR_FAULT_H**. Refer to Table 7–33 for more details on reserved addressing mode detection.

7.3.2.11 Loading a new opcode

A new opcode is loaded in the IBU under the following conditions:

1. All operand specifiers, branch displacements and Ebox assists for the current instruction have been parsed (which asserted **INSTR_DONE**).
2. The parser is not stopped.
3. There is at least one byte of data available from the PFQ.
4. **ISSUE_STALL** is not being asserted by the IIU.
5. **BRANCH_STALL** is not being asserted by the BPU.

New opcodes are loaded and passed directly to the instruction PLAs and IROM. In parallel, the instruction issue and specifier identification process for the new instruction begins.

When a the new opcode is loaded, a check is made to see if the value of the opcode is FD. If it is, no instruction parsing is done this cycle. **FD_OPCODE** is set, the byte is retired from the PFQ, and another opcode load is enabled for the following cycle. The opcode sent to the IIU and the BPU on **OPCODE**<8:0> is a concatenation of **FD_OPCODE** and the opcode byte. **FD_OPCODE** is bit 8, and the opcode is in <7:0>.

7.3.2.12 Reserved Opcodes

Each time a new opcode is loaded in the IBU, instruction and operand specifier information is extracted from a set of PLAs and from the IROM in the IBU for that opcode. This information is specified in Table 7–12. When a reserved or unimplemented opcode is detected, the following occurs:

1. The IBU IROM has one of the **STOP_PARSER** bits set. This signals the IBU to stop parsing instruction stream data.
2. The IBU IROM provides the reserved opcode dispatch address for Ebox microcode.

7.3.2.13 Instruction Parse Completion

Once all the operand specifiers, branch displacements and Ebox assists have been processed, instruction parsing is complete and **INSTR_DONE** is asserted. **INSTR_DONE** is used by the CSU to make RLOG base queue entries and by the IBU to control loading of the **BPU_PC** under certain conditions.

Additionally, if instruction parsing is complete and if there is no PC load pending, **RETIRE_OPCODE** is asserted and sent to the PFQ control logic and the IIU PC queue logic. In the PFQ this signal increments the number of specifier bytes retired by 1 in order to retire the previous opcode and allow for loading of the new opcode. It is used in the IIU to update the PC queue pointer under certain conditions.

7.3.2.14 Operands with Access Type VR and VM

One of the outputs from the instruction PLAs is a bit that indicates how many source queue entries should be written for VR and VM access type operands with register mode specifiers. When this bit is 0, only one source queue entry is written; when it is 1, two are written. This bit is available in the middle of the opcode load cycle and is sent to the OQU on **VS**. This signal remains valid throughout the instruction parsing operation.

7.3.2.15 I%IMEM_MEXC_H and I%IMEM_HERR_H

The IBU forwards Istream errors to the Ebox on **I%IMEM_HERR_H** and **I%IMEM_MEXC_H**. These signals flag memory management exceptions and hardware errors. The IBU receives three error signals from the VIC which are used to determine when to assert **I%IMEM_HERR_H** and **I%IMEM_MEXC_H**: **IHARD_ERR**, **MHARD_ERR**, and **IMMGT_EXC**. Refer to Section 7.2.1.7 for more detail on these signals.

The IBU asserts **I%IMEM_MEXC_H** if **IMMGT_EXC** is asserted from the VIC and the PFQ is empty or contains insufficient data to complete parsing of the current specifier, and parsing is not stopped. **I%IMEM_MEXC_H** remains asserted as long as these conditions are met.

The IBU asserts **I%IMEM_HERR_H** under two different conditions. First, if **MHARD_ERR** is asserted from the VIC and the PFQ is empty or contains insufficient data to complete parsing of the current specifier, and parsing is not stopped. Additionally, if **IHARD_ERR** is asserted from the VIC, **I%IMEM_HERR_H** is asserted immediately without waiting for the PFQ to run dry or contain insufficient data. **I%IMEM_HERR_H** remains asserted as long as these conditions are met.

7.3.2.16 IBU stop and restart conditions

Two categories of conditions cause the IBU to stop parsing: the first is exceptions, the second is instructions which need pipeline synchronization. When the IBU is stopped, **PARSER_STOPPED** is asserted.

Table 7–14 summarizes all IBU stop and restart conditions.

Table 7–14: IBU stop and start summary

Stop Condition	Start Condition	Description
E%STOP_IBOX_H	E%RESTART_IBOX_H	stop ibox, Ebox restarts parser
I%RSVD_ADDR_FAULT_H	E%RESTART_IBOX_H	reserved addressing mode fault, Ebox restarts parser
IHARD_ERR	E%RESTART_IBOX_H	VIC hardware error, Ebox restarts parser
FPD and load opcode	E%RESTART_IBOX_H	FPD is set, parse opcode and stop parser, Ebox restarts parser
E%BRANCH_MISPREDICT_L	I_CSU%IBOX_RESTART	branch mispredict, ibox restarts parser
stop parser set - case 1	I_CSU%IBOX_RESTART	parser stopped when STP_RESTART_IBOX and INSTR_DONE are both asserted, ibox restarts parser
stop parser set - case 2	I_IBU%CSU_LD_RESTART	parser stopped when STP_SUPPRESS_PCQ and and INSTR_DONE are both asserted and STP_RESTART_IBOX is de-asserted, restart occurs when the CSU supplies the BPU with the new PC and all other instruction parsing is complete

7.3.2.17 First Part Done (FPD) Set

Some long instructions can be interrupted in the middle of their execution sequence (e.g. MOVC instructions). When such an instruction is interrupted, the first part done bit (FPD) in the Processor Status Longword (PSL) is set indicating that the interrupted instruction will be resumed at the execution point where the interrupt occurred, rather than at the beginning of the instruction. All such instructions have one of the **STOP_PARSER** bits set in the IROM. This allows the FPD pack-up to IPR read the current PC (from the top of the PC queue) and then load the PC of the interrupt handler.

When an instruction such as MOVC is interrupted, and the interrupt is processed, processor context is switched back to the interrupted process by the REI instruction. This instruction causes the PSL of the interrupted process to be reloaded with the FPD bit set. The Ebox sends the **E%FPD_SET_L** signal to the Ibox. If **E%FPD_SET_L** is asserted the Ibox will re-issue the interrupted instruction when valid opcode data is parsed by the IBU. However, after parsing and issuing the instruction, no further data is parsed by the IBU.

When the interrupted instruction is complete, the Ebox loads the PC of the next instruction and parsing is restarted by the IBU.

7.3.3 The Instruction Issue Unit

The IIU takes opcodes received from the IBU and generates the information needed by the Ebox to begin instruction execution. An instruction is said to be issued when this information is sent to the Ebox instruction queue. Table 7–15 shows the format of the instruction queue entries created by the IIU. This information is sent to the Ebox on I%IQ_BUS_H<21:0>.

The IIU must also keep track of the program counter (PC) values of the opcodes that are either in the instruction queue or are in Ebox execution. If the Ebox detects a fault during the execution of an instruction, it needs to be able to get at the PC of the faulting opcode. These PCs are kept in the PC queue.

Table 7–15: Instruction Queue Entry Format

Bit Field	Field Name	Description
<0>	VALID	1 when this queue entry is valid
<9:1>	DISPATCH	Ebox microcode dispatch address
<10>	FB	1 when this is an Fbox instruction
<12:11>	DL	Data length for instruction execution
<21:13>	OPCODE	Instruction Opcode

Most of the information needed to create an instruction queue entry is stored in the instruction ROM located in the IBU. See Table 7–12. The opcode used to access the ROM is a 9-bit composite opcode consisting of 8 true opcode bits and 1 bit indicating whether or not this is a two byte FD opcode. This extra bit is generated by the IBU and passed along with the other 8 opcode bits.

The IIU issues an instruction as soon as the instruction ROM access completes unless the instruction queue is full. The instruction queue full status is computed and maintained locally in the IIU.

7.3.3.1 Issue Stall

The IIU maintains a counter of the number of slots filled in the Ebox instruction queue. Each time a new opcode is issued to the IIU, the counter is incremented. When the Ebox removes an entry from the queue as indicated by the E%RETIRE_INSTR_L signal, the counter is decremented. When the counter equals 6, the depth of the instruction queue, ISSUE_STALL is asserted, blocking the IBU from parsing a new opcode.

7.3.3.2 PC Queue and PC loads

The PC queue is a 7 entry FIFO which contains PC values of opcodes that are either in the instruction queue or are in Ebox execution. Opcode PCs are added to the back of the queue as instructions are issued and removed from the front of the queue when the Ebox retires an instruction as indicated by E%RETIRE_INSTR_L. The PC of the next instruction to be retired by the Ebox is always at the front of the queue unless the PC queue is empty. The PC queue is flushed on chip reset or when either E%FLUSH_PCQ_H or E%BRANCH_MISPREDICT_L is asserted by the Ebox.

Any time the Ibox broadcasts a new PC on NEW_PC<31:0>, as signaled by LOAD_NEW_PC, it is loaded into the next available slot in the PC queue. If E%BRANCH_MISPREDICT_L caused the PC load or if the Ebox stops the Ibox as signaled by E%STOP_IBOX_H, then following additional actions are taken:

- The instruction queue counter is cleared.
- ISSUE_STALL is cleared if set.

In the event of an Ebox PC load, the parser is guaranteed to stop either by E%STOP_IBOX_H, STP_SUPPRESS_PCQ, or STP_RESTART_IBOX several cycles before the actual PC load occurs. These signals are used in the IBU to stop instruction parsing. When the new PC arrives, the PC queue is empty and ready to accept the new PC into the first available slot.

The value of STP_SUPPRESS_PCQ affects whether the PC queue loads the next PC as the parser stops. If STP_SUPPRESS_PCQ is asserted then the next PC is entered in the PC queue.

The value of the IBU_PC is loaded into the PC queue if LOAD_NEW_PC is not asserted, the burst unit signals that the parsing is complete with RETIRE_OPCODE, E%FPD_SET_L is not asserted, and either of the following conditions are true:

- STP_SUPPRESS_PCQ is not asserted or STOP_VIC_PREFETCH is not asserted, and the BPU is not stalled
- BSTL_FRC_PCQ (from the BPU) is asserted and the instruction is done.

The PC at the front of the PC queue is readable by the CSU. When the Ebox needs access to this PC, it stops the Ibox and sends an IPR read request to the CSU. The CSU responds by reading the front of the PC queue and then writing that value to the Ebox working register (WX) specified by a register index supplied with the IPR command. See Section 7.4.2.8 for more details on IPR transactions.

MICROCODE RESTRICTION

For proper operation, retire_instr and IPR read of the BPC (Backup PC) from the PC queue must not occur in the same microword. This guarantees that the PC queue does not decrement in the same cycle that an IPR read of the BPC occurs.

7.4 Operand Specifier Processing

Operand Specifier Parsing prepares instruction operands for access by the Ebox. The three Ibox sub-sections which together perform this function are the Operand Queue Unit (OQU), the Complex Specifier Unit (CSU), and the Scoreboard Unit (SBU). The OQU handles simple specifiers and acts as the interface to the Ebox source and destination queues; the CSU is responsible for processing complex specifiers, and the SBU provides the CSU with information about the number of outstanding GPR read and write references in the source and destination queues.

7.4.1 Operand Queue Unit

The OQU controls the passing of operand information into the Ebox operand queues and the allocation of Ebox Memory Data registers (MDs).

Simple specifiers are processed entirely in the OQU. Register mode specifiers are passed into the source or destination queues as pointers to the corresponding Ebox register file location. The OQU passes short literal specifiers as immediate data.

The 6 MD registers in the Ebox register file are used as destinations for operand data requests made by the CSU. When a complex specifier appears on the specifier control bus, the OQU allocates both the source queue entries and Ebox MDs and passes the Ebox register file index of the first allocated MD to the CSU.

The **I%OPERAND_BUS_H<14:0>** transfers source and destination queue entry information to the Ebox. There may be up to 2 source queue entries and 2 destination queue entries made via the **I%OPERAND_BUS_H<14:0>** in a given cycle. The format for this bus is shown in Figure 7–9.

Short literals:

Figure 7–9: Source/Destination Queue Entry Formats

```
SHORT LITERAL Mode:
 14  13  12 |11  10   9   8 | 7   6   5   4 | 3   2   1   0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   | 0 | 0 | 1 |                       |               | <--I%OPERAND_BUS_H
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  |   |           |             |                   |     MBZ if SQ_VALID2=1
  |   |           |             |                   +---short literal value2 (quad)
  |   |           |             +-----------------------short literal value1
  |   |           +-------------------------------------SHLIT (1=short lit)
  |   +-------------------------------------------------SQ_VALID2 (1=quad operand)
  +-----------------------------------------------------SQ_VALID1

Register Mode:
 14  13  12 |11  10   9   8 | 7   6   5   4 | 3   2   1   0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   | 0 |   | 1 |      GPRn     |    GPRn+1     | <--I%OPERAND_BUS_H
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |         |               |
  |   |   |   |   |   |   |         |               +---REG2 (GPRn+1 tag for quad)
  |   |   |   |   |   |   |         +-------------------REG1 (GPRn tag)
  |   |   |   |   |   |   +-----------------------------GPR (1=gpr)
  |   |   |   |   |   +---------------------------------VFIELD (1=Field Queue Entry)
  |   |   |   |   -------------------------------------SHLIT (0=not short lit)
  |   |   |   +-----------------------------------------DQ_VALID2 (1=quad w/m operand)
  |   |   ----------------------------------------------DQ_VALID1
  |   +-------------------------------------------------SQ_VALID2 (1=quad r/m operand)
  ------------------------------------------------------SQ_VALID1

All Other Modes for access types read and modify:
 14  13  12 |11  10   9   8 | 7   6   5   4 | 3   2   1   0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 1 |   |   |   | 0 |   | 0 | 1    MDn      | 1  MDn+1      | <--I%OPERAND_BUS_H
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |         |               |
  |   |   |   |   |   |   |         |               +---REG2 (MDn+1 tag for quad)
  |   |   |   |   |   |   |         +-------------------REG1 (MDn tag)
  |   |   |   |   |   |   +-----------------------------GPR (0=MD)
  |   |   |   |   |   +---------------------------------VFIELD (1=Field Queue Entry)
  |   |   |   |   +-------------------------------------SHLIT (0=not short lit)
  |   |   |   +-----------------------------------------DQ_VALID2 (1=quad w/m operand)
  |   |   +---------------------------------------------DQ_VALID1
  |   +-------------------------------------------------SQ_VALID2 (1=quad r/m operand)
  +-----------------------------------------------------SQ_VALID1

All Other Modes for access type write:
 14  13  12 |11  10   9   8 | 7   6   5   4 | 3   2   1   0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 | 0 | 1 |   | 0 |   | 0 |     GPRn      |    GPRn+1     | <--I%OPERAND_BUS_H
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
  |   |   |   |   |   |   |         |               |
  |   |   |   |   |   |   |         |               +---REG2 (GPRn+1 for quad)
  |   |   |   |   |   |   |         +-------------------REG1 (GPRn tag)
  |   |   |   |   |   |   +-----------------------------GPR (0=mdest)
  |   |   |   |   |   +---------------------------------VFIELD (1=Field Queue Entry)
  |   |   |   |   +-------------------------------------SHLIT (0=not short lit)
  |   |   |   +-----------------------------------------DQ_VALID2 (1=quad w/m operand)
  |   |   +---------------------------------------------DQ_VALID1
  |   +-------------------------------------------------SQ_VALID2 (1=quad r/m operand)
  +-----------------------------------------------------SQ_VALID1
```

Table 7–16: I%OPERAND_BUS_H Definition

Bit Field	Field Name	Description
VALUE2	3:0	Upper bits for quadword short literal, must be zero's
VALUE1	9:4	Short literal value. Lower bits for quadword
SHLIT	10	Short literal. 1 if short literal, 0 otherwise
DQ_VALID2	11	Valid second destination queue entry - always 0 for short literal
DQ_VALID1	12	Valid destination queue entry - always 0 for short literal
SQ_VALID2	13	Valid second source queue entry - set if quadword short literal
SQ_VALID1	14	Valid source queue entry

All other modes:

Table 7–17: I%OPERAND_BUS_H Definition

Bit Field	Field Name	Description
REG2	3:0	Register or MD for 2nd source/dest queue entry of a quadword specifier
REG1	7:4	Register or MD for 1st source/dest queue entry
GPR	8	Source/dest queue entry is for a register mode specifier
VFIELD	9	Field queue entry to be made
SHLIT	10	Short literal. 1 if short literal, 0 otherwise
DQ_VALID2	11	Valid second destination queue entry for quadword specifiers
DQ_VALID1	12	Valid destination queue entry
SQ_VALID2	13	Valid second source queue entry for quadword specifiers
SQ_VALID1	14	Valid source queue entry

7.4.1.1 Source Queue Interface

The OQU can write up to two source queue entries each cycle depending on the access type and data length of the operand they specify. I%OPERAND_BUS_H<SQ_VALID1> and I%OPERAND_BUS_H<SQ_VALID2> are the source queue entry valid bits. I%OPERAND_BUS_H<SQ_VALID1> indicates that the information on I%OPERAND_BUS_H<10:4> is for a valid source queue entry. I%OPERAND_BUS_H<SQ_VALID2> indicates the information on I%OPERAND_BUS_H<3:0> is for a valid source queue entry. I%OPERAND_BUS_H<10:4> contains the information for any specifier that is placed on SPEC_CTRL. I%OPERAND_BUS_H<3:0> contains the second source queue entry whenever the specifier on SPEC_CTRL has an access type of Read or Modify and a data length of quadword or it is an RMODE specifier with access type VR or VM and the VS bit is set. I%OPERAND_BUS_H<SQ_VALID2> is set only if I%OPERAND_BUS_H<SQ_VALID1> is set.

The addressing mode of the operand specifiers determines the value of the source queue entries. For short literal (Modes 0..3) addressing modes, **I%OPERAND_BUS_H**<VALUE1> contains the short literal data directly, with **I%OPERAND_BUS_H**<SHLIT> set. Source queue entries for register (Mode 5) addressing mode specifiers contain pointers to the referenced GPR, with **I%OPERAND_BUS_H**<GPR> set and **I%OPERAND_BUS_H**<SHLIT> cleared. Source queue entries for all other addressing modes contain pointers to Memory Data (MD) registers in the Ebox, with **I%OPERAND_BUS_H**<GPR> and **I%OPERAND_BUS_H**<SHLIT> both cleared. **I%OPERAND_BUS_H**<VFIELD>, is set for variable bit field specifiers and cleared otherwise. This bit is used by the Ebox to make Field Queue entries.

The access type and data length of the operand being specified determines the number of source queue entries that are written for all operands except those with access types VR or VM. Read (R) and Modify (M) access type operands write one source queue entry if the operand data length is byte, word, or longword, and two source queue entries if the operand data length is quadword. Write (W) access type operands never write any source queue entries. Address (A) access type operands always write one source queue entry regardless of the operand data length. The number of source queue entries written for non-field access type operands is summarized in Table 7–18.

Table 7–18: Source Queue Entries Written for Non-field Access Type Operands

Access Type	Data Length	Number of Source Queue Entries written
Read (R)	Byte, Word, Long	1 source queue entry written
Modify (M)	Byte,Word, Long	1 source queue entry written
Write (W)	Byte, Word, Long, Quad	0 source queue entries written
Address (A)	Byte, Word, Long, Quad	1 source queue entry written
Read (R)	Quad	2 source queue entries written
Modify (M)	Quad	2 source queue entries written

For VR and VM operands, the **VS** bit associated with the instruction and the addressing mode determine the number of source queue entries that are written. For these variable bit field access type operands, **VS** performs a function similar to the data length in non-field operands. The **VS** bit specifies how many source queue entries to write for VM and VR operands with RMODE specifiers. The value of **VS** is ignored if the access type of the operand is not VR or VM. If **VS** is 0 then one source queue entry is written for VR and VM operands with an RMODE specifier. If **VS** is 1 then two source queue entries are written for VR and VM operands with an RMODE specifier. Only one source queue entry is written for VR and VM operands with non-RMODE specifiers, regardless of the value of **VS**. Table 7–19 shows the number of source queue entries written for operands with VR or VM access types.

Table 7–19: Source queue Entries Written for VR or VM Access Type Operands

VS	Access Type	Number of Source Queue Entries Written
0	RMODE	1 source queue entry written
1	RMODE	2 source queue entries written
X	non-RMODE	1 source queue entry written

VS is supplied by the IBU in the middle of the cycle in which the opcode is loaded and is held throughout the parsing of the instruction.

7.4.1.1.1 Short Literal Specifiers (Modes 0..3)

Short literal specifiers create a source queue entry with the SHLIT flag set and the short literal data in I%OPERAND_BUS_H<VALUE1>. The short literal data is the full RN_SHORT_LITERAL<6:1> from the specifier control bus. For quadword operands the OQU writes two source queue entries. In this case, I%OPERAND_BUS_H<VALUE2> is 0, I%OPERAND_BUS_H<VALUE1> contains the short literal value, I%OPERAND_BUS_H<SHLIT> is set, and I%OPERAND_BUS_H<SQ_VALID1> and I%OPERAND_BUS_H<SQ_VALID2> are both set to indicate 2 source queue entries.

Short literal addressing modes for VM and VR access type operands cause a reserved addressing mode fault to be signaled to the Ebox. All reserved addressing mode faults block the OQU from writing any source or destination queue entries. See Section 7.9.5 for details on these faults.

7.4.1.1.2 RMODE Specifiers (Mode 5)

Register mode specifiers create source queue entries with I%OPERAND_BUS_H<REG1> pointing to the specified Ebox GPR index and the SHLIT bit clear. The contents of I%OPERAND_BUS_H<REG1> are taken directly from the specifier control bus RN field. I%OPERAND_BUS_H<GPR> is equal to 1 for register mode operands. If two entries are allocated for an operand due to quadword data length or the VS bit, the value for the second entry on I%OPERAND_BUS_H<REG2> is the value of the first entry on I%OPERAND_BUS_H<REG1> incremented by 1 and modulo 16. For specifiers of type VR or VM, the I%OPERAND_BUS_H<VFIELD> is set to indicate a variable bit field specifier and cleared otherwise.

7.4.1.1.3 Index Mode Specifiers (Mode 4)

Indexed specifiers are processed by the IBU as two specifiers. Only the second specifier, the base, may create a source queue entry. The first specifier is recognized and ignored by the OQU if it is a complex specifier with the dispatch field of the specifier control bus pointing to index mode. Therefore, if SPEC_CTRL<COMPLEX> is set and SPEC_CTRL<DISPATCH> is index mode, then no source queue entries will be made for the specifier.

7.4.1.1.4 All Other Addressing Modes

Specifiers which are not literal or register mode create source queue entries with the I%OPERAND_BUS_H<REG1> fields pointing to Ebox MDs and the SHLIT and GPR bits clear. One MD is allocated for each source queue entry of this type written. See Section 7.4.1.4 for more detail on MD allocation. If two entries are allocated for an operand due to quadword data length or RMODE with the VS bit set, the I%OPERAND_BUS_H<REG2> field for the second entry is equal the I%OPERAND_BUS_H<REG1> field of the first incremented by 1 and modulo 6. The most significant bit for both I%OPERAND_BUS_H<REG1> and I%OPERAND_BUS_H<REG2> are set to 1 to correspond with Ebox register file addressing. For specifiers of type VR or VM, the VFIELD bit is set to indicate a variable bit field specifier and cleared otherwise. Only one specifier per instruction may be of access type VR or VM, so as not to overflow the field queue.

7.4.1.2 Destination Queue Interface

The OQU can write up to two destination queue entries each cycle depending on the access types and data lengths of the operands they specify. The addressing mode of the operand specifier determines the contents of the destination queue entries written. Destination queue entries for register (Mode 5) addressing mode specifiers contain pointers to the referenced GPR and the **GPR** flag is set to indicate a register mode destination. All destination queue entries for specifiers with an access type write will contain pointers to the referenced GPR, regardless of addressing mode. For non-register mode specifiers of access types read and modify, the **I%OPERAND_BUS_H**<REG1> and **I%OPERAND_BUS_H**<REG2> fields are used by the source queue and ignored by the destination queue. All addressing modes other than register mode (Mode 5) and short literal (Modes 0..3) clear the **GPR** flag to indicate a memory destination. **I%OPERAND_BUS_H**<DQ_VALID1> is set if there is a valid destination queue entry. **I%OPERAND_BUS_H**<DQ_VALID2> indicates a second destination queue entry is also valid. **I%OPERAND_BUS_H**<DQ_VALID2> will only be set if **I%OPERAND_BUS_H**<DQ_VALID1> is also set.

Short literal addressing mode specifiers for operands with access types of Write (W), Modify (M), and VM cause Reserved Addressing Mode Faults. Reserved Addressing Mode Faults block the OQU from writing any source or destination queue entries. See Section 7.9.5 for details on these faults.

The access type and data length of the operand being specified determines the number of destination queue entries that are written for all operands except those with with access types VR or VM. Write (W) and Modify (M) access type operands write 1 destination queue entry if the operand data length is byte, word, or longword, and two destination queue entries if the operand data length is quadword. The number of destination queue entries written for non-field access type operands is summarized in Table 7–20.

Table 7–20: Destination Queue Entries Written for Non-field Access Type Operands

Access Type	Data Length	Number of Destination Queue Entries Written
Read (R)	Byte, Word, Long	0 destination queue entries written
Modify (M)	Byte,Word, Long	1 destination queue entry written
Write (W)	Byte, Word, Long	1 destination queue entry written
Address (A)	Byte, Word, Long	0 destination queue entries written
Read (R)	Quadword	0 destination queue entries written
Modify (M)	Quadword	2 destination queue entries written
Write (W)	Quadword	2 destination queue entries written
Address (A)	Quadword	0 destination queue entries written

For VR access type operands no destination queue entries are written. For VM access type operands, the **VS** bit associated with the instruction and the addressing mode of the operand specifier determine the number of destination queue entries that are written. The **VS** bit specifies how many destination queue entries to write for VM access type operands with RMODE specifiers. The value of **VS** is ignored if the access type of the operand is not VM. If **VS** is 0 then one destination queue entry is written for VM access type operands with an RMODE specifier. If **VS** is 1 then two destination queue entries are written for VM access type operands with an RMODE specifier. VM access type operands with non-RMODE specifiers create no destination queue entries. Table 7–21 shows the number of destination queue entries written for operands with VM access type.

Table 7–21: Destination Queue Entries Written for VM Access Type Operands

VS	Access Type	Number of Destination Queue Entries Written
0	RMODE	1 destination queue entry written
1	RMODE	2 destination queue entries written
X	non-RMODE	0 destination queue entries written

7.4.1.2.1 RMODE Specifiers (Mode 5)

Register mode specifiers create destination queue entries with I%OPERAND_BUS_H<REG1> pointing to the specified Ebox GPR and the I%OPERAND_BUS_H<GPR> bit set. The contents of the I%OPERAND_BUS_H<REG1> field are taken directly from the specifier control bus RN field. If two entries are allocated for an operand due to quadword data length or the VS bit, I%OPERAND_BUS_H<REG2> for the second entry is I%OPERAND_BUS_H<REG1> incremented by 1 and modulo 16. I%OPERAND_BUS_H<DQ_VALID1> and I%OPERAND_BUS_H<DQ_VALID2>, the destination queue entry valid bits, are both set.

7.4.1.2.2 Index Mode Specifiers (Mode 4)

Indexed specifiers are processed by the IBU as two specifiers. Only the second specifier, the base, may create a destination queue entry. The first specifier is recognized and ignored when the specifier control bus has a complex specifier with the dispatch field pointing to index mode. In other words, if SPEC_CTRL<COMPLEX> is set and SPEC_CTRL<DISPATCH> equals index mode, then no destination queue entries will be made for the specifier.

7.4.1.2.3 All Other Addressing Modes

All other addressing modes create destination queue entries with the GPR bit clear. If two entries are allocated for an operand due to quadword data length or VM access type with the VS bit set, the GPR bit applies to both entries.

7.4.1.3 Queue Entry Allocation

The OQU maintains a count of available Source and Destination Queue entries using an up-down counter for each. When the OQU allocates source queue entries, the source queue counter increments by the number of entries allocated. When the OQU allocates destination queue entries, the destination queue counter.increments by the number of entries allocated. When the source queue counter equals 12, the source queue is full. When the destination queue equals 6, the destination queue is full.

The source and destination queue counters decrement whenever the Ebox retires entries from the respective queues. The signals E%SQ_RETIRE_H<1:0> and E%DQ_RETIRE_H<0> are generated by the Ebox, and indicate the number of source and destination queue entries, respectively, to be retired this cycle. Up to two source queue entries and one destination queue entry may be retired each cycle. The E%SQ_RETIRE_H<1:0> signal decode is demonstrated in Table 7–22

7.4.1.4 MD Allocation

MDs are allocated in the OQU using an up-down allocate counter and an index counter. When the OQU allocates a new MD, the allocate counter increments and the current value of the index pointer is sent to the CSU and then incremented modulo 6. Whenever a source queue entry which points to an MD is retired by the Ebox, the allocate counter decrements. The value of the allocate counter always represents the number of previously allocated MDs and the index counter always points to the next MD to allocate. When the allocate counter equals 6 there are no MDs left to allocate. The signals E%SQ_RETIRE_MD_H<1:0> are generated by the Ebox and indicate the number of MD source queue entries to be retired this cycle. The E%SQ_RETIRE_MD_H<1:0> signal decode is demonstrated in Table 7–22.

Table 7–22: Source Queue Entries Retired

E%SQ_RETIRE_H<>	# SQ Entries	E%SQ_RETIRE_MD_H<>	#MD SQ Entries
1:0	Retired	1:0	Retired
0 0	0	0 0	0
0 1	1	0 1	1
1 0	1	1 0	1
1 1	2	1 1	2

7.4.1.5 Specifier Bus Enable

The OQU applies back-pressure to the IBU whenever there are insufficient MDs or source and destination queue entries to hold more operands. SPEC_CTRL_ENABLE is driven by the OQU to enable the driving of specifier data on the specifier control bus. SPEC_CTRL_ENABLE, when asserted, allows the IBU to drive a specifier on SPEC_CTRL<21:0>.

The number of available source queue entries, destination queue entries, and MDs determine whether a specifier may be parsed by the IBU and driven on the specifier control bus. SPEC_CTRL_ENABLE is asserted if there are at least 2 source queue entries, 2 destination queue entries, and 2 MDs available.

7.4.1.6 E%STOP_IBOX and Branch Mispredict

The following actions take place when the Ebox issues a E%STOP_IBOX_H or a branch mispredict.

- The MD allocation counter and index counter are both reset to 0
- The source queue counter is reset to 0
- The destination queue counter is reset to 0
- Any specifiers currently being processed will not make a queue entry.

7.4.2 Complex Specifier Unit

The Complex Specifier Unit (CSU) processes all specifiers with modes other than short literal or register. It receives parsed instruction stream data and parameters on the specifier control fields. Using a 32-bit, 3-stage pipelined datapath with microcode control, the CSU performs the register and memory data operations required to provide the Ebox with instruction operands. Final operand values are routed to the Ebox memory data registers.

7.4.2.1 CSU Microcode Control

The CSU microsequencer provides microcoded control for the 3-stage pipelined datapath. Under typical operation, a control store address is generated for the 128-entry X 29-bit control store array and a new microword is referenced every cycle. The complete microword depicted in Table 7–23 is issued and forwarded to the subsequent pipeline stages in consecutive cycles in order to control the datapath logic in those stages.

Figure 7–10: Microword Format

```
 2   2   2   2   2   2   2   2   2   1   1   1   1   1   1   1   1   1   1
 8   7   6   5   4   3   2   1   0   9   8   7   6   5   4   3   2   1   0   9   8   7   6      ...      0
+-----------------------------------------------------------------------------------------------------------+
|  ALU.FNC  | ML |         A         B       |    DST          MISC    |    MREQ.FNC    |DEC.NEXT|   NXT.ADDR   |
+-----------------------------------------------------------------------------------------------------------+
```

Table 7–23: Microword Fields

field	description
ALU.FNC	controls the ALU function
ML	selects mem req data length = long or DL
A	IA_bus source
B	IB_bus source
DST	IW_bus destination
MISC	miscellaneous functions
MREQ.FNC	controls memory request function
DEC.NEXT	conditional control of decoder next
NXT.ADDR	full next microaddress field

The 128-entry control store array is arranged as 8 pages of 16 microwords per page. Bits <6:4> of the control store address designate the microcode page, bits <3:0> designate the microword address within a page. The page organization places the microcode corresponding to a unique complex specifier flow within a particular page.

Table 7–24: Microcode Page Allocation

page	description
000	displacement flows, modes=A,C,E
001	displacement deferred flows, modes=B,D,F
010	auto increment flow, mode=8
011	auto increment deferred flow, mode=9
100	register deferred flow, mode=6
101	auto decrement flow, mode=7
110	IPR and utility routines, index flow, mode=4
111	Ebox assists, idle address

The CSU specifier microcode processes VAX defined specifiers 4 and 6-F. These are the operand specifiers that the Ibox defines as complex. Displacement data will be sign extended by the IBU so the CSU can process byte, word and longword displacement specifiers in a longword microcode flow. Displacement deferred specifiers merge together in a similar fashion. Ebox assists are "implicit" operands in some of the VAX opcodes. In order to simplify Ebox microcode to handle instruction execution only, the implicit specifiers are processed up front by the Ibox. These assists appear to the Ebox as typical complex operands. See Section 7.3.2.7 for more information on assists.

7.4.2.2 CSU Pipeline

The 3-stage CSU pipeline operates under microcode control during the S1, S2, and S3 stages of the Ibox pipeline. Control store address generation, control store lookup, and microword issue occurs in the S1 stage. The datapath source busses are driven during the S2 pipeline stage. The S3 stage contains the ALU and write destination bus logic, and memory request logic.

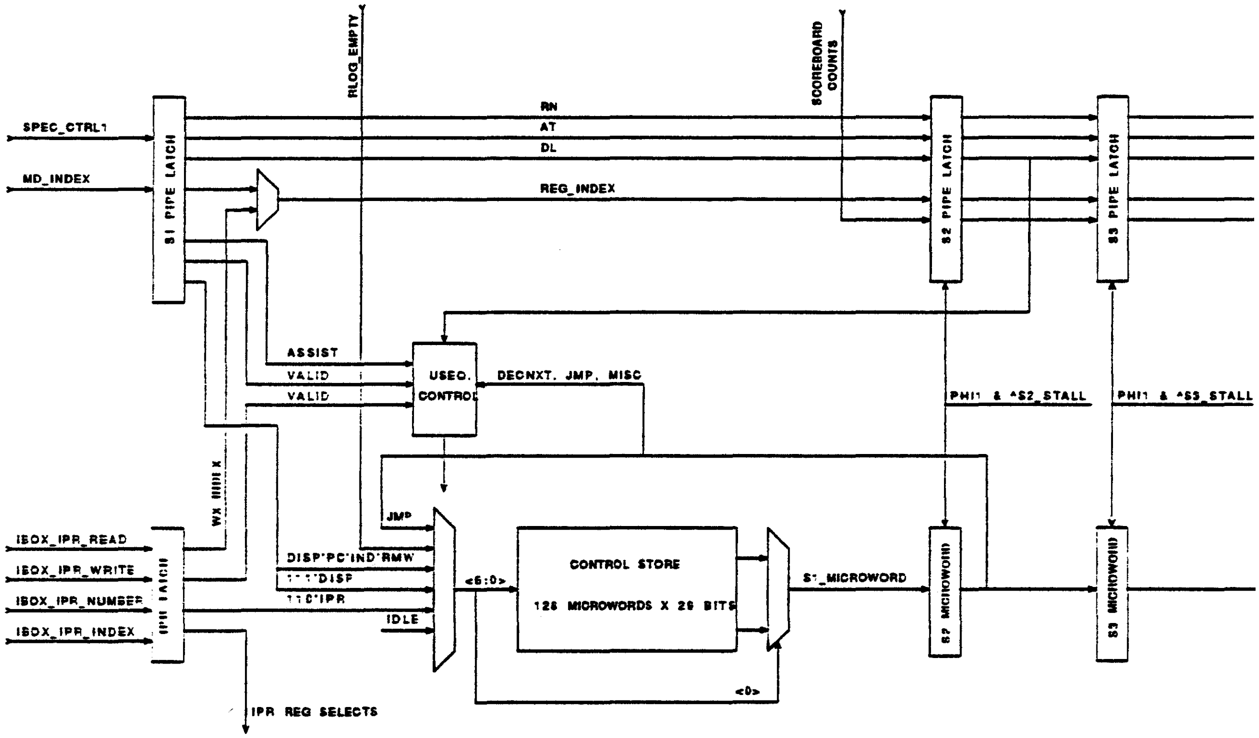
Ordinarily, microwords move through the pipeline synchronously, advancing every cycle. Stalls occur when a resource required for a particular pipeline stage is unavailable. Stalls operate synchronously and transparently to the microcode flow by freezing the sequence and the pipeline, thereby causing the CSU logic to repeat the operation performed in the previous cycle. The stall terminates upon acquisition of the resource which caused the stall and the pipeline flow returns to normal, advancing every cycle.

7.4.2.2.1 S1 Pipeline Stage

The S1 pipe latch, also called the dispatch latch, controls the S1 pipeline logic. The S1 pipe latch is loaded from the parsed instruction stream data and parameters shown in Table 7–25.

S1_RN, **S1_AT**, **S1_DL**, **S1_DISPATCH**, **S1_AT_RMW**, **S1_INDEXED**, **S1_ASSIST**, and **S1_PC_MODE** load directly from the specifier control field and the specifier complex control field as driven by the IBU. The **S1_REG_INDEX** loads from the **MD_INDEX** lines coming from the OQU. The 32-bit **S1_IB_DATA** and **S1_IBOX_PC** are loaded from **SPEC_DATA**<31:0> and **IBU_PC**<31:0> respectively.

Figure 7–11: Complex Specifier Unit Control Path Block Diagram



S1_RXS_SCORE and S1_RXD_SCORE load from the entry in the SBU scoreboard array pointed to by the GPR number of the specifier. S1_RXS_SCORE and S1_RXD_SCORE represent "snapshot" values of the scoreboard, taken when a specifier dispatch enters the S1 pipe latch. The scoreboard updates the value of these entries based on the Ebox retiring source and destination queue entries. See Section 7.4.3 for scoreboard details. The snapshot values decrement in parallel with the SBU values.

Table 7–25: S1 Pipe Latch

Bit Field	Field Name	Description
<3:0>	S1_RN	GPR number from the specifier.
<6:4>	S1_AT	Access Type of the operand associated with the specifier.
<8:7>	S1_DL	Data length of the operand associated with the specifier.
<12:9>	S1_RXS_SCORE	Value of scoreboard source queue counter indexed by GPR number.
<15:13>	S1_RXD_SCORE	Value of scoreboard dest queue counter indexed by GPR number.
<18:16>	S1_DISPATCH	Control store dispatch address.
<19>	S1_AT_RMW	Access Type of operand is R, M or W.
<20>	S1_INDEXED	The base specifier has an index specifier.
<21>	S1_ASSIST	Ebox assist specifier.
<22>	S1_PC_MODE	The specifier uses program counter addressing
<25:23>	S1_REG_INDEX	Value of OQU MD allocation pointer.
<57:26>	S1_IB_DATA	Data for Immediate and displacement mode specifiers.
<89:58>	S1_IBOX_PC	The PC of the next Istream byte following this specifier.
<90>	S1_VALID	S1 pipe latch valid bit.
<91>	S1_JMP_OR_JSB	Indicates whether the instruction was JMP or JSB.

The S1_VALID bit indicates that the S1 pipe latch contains valid dispatch arguments waiting to be serviced. The CSU recognizes the availability of the valid complex dispatch, and performs the control store access. The microword is issued in S1 and loaded into the S2 pipe latch. The CSU sets S1_VALID when a complex specifier is parsed by the IBU and doesn't advance to stage S2 the following cycle. This is a result of a S1_STALL. The S1 logic clears S1_VALID upon successful transition of the S1 microword into the S2 pipe latch. The clear S1_VALID bit indicates the availability of the S1 pipe stage for a new complex specifier dispatch next cycle.

The S1_STALL condition occurs when the S1 context latch cannot be loaded immediately into the S2 pipe latch. This condition may occur during an S2_STALL, when I_IBU%QUAD_FLAG_H<0> is asserted, or a multiple microword flow. S2_STALL indicates that the S2 pipe latch cannot currently advance (see Section 7.4.2.2.2 for more details on the S2_STALL). Naturally this stall ripples back to become an S1_STALL as well because the S1 microword cannot advance into the S2 pipe latch. I_IBU%QUAD_FLAG_H<0> indicates the IBU is waiting for the second longword of a quadword immediate mode specifier. Once the second longword is retired, I_IBU%QUAD_FLAG_H<0> is de-asserted and the CSU is allowed to process the quadword immediate mode specifier. During multiple microword flows, the next control store address is generated from the microword in the S2 pipe latch. Consequently, the S1 pipe latch may accept one dispatch from the IBU which sets S1_VALID. The dispatch in the S1 pipe latch is then in the S1_STALL condition waiting for service.

The IBU uses S1_VALID as part of the parser enable equation. If S1_VALID is clear then the IBU may parse a complex specifier and retire the instruction stream from the PFQ. If S1_VALID is set then if the IBU parses a complex specifier it cannot retire the instruction stream because the S1 pipe latch cannot accept the dispatch. The IBU stalls the parser such that the same specifier is parsed in subsequent cycles.

Typical microcode flows begin at a microcode address determined by a complex specifier dispatch. A DECODER_NEXT directive in the S2 pipe latch tells the microsequencer that the next microcode address is not related to the current flow. If S1_VALID indicates a valid dispatch waiting in the S1 pipe latch and the S2 pipe latch contains a DECODER_NEXT, then the microsequencer selects the S1 pipe latch as the source of the next microaddress. This begins a new microcode flow for the specifier being dispatched. The microcode sequences through a flow using microaddress jumps. A jump selects the NXT_ADDR<6:0> field of the microword in the S2 pipe latch directly for the next microword address. The final microword of each flow contains a DECODER_NEXT which once again requests a new dispatch address.

Requests for IPR references which are detailed in Section 7.4.2.8 must guarantee that the CSU is idle. Thus, whenever the S1 logic detects an IPR read strobe from the Ebox, then the next microaddress is selected by the IPR number. The request immediately dispatches to the utility microcode page.

The unwind_mispredict routine is selected when the Ebox signals a branch mispredicted. The RLOG unwinds restoring the GPRs until the RLOG is empty, then the Ibox is restarted.

The CSU dispatches to the common entry point for the single microword index routine when the dispatch number of a specifier indicates that it is an index. The index register is read from the Ebox and shifted by length = DL.

The microaddress control selects the IDLE address when no valid dispatch or utility dispatch awaits processing. The IDLE microword simply jumps to its own address and executes the DECODER_NEXT directive, awaiting a valid dispatch.

In addition to the standard DECODER_NEXT directive, the microcode and next address logic supports a conditional DECODER_NEXT. The DECODER_NEXT_IF_BWL performs a standard DECODER_NEXT if the data length associated with the specifier is byte, word, or longword. For quadword data length the next address logic performs a microaddress jump.

The microcode and next address logic supports one conditional jump. The BRANCH_IF_RLOG_EMPTY directive causes the next microaddress logic to perform a standard jump, but in addition the logic OR function of a 1 and the next microaddress bit <0> is performed if the RLOG is empty. The RLOG unwind microcode uses this conditional jump feature. A single microword jumps to itself as long as the RLOG still has valid entries. When the RLOG empties, the microword conditionally jumps out of the loop. See Section 7.4.2.3 for RLOG details.

The S1 logic uses a five-input multiplexer to select the source of the next control store address. Both the complex specifier multiplexer input and Ebox assist multiplexer input use data from the S1 pipe latch to form the next address. The IPR multiplexer input uses the latched IPR number from the Ebox, to select which IPR type field will be used to form the next address. The next address field from the S2 microword enters another multiplexer input in order to perform the microaddress jump. The final multiplexer input is the idle address. Next address generation is summarized by Table 7–26.

Table 7–26: Next Address Generation Fields

bit		
field	**field name**	**description**
		Specifier Dispatch
<0>		forced to 0
<1>	S1_INDEXED	index specifier
<2>	S1_PC_MODE	base register is the PC
<3>	S1_AT_RMW	access type = read,modify, or write
<6:4>	S1_DISPATCH	S1_DISPATCH<2:0> field from the IBU
		Assist Dispatch
<0>		forced to 0
<3:1>	S1_DISPATCH<2:0>	assist type
<6:4>		forced to 111, assist page number
		IPR and Utility Dispatch
<0>		forced to 0
<3:1>	000	index routine
<3:1>	001	IPR unwind RLOG read back-up PC
<3:1>	010	E%BRANCH_MISPREDICT_L
<3:1>	011	IPR read
<6:4>		forced to 110, IPR/utility page number
		Idle Dispatch
<6:0>		forced to 1111111, idle address
		Next Address
<0>	NXT_ADDR	next address field from the S2_MICROWORD. For conditional jump OR in 1 if RLOG is empty
<6:1>	NXT_ADDR	next address field from the S2_MICROWORD

7.4.2.2.2 S2 Pipeline Stage

The S2 pipe latch controls the S2 pipeline datapath. Each cycle, the S2 pipe latch attempts to load a microword and specifier specific parameters from the instruction stream. The S2 pipe latch is shown in Table 7–27.

Table 7–27: S2 Pipe Latch

Bit Field	Field Name	Description
<3:0>	S2_RN	GPR number from the specifier.
<6:4>	S2_AT	Access Type of the operand associated with the specifier.
<8:7>	S2_DL	Data length of the operand associated with the specifier.
<11:9>	S2_REG_INDEX	Current value of S2 MD allocation pointer or WX index.
<15:12>	S2_RXS_SCORE	Value of scoreboard source queue counter indexed by GPR number.
<18:16>	S2_RXD_SCORE	Value of scoreboard dest queue counter indexed by GPR number.
<47:19>	S2_MICROWORD	The microword issued in S1.
<48>	S2_NEW_FLOW	Indicates the first microword of a flow.
<49>	S2_JSB_OR_JMP	Indicates whether the instruction was JMP or JSB.

S2_RN, S2_AT, S2_DL, S2_JSB_OR_JMP, S2_RXS_SCORE, and, S2_RXD_SCORE load directly from the S1 pipe latch. S2_RXS_SCORE and S2_RXD_SCORE decrement in parallel with their corresponding SBU values. S2_REG_INDEX typically loads directly from S1_REG_INDEX, however, if the dispatch is for an IPR read, it loads a copy of WX_INDEX from the Ebox.

The S2_MICROWORD field of the S2 pipe latch updates from the microword issued by the S1 pipe stage. During an initial specifier dispatch, all of the S2 pipe latch updates. Bits <48:19> of the latch update every cycle, assuming no stalls. However, bits <49,18:0> of the latch remain constant throughout the context of one specifier flow, except for local scoreboard decrements of S2_RXS_SCORE and S2_RXD_SCORE. This part of the S2 pipe latch does not load again until another dispatch occurs. This allows for multiple microword flows within the context of a given specifier.

S2_NEW_FLOW indicates that contents of the S2 pipe latch represents the first microword of a new dispatch. In other words, the microword address for the microword in S2 was generated in any manner other than a microaddress jump. This pipe bit aids the S3 stage in loading the specifier context portion of the S3 latch. See section Section 7.4.2.2.3 for details.

The S2 datapath contains the CSU register set and constant generator. The CSU ALU source busses, the IA_bus and IB_bus, are controlled by the microcode /A and /B fields to drive the source busses in the S2 pipeline stage. The CSU microcode may also requests an Ebox GPR to source the IA_bus by providing the I%IBOX_IA_ADDR_H<3:0> from the S2_RN field of the S2 pipe latch. The Ebox register read is strobed with I%IBOX_IA_READ_H. The Ebox returns GPR data later that cycle on the E%IBOX_IA_BUS_H<31:0> lines. This provides a path for the CSU to obtain the base specifier register of the operand currently being processed. When the S2_microword is sourcing a GPR which is identical to the S3_microword destination register, the IW_BUS will be driven onto the source bus, bypassing the GPR read.

Table 7–28: CSU Registers

Register Name	Available On	Written From	Description
T0<>	IA,IB	IW	temporary register

Table 7–28 (Cont.): CSU Registers

Register Name	Available On	Written From	Description
IB_DATA	IA,IB	SPEC_DATA	immediate and displacement data
RX	IA	IW	base specifier register
IMD	IA	MD	Ibox memory data
MD	-	IW	Ebox memory data register
WX	-	IW	Ebox working register
KDL	IB	-	1 for DL=Byte, 2 for Word, 4 for LONG, 8 for QUAD
IDL	IB	-	1 for DL=Byte, 2 for Word, 4 for LONG, 8 for QUAD
K4	IB	-	Constant 4
K12	IB	-	Constant 12
RLOG_RX	IA	IW_BUS	Register pointed to by top of RLOG
RLOG_KDL	IB	-	Same as KDL except using DL from top of RLOG stack
IBOX_PC	IA	IBU_PC	PC of instruction byte following last byte in specifier

T0 is a temporary register for microcode use. IB_DATA and IBOX_PC are the S2 pipeline copies of S1_IB_DATA and S1_IBOX_PC respectively. IB_DATA and IBOX_PC are loaded along with the S2_PIPE_LATCH<18:0> on the first microword of a dispatch. Then the CSU microcode maintains control of these registers throughout the context of a given specifier flow.

RX refers to the Ebox GPR register indexed by S2_RN. RLOG_RX refers to the Ebox GPR register indexed by the RLOG_RN. See Section 7.4.2.3 for more details. MD addresses the Ebox MD register indexed by S2_REG_INDEX. WX points to the Ebox working register also indexed by S2_REG_INDEX. K4 and K12 are constants. KDL is a constant based on S2_DL. The value of the constant is 1 for DL=0 (byte), 2 for DL=1 (word), 4 for DL=2 (longword), and 8 for DL=3 (quadword). IDL is a constant based on S2_DL for immediate mode specifier with access type A or V. IDL differs from KDL in the fact that the constant value is 4 for DL=3 (quadword). RLOG_KDL is a constant similar to KDL, but based on RLOG_DL. See Section 7.4.2.3 for more details.

For a majority of memory requests started by the CSU microcode, the Ibox memory data returns to the IMD register. The Mbox drives M%IBOX_DATA_L when M%MD_BUS_H<31:0> contains valid data from a specifier memory request. The IMD has a signal IMD_VALID associated with it. Each time the CSU microcode initiates a memory request IMD_VALID is set. Each time memory data returns to IMD, IMD_VALID is reset.

When M%MME_FAULT_H or M%HARD_ERR_H is asserted by the Mbox along with M%IBOX_DATA_L, this indicates that Ibox data on M%MD_BUS_H<63:0> is invalid and that the corresponding reference was associated with either a memory management exception or a hard error condition. In both cases the CSU continues to process the specifier, but sets flags indicating the IMD contains invalid data. The flags are reset at the end of each specifier flow. They are forwarded to stage S3 whenever the IMD is selected to source the IA_bus. They are called I%FORCE_MME_FAULT_H and I%FORCE_HARD_FAULT_H. When set they indicate to the Ebox and Mbox that the associated register write or Ibox reference should be forced to "look" like a memory management fault or a hardware fault from the Ibox point of view.

The S2 pipeline stage stalls for three reasons: GPR destination queue stall (RXD_STALL), Ibox memory data stall (IMD_STALL) and S3_STALL. RXD_STALL occurs when the CSU microcode attempts a read of a GPR for which there exist outstanding writes in the Ebox destination queue. The S2 pipeline logic detects RXD_STALL when S2_RXD_SCORE does not equal 0, and the S2_MICROWORD attempts to read the GPR from the Ebox indexed by S2_RN. The stall breaks when the Ebox retires a destination queue entry that causes both the SBU counter and the snapshot S2_RXD_SCORE to decrement. Multiple destination queue entries may have to be retired, causing multiple decrements, before S2_RXD_SCORE equals 0.

IMD_STALL occurs when the S2_MICROWORD attempts to read the IMD when IMD_VALID is set. This condition implies that a memory request was initiated by CSU microcode which set IMD_VALID, but memory data which resets the signal has not yet been returned. IMD_STALL can only happen in the context of one complex specifier flow when the Ibox requests then waits for memory data to be returned to IMD.

S2_STALLS block the S2 pipeline latch update, causing the S2 stage to execute the same stalled MICROWORD until the stall breaks. If an S2 stall occurs, not resulting from a S3 stall, the S3 pipeline latch continues to updates; however, NOPs are fed into the S3 pipeline latch while the S2 stall is in progress. When the stall breaks, the pipeline latches resume normal operation.

7.4.2.2.3 S3 Pipeline Stage

The S3 pipe latch controls the S3 pipeline datapath. Each cycle, the S3 pipe latch attempts to load a microword and the specifier-specific parameters from the instruction stream. The S3 pipe latch is shown in Table 7–29.

Table 7–29: S3 Pipe Latch

Bit Field	Field Name	Description
<3:0>	S3_RN	GPR number from the specifier.
<6:4>	S3_AT	Access Type of the operand associated with the specifier.
<8:7>	S3_DL	Data length of the operand associated with the specifier.
<11:9>	S3_REG_INDEX	Current value of S3 MD allocation pointer or WX index.
<15:12>	S3_RXS_SCORE	Value of scoreboard source queue counter indexed by GPR number.
<46:16>	S3_MICROWORD	The microword issued in S1.
<47>	S3_JSB_OR_JMP	Indicates whether the instruction was JMP or JSB.

S3_RN, S3_AT, S3_DL, S3_REG_INDEX, S3_JSB_OR_JMP, and S3_RXS_SCORE load directly from the S2 pipe latch. S3_RXS_SCORE decrements in parallel with its corresponding SBU value. When S3 logic initiates a memory reference with an MD destination, S3_REG_INDEX specifies the index into the MD register array for the memory data write. Such memory requests cause MD_INDEX to increment modulo the size of the MD register file, so that the data for quadword operands, which require two memory requests, occupy successive MD registers.

The S3_MICROWORD field of the S3 pipe latch updates from the S2_MICROWORD. During the first instruction of a specifier dispatch flow, as indicated by the contents of S2_NEW_FLOW, all of the S3 pipe latch updates. The microword field in bits <46:16> continues to update every cycle, loading the new microword from S2. However, bits <47,15:0> of the latch remain constant throughout the context of one specifier flow, except for local scoreboard decrements of **S2_RXS_SCORE**, and local increments of **S3_REG_INDEX**. This part of the S3 pipe latch does not reload until another dispatch occurs, allowing for multiple microword flows within the context of a given specifier.

The S3 datapath contains the CSU ALU and register write logic. The ALU maintains 32-bit input latches which load the IA_BUS and IB_BUS during an S3 pipe latch update. Under control of the microcode /ALU.FNC field the ALU performs 32-bit add, subtract, pass, and left bit-shift equal to **S2_DL**. The destination bus, IW_bus, provides the path to write the ALU results to one of the CSU registers under control of the microcode /DST field. The IW_BUS_bus can also be selected to write to the Ebox GPR, MD, and working (WX) registers. The **I%IBOX_IW_BUS_H**<31:0> lines are driven from the ALU output, and the **S3_RN** field of the S3 pipe latch provide **I%IBOX_IW_ADDR_H**<4:0> as an index into the GPR array. MD and WX writes both use the **S3_REG_INDEX** field of the S3 pipe latch to provide **I%IBOX_IW_ADDR_H**<4:0> as an index into the Ebox register array. The Ebox register write is strobed with **I%IBOX_IW_WRITE_H**

The S3 stage logic initiates CSU memory requests based on the **S3_MICROWORD**. Along with a memory request command, the full 32-bit address is sent to the Mbox on the **I%IBOX_ADDR_H**<31:0> lines. These lines may be sourced from either the IA_BUS or IW_BUS, under the **S3_MICROWORD** /MREQ field control. If microcode selects the IA_BUS for memory request address, the S3 pipe latch for the IA_BUS sources the address. The S3 logic also forwards **VIC_REQ** from VIC Istream requests to the Mbox when there are no specifier memory requests in the **S3_MICROWORD**. In this case, the **I%IBOX_ADDR_H**<31:0> is sourced by **VIC_REQ_ADDR** from the VIC.

The following control signals accompany **I%IBOX_ADDR_H**<31:0>. **I%IBOX_CMD_L**<4:0> indicates reference type to the Mbox. See Section 12.3.1 in Chapter 12 for valid values. **I%IBOX_TAG_L**<4:0> contains the Ebox register file destination of a memory request, a copy of **S3_REG_INDEX**. **I%IBOX_AT_L**<1:0> and **I%IBOX_DL_L**<1:0> provide the Mbox with the access type and data length. **I%IBOX_AT_L**<1:0> is either a copy of **S3_AT** or forced to read or write depending on control of the microcode /MREQ field. **I%IBOX_DL_L**<1:0> is either a copy of **S3_DL** or forced to longword depending on control of the microcode /ML field. **I%IBOX_REF_DEST_L**<1:0> specifies the destination for memory request data. **I%IBOX_REF_DEST_L**<1> indicates that the Ebox MD registers are the destination. **I%IBOX_REF_DEST_L**<0> indicates that the Mbox IMD register is the destination. This field is decoded from the **S3_MICROWORD** memory field. The **I%SPEC_REQ_H** strobe is asserted for CSU specifier memory requests. The **I%IREF_REQ_H** strobe is asserted for VIC Istream memory requests.

For JMP, JSB, and certain Ebox assists, the S3 logic sends requests to the BPU to load a new PC. The PC value may be sourced from either the **I%IBOX_IW_BUS_H**<31:0> or **M%MD_BUS_H**<31:0> under **S3_MICROWORD** /MISC field control, as indicated by **LD_PC_WBUS** or **LD_PC_MD** respectively.

The S3 pipeline stage stalls for three reasons: GPR source queue stall (**RXS_STALL**), memory request stall (**MRQ_STALL**), and (**RLOG_STALL**). **RXS_STALL** occurs when the CSU microcode attempts to write a GPR destination for which there exist outstanding read in the Ebox source queue. The S3 pipeline logic detects **RXS_STALL** when **S3_RXS_SCORE** does not equal 0, and the **S3_MICROWORD** attempts to write the GPR in the Ebox indexed by **S3_RN**. The stall breaks when the Ebox retires a source queue entry that causes both the SBU counter and the snapshot **S3_RXS_SCORE** to decrement. Multiple destination queue entries may have to be retired, causing multiple decrements, before **S3_RXS_SCORE** equals 0.

RLOG_STALL occurs when **RLOG_FULL** is asserted and the microword in the S3 pipe requests a GPR write. The stall effect is exactly the same as **RXS_STALL**. The stall breaks when the Ebox retires an instruction which in turn relinquishes RLOG resources.

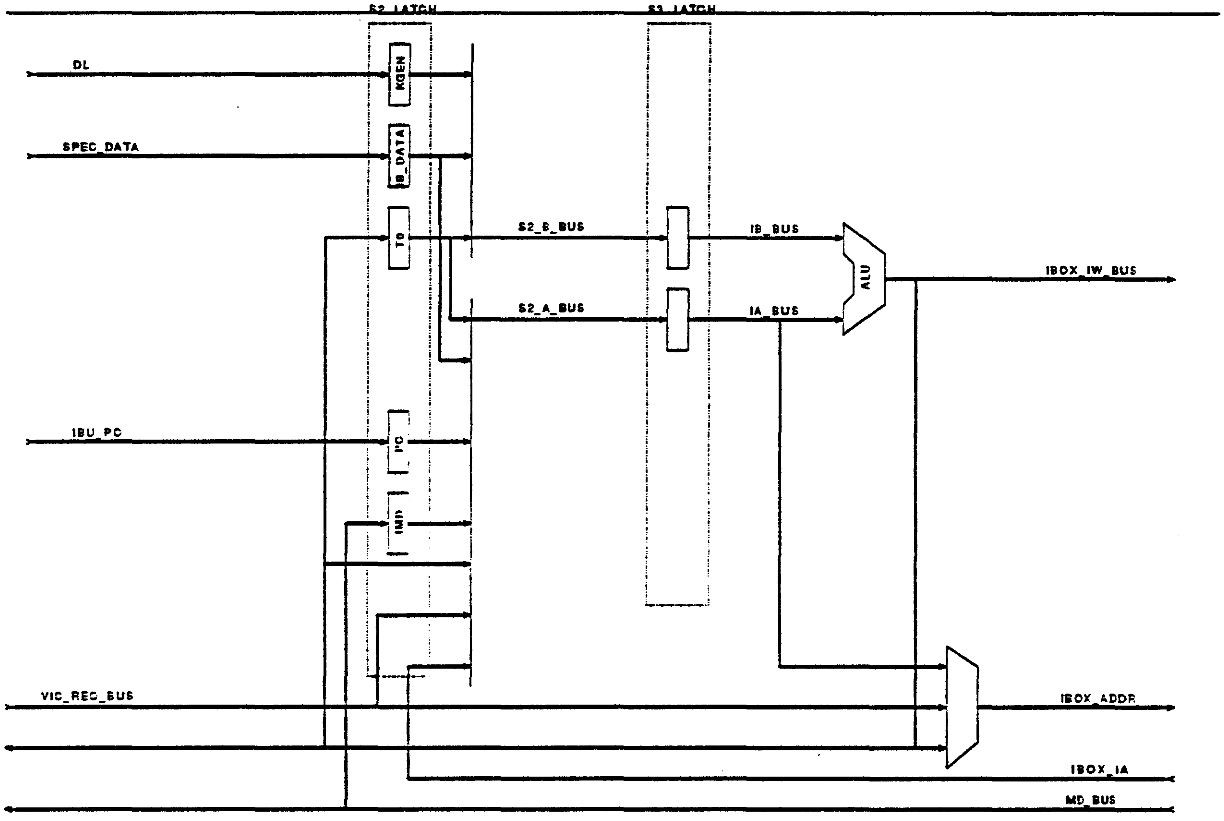
MRQ_STALL occurs when the S3_MICROWORD attempts a memory request but the M%SPEC_Q_FULL_H signal from the Mbox indicates that the request cannot be accepted.

S3_STALLS block the S3 pipeline latch update, causing the S3 stage to execute the same stalled MICROWORD until the stall breaks. S3_STALLS also back-stall the S2 stage, in effect causing S2_STALL which blocks the S2 pipeline latch update. Both pipeline stages execute their respective stalled microwords until the stall condition breaks, allowing successful completion of the microword. The pipeline latches then continue to update as usual.

RXS_STALL does not block the initiation of a memory request by the S3_MICROWORD. In other words, if the S3_MICROWORD indicates a memory request operation and no MRQ_STALL or RLOG_STALL exists, the request is initiated regardless of RXS_STALL. This somewhat de-coupled operation of the S3_STALLS breaks possible macroinstruction deadlocks due to the R0 (R0)+ case. While processing the specifier (R0)+ the CSU microcode performs a write to the GPR R0. A RXS_STALL will hold until the Ebox retires the first source, R0. The Ebox must retire two source operands at a time, and therefore cannot retire the R0 specifier until the MD for the second specifier is valid.

The converse case, whether MRQ_STALL blocks a register write, is not an architectural or performance issue. This implementation blocks register writes during an MRQ_STALL.

Figure 7–12: Complex Specifier Unit Data Path Block Diagram



7.4.2.3 RLOG

The register log or RLOG allows the Ibox to restore the state of the GPRs under certain exception conditions. Because of the pipeline organization, the Ibox works on macroinstructions ahead of the Ebox execution. Any or all of six possible operand specifiers for any distinct macroinstruction may be auto-increment or auto-decrement mode, which by definition modify the GPRs. The Ibox must log all modifications to the GPRs for these operand specifiers and keep the log until the Ebox has retired the associated instruction. If the instruction stream gets redirected due to a branch or exception, then the Ibox uses the RLOG to restore the GPR registers to the condition expected at the time of the redirection.

The RLOG is an 8-entry circular queue with read and write pointers. Each entry is composed of 7 bits, 4 bits contain the GPR number, 2 bits specify DL, and 1 bit indicates auto-increment or auto-decrement.

Elements are added to the RLOG under control of the S3_MICROWORD /DST field. When the microword specifies a register log operation, then S3_RN, S3_DL, and the encoded /ALU.FNC are entered in the RLOG entry pointed to by the write pointer. The write pointer is then incremented modulo 8. If the RLOG write pointer reaches the state in which another increment causes the write pointer to equal the read pointer, then the RLOG is full. The RLOG full condition may cause an RLOG_STALL as described in Section 7.4.2.2.3.

The RLOG only contains specifier state for macroinstructions which the Ebox has not executed. When the Ebox retires a macroinstruction, the RLOG discards RLOG entries associated with that macroinstruction, by advancing the RLOG read pointer. The **RLOG_BASE_POINTER** and **RLOG_BASE_QUEUE** provide the means for read pointer advancement.

The **RLOG_BASE_POINTER** increments anytime a valid auto-increment address mode specifier, auto-decrement address mode specifier, auto-increment assist, or auto-decrement assist appears on **SPEC_CTRL**. In effect, the **RLOG_BASE_POINTER** allocates RLOG spaces for the CSU to make subsequent entries. The **RLOG_BASE_POINTER** is loaded into the 6-entry **RLOG_BASE_QUEUE** each time a new PC is loaded into the **PC_QUEUE**. The **RLOG_BASE_QUEUE** thus maintains an RLOG read pointer for every PC in the **PC_QUEUE**. The **RLOG_BASE_QUEUE** and the **PC_QUEUE** both retire entries when the Ebox asserts **E%RETIRE_INSTR_L** indicating that it has retired a macroinstruction. The RLOG read pointer loads the value of the next **RLOG_BASE_QUEUE** entry at this time.

The CSU microcode controls the RLOG unwind procedure. RLOG unwind consists of repeatedly executing a microword that updates the GPR registers based on indirect references to RLOG_RN, RLOG_DL, and RLOG_FUNC. The RLOG supplies the values for the indirect references from the entry pointed to by the read pointer. This entry is retired by incrementing the read pointer. The RLOG retires successive entries until the read pointer is equal to the write pointer, then the RLOG is empty. At this point the unwind procedure completes and the RLOG is flushed by resetting the RLOG read and write pointers, the **RLOG_BASE_POINTER**, and the **RLOG_BASE_QUEUE** read and write pointers. If the RLOG is empty when the microcode initiates an unwind, 0 will be added to whatever GPR is pointed to by the read pointers.

7.4.2.4 Branch Mispredict effects

When the Ebox asserts **E%BRANCH_MISPREDICT_L**, the NOP microword is forced into the S3 pipeline stage, the S1 pipe latch valid bit is cleared, and the next microaddress logic selects the MISPREDICT.UNWIND utility routine address. The microcode at this location unwinds the RLOG and then restarts the Ibox. If the RLOG is empty when the microcode initiates an unwind, 0 will be added to whatever GPR is pointed to by the read pointers. Note that the RLOG is NOT flushed on the assertion of **E%BRANCH_MISPREDICT_L**. It needs to remain intact to be unwound by CSU microcode.

IMD_VALID is reset upon the assertion of **E%BRANCH_MISPREDICT_L**.

7.4.2.5 E%STOP_IBOX Effects

When the Ebox asserts **E%STOP_IBOX_H**, the microsequencer jams the CSU to the idle state, except in the case when the CSU is in the middle of IPR transaction unwind RLOG/read back-up PC. In this situation, the RLOG will unwind until completion, and the read of the back-up PC will be disabled. The CSU is put into the idle state by forcing NOP microwords into the S2 and S3 pipeline stages, clearing the S1 pipe latch valid bit, and selecting the IDLE microaddress.

7.4.2.6 RSVD_ADDR_FAULT effects

When **I%RSVD_ADDR_FAULT_H** is asserted for a complex specifier the S1 pipe latch valid bit is cleared. If there isn't a S1 stall the NOP microword is forced into the S2 pipeline stage. Complex specifiers already in the CSU pipeline when **I%RSVD_ADDR_FAULT_H** is asserted are allowed to finish processing.

7.4.2.7 CSU Microcode Restrictions

The CSU microcode must guarantee, for all auto-increment, auto-increment deferred, and auto-decrement specifier microcode flows, that any specifier memory requests destined for the MD is issued before or during the microword that modifies the GPR. Otherwise, it is possible for the CSU to infinitely stall due to an **RXS_STALL**. This is evident in the case ADDL2 R0,@(R0)+ where the Ebox must retire two source operands, and therefore cannot retire the R0 specifier until the MD for the second specifier is valid. The CSU microcode must also guarantee, for all auto-increment, auto-increment deferred, and auto-decrement specifier microcode flows, that the microword which initiates the memory request destined for the MD must have the misc field stall_if_rlog_full if the following microword modifies the gpr.

The CSU microcode must guarantee, for all auto-increment, auto-increment deferred, auto-decrement and auto-decrement deferred specifier microcode flows with access type AV, that the microword which writes the MD is immediately followed by the microword that modifies the gpr. This, in conjunction with an EBOX microcode restriction, is necessary in order to prevent an infinite RXS stall from occurring.

The CSU microcode must guarantee that memory requests which specify the Ibox IMD as the data destination, are used only for deferred operand evaluation. For a microword with a [IMD] source, the previous microword must initiate the memory request with destination IMD and must not perform a GPR write and not have the misc field stall_if_rlog_full. All this is necessary to protect the use of an unconditional MD latch in the CSU datapath.

7.4.2.8 Ibox IPR Transactions

The Ebox microcode communicates with the Ibox in part through internal processor registers (IPRs). The IPR reads are handled by CSU microcode. The IPR write control is distributed, however the description is included here for completeness.

Ebox microcode conventions guarantee that the Ibox is idle before initiating Ibox IPR transactions. This is accomplished either by the knowledge that the current Ebox microcode flow takes place in a macroinstruction with an drain Ibox assist or by asserting an explicit **E%STOP_IBOX_H** command. The only exception involve the issuing of an IPR transaction when the CSU is involved in an RLOG unwind operation. In this case the unwind finishes in the CSU, then the CSU processes the latched IPR command. If the RLOG is empty when the microcode initiates an unwind, 0 will be added to whatever GPR is pointed to by the read pointers.

MICROCODE RESTRICTION

E%IBOX_LOAD_PC_L and **E%IBOX_IPR_WRITE_H** must not occur in the same cycle.

7.4.2.8.1 IPR Reads

The Ebox signifies an IPR read by asserting the **E%IBOX_IPR_READ_H** strobe, the **E%IBOX_IPR_TAG_H**<2:0>, and the **E%IBOX_IPR_NUM_H**<3:0>. This information is latched in the S1 logic stage, and an IPR request flag is posted. The S1 next address logic responds by creating an IPR dispatch to an IPR microaddress in the utility page of microcode, and by clearing the IPR request flag. All Ibox logic blocks associated with IPR reads examine the **E%IBOX_IPR_TAG_H**<2:0>. If the IPR source is within a section, that section prepares to drive the IPR read data onto the **VIC_REQ_ADDR**. The microcode at the common IPR routine reads the **VIC_REQ_ADDR**, passes the value through the ALU, and writes the data to an Ebox working register located at the

E%IBOX_IPR_NUM_H<3:0> offset in the register array. The **VIC_REQ_ADDR** is used for IPR read data source simply because it is a convenient 32-bit bus that runs through the entire section.

7.4.2.8.2 IPR Writes

The Ebox signifies an IPR write by asserting the **E%IBOX_IPR_WRITE_H** strobe and the **E%IBOX_IPR_TAG_H**<2:0>. All Ibox logic blocks associated with IPR writes examine the **E%IBOX_IPR_TAG_H**<2:0>. If the IPR destination is within a section, that section prepares to accept the IPR write data from the **M%MD_BUS_H**<63:0>. The Mbox drives the **M%MD_BUS_H**<63:0> with IPR data and asserts **M%IBOX_IPR_WR_H** to complete the transaction.

7.4.3 Scoreboard Unit

The Scoreboard Unit (SBU) keeps track of the number of outstanding references to GPRs in the source and destination queues. The SBU contains two arrays of 15 counters: the **RXS_ARRAY** for the source queue and the **RXD_ARRAY** for the destination queue. The counters in the arrays map one-to-one with the GPRs. There is no scoreboard counter corresponding to GPR 15, the PC, because RMODE operations to the PC are unpredictable. The maximum number of outstanding operand references determines the maximum count value for the counters. This value is based on the length of the source and destination queues. The **RXS_ARRAY** counts up to 12 and the **RXD_ARRAY** counts up to 6.

Each time valid register mode source specifiers appear on **SPEC_CTRL**<13:0>, the **RXS_ARRAY** counters that correspond with those registers are incremented. At the same time, the OQU inserts entries pointing to these registers in the source queue. In other words, for each register mode source queue entry, there is a corresponding **RXS_ARRAY** counter increment. This implies a maximum of 2 counters incrementing each cycle when a quadword register mode source operand is parsed. Each counter may only be incremented by 1. When the Ebox removes the source queue entries, the counters are decremented. The Ebox removes up to 2 register mode source queue entries per cycle as indicated on **E%SQ_RETIRE_RMODE_H**<1:0>. The GPR numbers for these registers are provided by the Ebox on **E%SQ_RETIRE_RN1_H**<3:0> and **E%SQ_RETIRE_RN2_H**<3:0>. A maximum of 2 counters may decrement each cycle, or any one counter may be decremented by up to 2, if both register mode entries being retired point to the same base register.

In a similar fashion, when a new register mode destination specifier appears on **SPEC_CTRL**<13:0>, the **RXD_ARRAY** counter that corresponds to that register is incremented. A maximum of 2 counters increment in one cycle for a quadword register mode destination operand. When the Ebox removes a destination queue entry, the counter is decremented. The Ebox indicates removal of a register mode destination queue entry on **E%DQ_RETIRE_RMODE_H**. The GPR number for the register is provided by the Ebox on **E%DQ_RETIRE_RN_H**<3:0>.

Whenever a complex specifier is parsed, the GPR associated with that specifier is used as an index into the source and destination scoreboard arrays and snapshots of both scoreboard counter values are passed to the CSU on **RXS_SCORE**<3:0> and **RXD_SCORE**<2:0>. The CSU stalls if it needs to read a GPR for which the destination scoreboard counter value is non-zero. A non-zero destination counter indicates that there is at least one pointer to that register in the destination queue. This means that there is a future Ebox write to that register and that its current value is invalid. The CSU also stalls if it needs to write a GPR for which the source scoreboard counter value is non-zero. A non-zero source scoreboard value indicates that there is at least one pointer to that register in the source queue. This means that there is a future Ebox read to that register and its contents must not be modified. For both scoreboards, the copies in the CSU pipe are locally decremented on assertion of the retire signals from the Ebox.

7.4.3.1 E%STOP_IBOX and Branch Mispredict PC Load Effects

Whenever a branch mispredict PC load occurs, or the Ebox issues a E%STOP_IBOX_H, all scoreboard array counters are cleared.

7.5 Branch Prediction

The Branch Prediction Unit (BPU) monitors each instruction opcode as it is parsed, looking for a branch opcode. Upon identification of a branch opcode, the BPU predicts whether or not the branch will be taken. If the BPU predicts the branch will be taken, it adds the sign extended branch displacement to the current PC and broadcasts the resulting new PC to the rest of the Ibox on the NEW_PC lines.

7.5.1 Branch Prediction Unit

7.5.1.1 The Branch Prediction Algorithm

The BPU uses a "Branch History" algorithm for predicting branches. The basic premise behind this algorithm is that branch behavior tends to be patterned. If one looks in a program at one particular branch instruction, and traces over time that instruction's history of branch taken vs. branch not taken, in most cases a pattern develops. Branch instructions that have a past history of branching seem to maintain that history and are more likely to branch than not branch in the future. Branch instructions which follow a pattern such as branch, no branch, branch, no branch etc., are likely to maintain that pattern. Branch history algorithms for branch prediction attempt to take advantage of this "branch inertia".

The NVAX branch prediction unit uses a table of branch histories and a prediction algorithm based on the past history of the branch. When the BPU encounters a conditional branch opcode, a subset of the opcode PC bits is used to access the branch history table. The output from the table is a 4 bit field containing the branch history information for the branch. From these 4 history bits, a new prediction is calculated indicating the expected branch path.

Many different opcode PCs map to each entry of the branch table because only a subset of the PC bits form the index. When a branch opcode changes outside of the index region, the history table entry that it indexes may be based on a different branch opcode. The branch table relies on the principle of locality, and assumes that, having switched PCs, the current process operates within a small region for a period of time. This allows the branch history table to generate pertinent history relating to the new PC within a few branches.

The branch history information consists of a string of 1's and 0's indicating what that branch did the last four times it was seen. For example, 1100, read from right to left, indicates that the last time this branch was seen it did not branch. Neither did it branch the time before that. But then it branched the two previous times. The prediction bit is the result of passing the history bits that were stored through logic which predicts the direction a branch will go given the history of its last four branches.

The prediction algorithm is accessible via IPR for software programming and testability reasons. After power-up, the Ebox microcode initializes the branch prediction algorithm segment of the BPCR register with an algorithm which is the result of extensive simulation and statistics gathering. While it would be possible to create a program for which this prediction logic is wrong all the time, on the average it does very well. This algorithm is shown in Table 7–30. The BPCR is discussed in greater detail in Section 7.5.1.8.

7.5.1.2 The Branch History Table

The 512 entries in the branch table are indexed by the opcode PC<8:0>. Each branch table entry, as depicted in Figure 7–13, contains the previous four branch history bits for branch opcodes at this index. The Ebox asserts E%FLUSH_BPT_H under microcode control during process context switches. This signal resets all branch table entries to a neutral value: history = 0100. This will result in a next prediction of 0.

MICROCODE RESTRICTION

E%FLUSH_BPT_H may only occur while the Ibox is stopped. E%FLUSH_BPT_H must be asserted before the first branch is executed.

Figure 7–13: Branch Table Entry Format

```
  3   2   1   0
+---------------+
|  History      |
+---+---+---+---+
              (most recent)
```

7.5.1.3 Branch Prediction Sequence

When the BPU encounters a conditional branch opcode it reads the branch table entry indexed by PC<8:0>. If the prediction logic indicates the branch taken, then the BPU sign extends and adds the branch displacement supplied by the IBU to the current PC, and broadcasts the result to the Ibox on the NEW_PC lines. If the prediction bit indicates not to expect a branch taken, then the current PC in the Ibox remains unaffected.

The alternate PC in both cases (current PC in predicted taken case, and branch PC in predicted not taken case) is retained in the BPU until the Ebox retires the conditional branch. When the Ebox retires a conditional branch, it indicates the actual direction of the branch. The BPU uses the alternate PC to redirect the Ibox in the case of an incorrect prediction. Section 7.5.1.7 has more details on mispredicted branches.

The branch table is written with new history each time a conditional branch is encountered. Once a prediction is made, the oldest of the branch history bits is discarded. The remaining 3 branch history bits and the new predicted history bit are written back to the table at the same branch PC index. When the Ebox retires a branch queue entry for a conditional branch, if there was not a mispredict, the new entry is unaffected and the BPU is ready to process a new conditional branch. If a mispredict is signaled, the same branch table entry is rewritten, this time the least significant history bit receives the complement of the predicted direction, reflecting the true direction of the branch.

The branch prediction logic is based on the contents of the BPCR register, described in Section 7.5.1.8. After power-up, as part of the initialization sequence, the Ebox microcode initializes the BPCR to ECC8 (HEX) which implements the truth table in Table 7–30.

MICROCODE RESTRICTION

> An IPR write to the BPCR register in the BPU is required after power-up to load the branch prediction algorithm.

Table 7–30: Branch Prediction Logic

Branch History	Prediction for Next Branch
0 0 0 0	Not taken
0 0 0 1	Taken
0 0 1 0	Not Taken
0 0 1 1	Taken
0 1 0 0	Not Taken
0 1 0 1	Not Taken
0 1 1 0	Taken
0 1 1 1	Taken
1 0 0 0	Not Taken
1 0 0 1	Taken
1 0 1 0	Taken
1 0 1 1	Taken
1 1 0 0	Taken
1 1 0 1	Taken
1 1 1 0	Taken
1 1 1 1	Taken

7.5.1.4 The Branch Queue

Each time the BPU makes a prediction on a branch opcode, it sends information about that prediction to the Ebox on the I%BRANCH_BUS_H<1:0> The Ebox maintains a queue of branch data entries containing information about branches that have been processed by the BPU but not by the Ebox. The bus is 2 bits wide: one valid bit and one bit to indicate whether the Ibox took the branch or not. Entries are made to the branch queue for both conditional and unconditional branches. For unconditional branches, the value of I%BRANCH_BUS_H<0> is ignored by the Ebox. The branch queue length is selected such that it does not overflow, even if the entire instruction queue is filled with branch instructions, and there are branch instructions currently in the Ebox pipeline. At any one time there may be only one conditional branch in the queue. A queue entry is not made until a valid displacement has been processed. In the case of a second conditional branch encountered while a first is still outstanding, the entry may not be made until the first conditional branch has been retired.

7.5.1.5 Branch Mispredict

When the Ebox executes a branch instruction and it makes the final determination on whether the branch should or shouldn't be taken, it removes the next element from the branch queue and compares the direction taken by the Ibox with the direction that should be taken. If these differ, then the Ebox sends **E%BRANCH_MISPREDICT_L** to the BPU. A mispredict causes the Ibox to stop processing, undo (using the RLOG) any GPR modifications made while parsing down the wrong path, and restart processing at the correct alternate PC.

7.5.1.6 Branch Stall

The BPU back-pressures the IBU by asserting **BRANCH_STALL** when it encounters a new conditional branch with a conditional branch already outstanding. If the BPU has processed a conditional branch but the Ebox has not yet executed it, then another conditional branch causes the BPU to assert **BRANCH_STALL**. Unconditional branches that occur with conditional branches outstanding do not create a problem because the instruction stream merely requires redirection. The alternate PC remains unchanged until resolution of the conditional branch. The Ebox informs the BPU with the **E%BCOND_RETIRE_L** each time a conditional branch is retired from the branch queue in order for the BPU to free up the alternate PC and other conditional branch hardware.

BRANCH_STALL blocks the Ibox from processing further opcodes. When **BRANCH_STALL** is asserted, the IBU finishes parsing the current conditional branch instruction, including the branch displacement and any assists, and then the IBU stalls. The branch queue entry to the Ebox is made after the first conditional branch is retired. At this time, **BRANCH_STALL** is de-asserted and the alternate PC for the first conditional branch is replaced with that for the second.

BSTL_FRC_PCQ is a signal used by the PC queue logic to force an entry into the PC queue when the second conditional branch is finally processed by the BPU after the release of a **BRANCH_STALL**. During a **BRANCH_STALL**, the PC queue refrains from updating the last entry to point to the next instruction until the stall breaks and the BPU finishes processing the second conditional branch.

7.5.1.7 PC Loads

The BPU distributes all PC loads to the rest of the Ibox.

Ibox PC loads from the CSU microcode load a new PC in one of two ways. When the CSU asserts **PC_LD_WBUS**, it drives a new PC value on the **I%IBOX_IW_BUS_H**<31:0> lines. **PC_LD_MD** indicates that the new PC is on the **M%MD_BUS_H**<63:0> lines. The BPU responds by forwarding the appropriate value onto the **NEW_PC**<31:0> lines and asserting **LOAD_NEW_PC**. These Ibox PC loads do not change conditional branch state in the BPU.

The Ebox signals its intent to load a new PC by asserting **E%IBOX_LOAD_PC_L**. The assertion of this signal indicates that the next piece of IPR data to arrive on the **M%MD_BUS_H**<63:0> is the new PC. The next time the Mbox asserts **M%IBOX_IPR_WR_H**, the new PC is taken from **M%MD_BUS_H**<31:0> and forwarded onto **NEW_PC**<31:0> and **LOAD_NEW_PC** is asserted.

The BPU performs unconditional branches by adding the sign extended branch displacement to the current PC, driving the new PC onto the **NEW_PC**<31:0> lines and asserting **LOAD_NEW_PC**. Conditional branches load the PC in the same fashion if the logic predicts a branch taken. The following actions occur on a conditional branch mispredict or Ebox PC load:

- any pending conditional branch is cleared
- pending unconditional branches are cleared
- any pending write to the Ebox branch queue is cleared
- **I%FLUSH_IREF_LAT_H** is asserted to abort pending Istream fill requests in the Mbox

7.5.1.8 Branch Prediction IPR Register

The **BPCR** IPR provides control for the BPU and read/write access to the history array. The write-only **BPCR<FLUSH_BHT>** bit causes a BPU branch history table flush. The flush is identical to the context switch flush, which resets all branch table entries to a neutral value: history bits = 0100. The write-only **BPCR<FLUSH_CTR>** bit causes the **BRANCH_TABLE_COUNTER<8:0>** to be cleared. The **BRANCH_TABLE_COUNTER** provides an address into the branch table for IPR read and write accesses. Each IPR read from the **BPCR** or write to the **BPCR** with **BPCR<LOAD_HISTORY>** = 1 increments the counter. This allows IPR branch table reads and writes to step through the branch table array. **BPCR<LOAD_HISTORY>** enables writes to the branch history table. A write to the **BPCR<HISTORY>** field with **BPCR<LOAD_HISTORY>** = 1 causes a BPU branch history table write. The history bits for the entry indexed by the counter is written with the IPR data. **BPCR** reads supply the history bits in **BPCR<HISTORY>** for the entry indexed by the counter. **BPCR<MISPREDICT>** will return a "1" if the last conditional branch mispredicted. **BPCR**<31:16> contain the branch prediction algorithm. Any IPR write to the **BPCR** will update the algorithm. An IPR read will return the value of the current algorithm. For example, a "0" in **BPCR**<16> means that the next branch encountered will not be taken if the history is "0000". A "1" in **BPCR**<21> means that the next branch encountered when the prior history is "0101" will be taken.

Figure 7–14: IPR D4 (hex), BPCR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10  9  8| 7  6  5  4| 3  2  1  0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|             BPU_ALGORITHM                                     |            0              |   |   |   |   |  0|   history     | :BPCR
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                                                                                              ^   ^   ^   ^           ^
                                                                                              |   |   |   |           |
                                                                              LOAD_HISTORY ---+   |   |   |           |
                                                                                     FLUSH_CTR ---+   |   |           |
                                                                                         FLUSH_BHT ---+   |           |
                                                                                            MISPREDICT ---+           |
                                                                                                          HISTORY  ---+
```

```
The microcode will write the following bit pattern as part of the power-up sequence:

 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10  9  8| 7  6  5  4| 3  2  1  0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 1   1   1   1   1   1   1   0   1   1   0   0   1   0   1   0 |                   All 0's                                     |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Table 7–31: BPCR Field Descriptions

Name	Extent	Type	Description
HISTORY	3:0	RW	Branch history table entry history bits.
MISPREDICT	5	RO	Indicates if last conditional branch mispredicted.
FLUSH_BHT	6	WO	Write of a 1 resets all history table entries to a neutral value, hardware clears bit.
FLUSH_CTR	7	WO	Write of a 1 resets BPCR address counter to 0, hardware clears bit.
LOAD_HISTORY	8	WO	Write history array addressed by BPCR address counter.
BPU_ALGORITHM	31:16	RW	Controls direction of branch for given history.

MACROCODE RESTRICTION

If an MTPR to the BPCR register is followed by a conditional branch instruction, the prediction algorithm used for this branch is unpredictable. Furthermore, the branch history table update is also unpredictable. The BPU functions correctly, but programs which depend on particular patterns of branch predictions (such as diagnostic tests) should avoid placing conditional branch instructions immediately after an MTPR instruction that writes to the BPCR register.

Bits 8,7,6 are defined in Table 7–32 for IPR writes to the BPCR. NOTE: The prediction algorithm will be updated on every IPR write to the BPCR.

Table 7–32: BPCR <8:6>

BIT 8	BIT 7	BIT 6	Write Action
0	0	0	Do nothing, except update algorithm
0	0	1	Flush branch table. History not written
0	1	0	Address counter reset to 0. History not written
0	1	1	Flush branch table, reset address counter, history not written
1	0	0	Write history to table, counter automatically increments
1	0	1	Undefined: Branch table flushed, new history written, counter incremented
1	1	0	Undefined: Write history to old counter value, counter reset to 0
1	1	1	Undefined: Branch table flushed, write history to old counter value, counter reset to 0

7.6 PC Load Effects

This section summarizes the various effects of loading a new PC in the Ibox. New PCs are loaded from four different sources. The BPU receives the new PCs from all these sources, drives the new PC on NEW_PC<31:0>, and asserts LOAD_NEW_PC. The four sources for new PCs in priority order are :

1. **Ebox PC load from the M%MD_BUS_H<31:0>**
   The Ebox loads a new PC as a result of an interrupt or exception or for instructions like REI, HALT, CASEx etc. After the Ebox asserts the E%IBOX_LOAD_PC_L signal, the PC is supplied on the M%MD_BUS_H<31:0>, along with the M%IBOX_IPR_WR_H signal. The BPU selects M%MD_BUS_H<31:0> to drive NEW_PC<31:0> and asserts LOAD_NEW_PC.
2. **Branch Mispredict PC**
   When a mispredict has been detected, the BPU drives NEW_PC<31:0> from the alternate PC latch containing the address of the branch path not taken, and asserts LOAD_NEW_PC.
3. **PC_LD_WBUS from the CSU**
   For instructions like JSB and JMP, the CSU computes a new PC and drives that PC up to the BPU. The BPU receives the PC on I%IBOX_IW_BUS_H <31:0>, drives NEW_PC<31:0> and asserts LOAD_NEW_PC.
4. **PC_LD_MD from the CSU**
   For instructions like JSB, JMP, RET and RSB, the CSU requests a new PC from the Mbox. The CSU asserts PC_LD_MD, and the next M%IBOX_DATA_L signals the new PC is on the M%MD_BUS_H<31:0>. The BPU receives the PC on M%MD_BUS_H <31:0>, drives NEW_PC<31:0> and asserts LOAD_NEW_PC.
5. **Branch Destination PC**
   For unconditional branches or when the BPU predicts a conditional branch as taken, it computes the branch destination, drives NEW_PC<31:0>, and asserts LOAD_NEW_PC.

The effects of loading a new PC are shown below. These effects take place regardless of the source of the PC.

- PREFETCH_ENABLE is set in the VIC.
- VIBA<31:3> in the VIC are loaded from NEW_PC<31:3>
- MHARD_ERR is cleared in the VIC.
- IMMGT_EXC is cleared in the VIC.
- MISS_PENDING is cleared in the VIC.
- WRITE_PENDING is cleared in the VIC.
- VIC_READ is set in the VIC, allowing a new cache read sequence from the new address.
- The PFQ is flushed and NEW_PC<2:0> are latched as the initial BYTES_RETIRED.
- The BPU asserts I%FLUSH_IREF_LAT_H indicating that the Mbox should flush its IREF latch.
- The IBU stops the parser and latches the new PC from NEW_PC<31:0>.
- The IIU latches the new PC as the next entry in PC queue.

7.6.1 Mispredict PC Loads

When a PC load is the result of a branch mispredict, additional actions must be taken as described below

- All pending conditional and unconditional branches are cleared in the BPU.
- Pending branch queue writes are aborted by the BPU.
- In the IIU, the instruction queue free counter is cleared.
- In the IIU, the PC queue is flushed
- In the IIU, ISSUE_STALL is cleared.
- The SBU clears the scoreboard array counters.
- In the CSU, the S1 stage produces the mispredict RLOG unwind microaddress. The S3 stage is forced to NOP. NOTE: The RLOG is NOT flushed.
- In the CSU, IMD_VALID is reset.
- In the OQU, the MD allocation pointer is reset and the MD allocation counter is cleared.
- In the OQU, the source queue free counter is cleared.
- In the OQU, the destination queue free counter is cleared.
- In the CSU, the LD_PC_MD latch is cleared.

7.6.2 Ebox PC Loads

When the Ebox is the source of the new PC, the signal E%IBOX_LOAD_PC_L is asserted several cycles before the actual PC arrives from the Mbox. After this signal is asserted, but before the new PC is loaded, the signal E%RESTART_IBOX_H may be asserted, starting the parser and VIC prefetching. To avoid parsing from the wrong instruction stream, the following actions are taken upon the assertion of E%IBOX_LOAD_PC_L.

- The PFQ is flushed, forcing PFQ_EMPTY to be asserted.
- VIC prefetching is disabled until LOAD_NEW_PC is asserted by the BPU. This also blocks VIC bypass to the PFQ.

- **MHARD_ERR** is cleared in the VIC.
- **IMMGT_EXC** is cleared in the VIC.

MICROCODE RESTRICTION

> **E%IBOX_LOAD_PC_L** and **E%IBOX_IPR_WRITE_H** must not occur in the same cycle. **E%IBOX_LOAD_PC_L** and **E%RESTART_IBOX_H** must not occur in the same cycle.

7.7 E%STOP_IBOX effects

When the Ebox microcode performs a **MISC/RESET_CPU** it asserts **E%STOP_IBOX_H**. The Ibox requires **E%STOP_IBOX_H** to be asserted whenever **RESET_L** is asserted.

MICROCODE RESTRICTION

> **E%STOP_IBOX_H** must always be followed by **E%IBOX_LOAD_PC_L** and then **E%RESTART_IBOX_H**. **E%STOP_IBOX_H** and **E%BRANCH_MISPREDICT_L** cannot occur in the same cycle.

The effects of this signal on the various sub-sections in the Ibox are shown below.

- **PREFETCH_ENABLE** is cleared in the VIC
- **MISS_PENDING**, **WRITE_PENDING**, and **READ_STATE** are cleared in the VIC, putting the VIC in an idle state.
- **IHARD_ERR** is cleared in the VIC.
- **MHARD_ERR** is cleared in the VIC.
- **IMMGT_EXC** is cleared in the VIC.
- In the IIU, the instruction queue free counter is cleared.
- In the IIU, **ISSUE_STALL** is cleared.
- The **IQ_VALID** signal, from the IIU to the Ebox, is cleared.
- The Istream parser in the IBU is stopped.
- The signals **I%IMEM_HERR_H** and **I%IMEM_MEXC_H** are cleared.
- The **PREV_NOT_DONE** signal is cleared in the IBU
- **CSU_LD_PC_PEND** is cleared in the IBU
- **LD_NEW_PC_PEND** is cleared in the IBU
- The FD opcode flip-flop is cleared in the IBU
- The IDLE microword is injected into all stages of CSU pipeline. However, NOTE: RLOG unwind is not aborted.
- If an IPR read to back-up PC with RLOG unwind is in progress, the unwind completes as normal, but the back-up PC write to the Ebox working register is disabled. All other Ibox IPR accesses are aborted.
- **IMD_VALID** is reset in the CSU
- The IREF-pending latch is cleared in the CSU
- The **PC_LD_MD** - pending latch is cleared in the CSU
- The IPR read/write select signals reset in the CSU
- The stage 1 valid bit is cleared in the CSU

- The source queue allocation counter is cleared in the OQU
- The destination queue allocation counter is cleared in the OQU
- The MD allocation counter is cleared in the OQU
- The MD index counter is cleared in the OQU
- The source and destination scoreboard counters are cleared in the SBU
- Branch stalls are cleared in the BPU
- I%FLUSH_IREF_LAT_H is asserted

7.8 Initialization

7.8.1 Mechanisms for Ibox State Reset

The Ibox depends on the E%STOP_IBOX_H signal to initialize the states shown in Section 7.7. In addition, RESET_L is used to clear those states listed below which cannot be initialized by E%STOP_IBOX_H.

- VIC_ENABLE is cleared in the VIC.
- RLOG pointers are reset in the CSU.
- The IDLE microword is injected into stage 1 of the CSU pipeline.
- PC queue pointers are reset in the IIU.

7.9 Errors, Exceptions, and Faults

7.9.1 Overview

The Ibox handles some of the processing for memory hardware errors, memory management exceptions, and reserved opcode faults, and reserved addressing mode faults. A global view of error, exception, and fault handling is presented here. Implementation details are distributed amongst the Ibox sub-section text.

Istream memory hardware errors may originate in the Mbox and memory subsystem or in the VIC array. Dstream memory hardware errors originate in the Mbox and memory subsystem. Istream and Dstream memory management exceptions originate in the Mbox. Reserved opcodes and reserved addressing modes are detected in Ibox hardware during instruction parsing.

7.9.2 Istream Memory Errors

When the Mbox conditions returning Istream data with M%MME_FAULT_H or M%HARD_ERR_H, the VIC and PFQ writes are inhibited, prefetching is disabled, and the VIC sets appropriate condition flags for the IBU. The IBU continues to parse until it attempts to parse the Istream data that caused the exception or error. The condition flags are then forwarded to the Ebox. If the Ebox detects an empty instruction queue, source queue, destination queue, or field queue while the exception or error condition is asserted, the Ebox initiates an exception microtrap.

Any PC load or E%STOP_IBOX_H resets the error and exception flags in the VIC. An Ibox PC load or E%RESTART_IBOX_H restarts prefetching and parsing. Thus if the error or exception gets forwarded to the Ebox, the Ebox can reset the Ibox flags, load a new PC and continue. If the instruction stream branches around the instruction stream data responsible for the error or exception, the Ibox resets the error flags and continues without reporting the condition.

If a VIC parity error is detected, VIC prefetching and IBU instruction parsing are halted immediately and the error forwarded to the Ebox. This action is taken because the data containing the error may already have been loaded into the PFQ. If the Ebox detects an empty instruction queue, source queue, destination queue, or field queue while the exception or error condition is asserted, the Ebox initiates an exception microtrap. Section 7.2.1.7 and Section 7.3.2.15 contain the Ibox implementation details of Istream error and exception handling. See Table 8–12 and Section 8.5.19 for Ebox implementation details.

7.9.3 Dstream Memory Errors

Memory errors on incoming Dstream data are detected during the processing of some deferred mode specifiers. In auto-increment deferred and displacement deferred specifier modes, the complex specifier unit reads the address of an operand from memory. This memory read is followed either by a direct write to an Ebox MD, or an operand memory reference to read the actual operand into an Ebox MD and/or create a PA queue entry for a result store.

If the Mbox returns M%MME_FAULT_H or M%HARD_ERR_H, then in the case of a direct MD write, the appropriate flag is sent with the MD write to the Ebox. If the Ebox detects one of the flags during an MD file access, it initiates an exception microtrap. If a memory operation is required to complete the processing of the specifier, the appropriate error or exception flag, sent with the memory request. The Mbox forces a memory management error or exception to occur for that reference, causing a fault flag to be returned to the appropriate Ebox MD.

Section 7.4.2.2.2 contains the Ibox implementation details of Dstream error and exception handling. See Table 8–12 and Section 8.5.19 for Ebox implementation details and Section 12.6.5 for Mbox implementation details.

7.9.4 Reserved Opcode Faults

Reserved opcode faults occur when the IBU detects unimplemented or reserved opcodes during instruction parsing. All such opcodes stop the parser and make an Ebox instruction queue entry containing a microcode dispatch for the reserved opcode routine. Section 7.3.2.12 contains the Ibox implementation details for reserved opcode handling.

7.9.5 Reserved Addressing Mode Faults

Reserved Addressing Mode Faults occur due to illegal combinations of specifier mode, specifier register, and access type. Unpredictable addressing modes occur due to combinations of specifier mode, specifier register, access type, and data length that do not make sense. Table 7–33 summarizes the behavior of the Ibox on reserved and unpredictable addressing modes. Reserved addressing modes as specified by the VAX Architecture Standard always cause reserved addressing mode faults. Unpredictable addressing modes may produce a fault, or may be allowed to continue even though the result does not make sense. The processing of unpredictable modes never hangs the machine.

Table 7–33: Reserved Addressing Mode Faults

Address Mode	Access Type	GPRs	Data Length	Indexed	Action
S^#literal	Modify	-	-	-	take required fault
S^#literal	Write	-	-	-	take required fault
S^#literal	Address	-	-	-	take required fault
S^#literal	Field	-	-	-	take required fault
S^#literal	-	-	-	Yes	take required fault
base[Rx]	-	PC	-	-	take required fault
base[Rx]	-	-	-	Yes	take required fault
Rn	Address	-	-	-	take required fault
Rn	-	-	-	Yes	take required fault
(Rn)+	Modify	PC	-	-	take required fault
(Rn)+	Write	PC	-	-	take required fault
Rn	-	PC	-	-	source/dest queue entry has Rn=PC
Rn	-	SP	q,d,g	-	2nd source/dest queue entry has Rn=PC
Rn	-	SP,AP,FP	o,h	-	unimplemented data lengths
(Rn)	-	PC	-	-	Operand address is unpredictable
–(Rn)	-	PC	-	-	Operand address is unpredictable
–(Rn)	-	-	-	Rx=Rn	Rx read for index, then Rn read for base
(Rn)+	-	-	-	Rx=Rn	Rx read for index, then Rn read for base
@(Rn)+	-	-	-	Rx=Rn	Rx read for index, then Rn read for base
(Rn)+	Address	PC	-	-	PC after specifier byte passed as address
(Rn)+	-	PC	-	Yes	Rx for index is read but not used

When a Reserved Addressing Mode Fault is detected, I%RSVD_ADDR_FAULT_H is asserted, VIC prefetching is stopped, the IBU is stopped, and the CSU goes idle. A Reserved Addressing Mode Fault also blocks the OQU from making the source queue or destination queue entry associated with the faulting operand.

If the Ebox detects an empty source queue, destination queue, or field queue while I%RSVD_ADDR_FAULT_H is asserted, the Ebox initiates an exception microtrap.

All reserved addressing mode fault conditions are cleared in the Ibox when the Ebox loads a new PC.

7.10 Ibox Signal Name Cross-Reference

All signal names referenced in this chapter have appeared in bold and reflect the actual name appearing in the NVAX schematic set. For each signal appearing in this chapter, the table below lists the corresponding name which exists in the behavioral model.

Table 7–34: Cross-reference of all names appearing in the Ibox chapter

Schematic Name	Behavioral Model Name
I%BRANCH_BUS_H<1:0>	I%BRANCH_BUS_H<1:0>
I%FORCE_HARD_FAULT_H	I%FORCE_HARD_FAULT_H
I%FORCE_MME_FAULT_H	I%FORCE_MME_FAULT_H
I%IBOX_IA_ADDR_H<3:0>	I%IBOX_IA_ADDR_H<3:0>
I%IBOX_IA_READ_H	I%IBOX_IA_READ_H
I%IBOX_IW_ADDR_H<4:0>	I%IBOX_IW_ADDR_H<4:0>
I%IBOX_IW_BUS_H<31:0>	I%IBOX_IW_BUS_H<31:0>
I%IBOX_IW_WRITE_H	I%IBOX_IW_WRITE_H
I%IBOX_S_ERR_L	I%IBOX_S_ERR_L
I%IMEM_HERR_H	I%IMEM_HERR_H
I%IMEM_MEXC_H	I%IMEM_MEXC_H
I%IQ_BUS_H<22:0>	I%IQ_BUS_H<22:0>
I%OPERAND_BUS_H<14:0>	I%OPERAND_BUS_H<14:0>
I%PMUX0_H	I%PMUX0_H
I%PMUX1_H	I%PMUX1_H
I%RSVD_ADDR_FAULT_H	I%RSVD_ADDR_FAULT_H
E%BCOND_RETIRE_L	E%BCOND_RETIRE_H
E%BRANCH_MISPREDICT_L	E%BRANCH_MISPREDICT_H
E%DQ_RETIRE_H	E%DQ_RETIRE_H
E%DQ_RETIRE_RMODE_H	E%DQ_RETIRE_RMODE_H
E%DQ_RETIRE_RN_H<3:0>	E%DQ_RETIRE_RN_H<3:0>
E%FLUSH_BPT_H	E%FLUSH_BPT_H
E%FLUSH_PCQ_H	E%FLUSH_PCQ_H
E%FLUSH_VIC_H	E%FLUSH_VIC_H
E%FPD_SET_L	E%FPD_SET_H
E%IBOX_IA_BUS_H<31:0>	E%IBOX_IA_BUS_H<31:0>
E%IBOX_IPR_NUM_H<3:0>	E%IBOX_IPR_NUM_H<3:0>
E%IBOX_IPR_READ_H	E%IBOX_IPR_READ_H
E%IBOX_IPR_TAG_H<2:0>	E%IBOX_IPR_TAG_H<2:0>
E%IBOX_IPR_WRITE_H	E%IBOX_IPR_WRITE_H

Table 7–34 (Cont.): Cross-reference of all names appearing in the Ibox chapter

Schematic Name	Behavioral Model Name
E%IBOX_LOAD_PC_L	E%IBOX_LOAD_PC_H
E%RESTART_IBOX_H	E%RESTART_IBOX_H
E%RETIRE_INSTR_L	E%RETIRE_INSTR_H
E%SQ_RETIRE_H<1:0>	E%SQ_RETIRE_H<1:0>
E%SQ_RETIRE_MD_H<1:0>	E%SQ_RETIRE_MD_H<1:0>
E%SQ_RETIRE_RMODE_H<1:0>	E%SQ_RETIRE_RMODE_H<1:0>
E%SQ_RETIRE_RN1_H<3:0>	E%SQ_RETIRE_RN1_H<3:0>
E%SQ_RETIRE_RN2_H<3:0>	E%SQ_RETIRE_RN2_H<3:0>
E%STOP_IBOX_H	E%STOP_IBOX_H
I%FLUSH_IREF_LAT_H	I%FLUSH_IREF_LAT_H
I%FORCE_HARD_FAULT_H	I%FORCE_HARD_FAULT_H
I%FORCE_MME_FAULT_H	I%FORCE_MME_FAULT_H
I%IBOX_ADDR_H<31:0>	I%IBOX_ADDR_H<31:0>
I%IBOX_AT_L<1:0>	I%IBOX_AT_H<1:0>
I%IBOX_CMD_L<4,1:0>	I%IBOX_CMD_H<4:0>
I%IBOX_DL_L<1:0>	I%IBOX_DL_H<1:0>
I%IBOX_REF_DEST_L<1:0>	I%IBOX_REF_DEST_H<1:0>
I%IBOX_TAG_L<2:0>	I%IBOX_TAG_H<2:0>
I%IREF_REQ_H	I%IREF_REQ_H
I%SPEC_REQ_H	I%SPEC_REQ_H
M%HARD_ERR_H	M%HARD_ERR_H
M%IBOX_DATA_L	M%IBOX_DATA_H
M%IBOX_IPR_WR_H	M%IBOX_IPR_WR_H
M%LAST_FILL_H	M%LAST_FILL_H
M%MD_BUS_H<63:0>	M%MD_BUS_H<63:0>
M%MD_BUS_QW_PARITY_L	M%MD_BUS_QW_PARITY_H
M%MME_FAULT_H	M%MME_FAULT_H
M%QW_ALIGNMENT_H<1:0>	M%QW_ALIGNMENT_H<1:0>
M%SPEC_Q_FULL_H	M%SPEC_Q_FULL_H
M%VIC_DATA_L	M%VIC_DATA_H

7.11 Testability

7.11.1 Overview

Ibox testability is enhanced by architecturally accessible features, and connections to the internal scan register and the parallel port.

7.11.2 Internal Scan Register and Data Reducer

Ibox state can be latched into the scan register and shifted off-chip through the global internal scan register. The shift out begins with scan register bit 0. See Chapter 19 for the implementation details of the internal scan register. Table 7–35 lists the states in the Ibox scan register. Under global control from the test port, the Ibox scan register can be configured as a LFSR.

Table 7–35: Ibox Scan Register Fields

Bit Field	Field Name	Description
<0>	STP_RESTART	Stop parser flag
<1>	STP_SUPPRESS	Stop parser flag
<2>	SHLIT	specifier control <0>, short literal
<8:3>	RN/SHORT LITERAL	specifier control <6:1>, register or shlit value
<11:9>	AT	specifier control <9:7>, access type
<13:12>	DL	specifier control <11:10>, data length
<14>	VALID	specifier control <12>, valid
<15>	COMPLEX	specifier control <13>, complex specifier
<18:16>	DISPATCH	specifier control <16:14>, dispatch address
<19>	AT_RMW	specifier control <17>, RMW
<20>	INDEXED	specifier control <18>, index
<21>	ASSIST	specifier control <19>, assist
<22>	PC_MODE	specifier control <20>, PC mode
<23>	JMP_OR_JSB	specifier control <21>, JMP or JSB
<25:24>	E_DL	execution data length <1:0>

7.11.3 Parallel Port

The CSU microcode address is routed to the chip parallel port. The microcode address can be monitored on a cycle by cycle basis during chip debug by selecting the Ibox as source to the parallel port. When selected, a buffered version of the control store address, MUX_H<6:0>, appears on PP_DATA<6:0>. See Chapter 19 for the implementation details of the parallel port.

7.11.4 Architectural Features

Internal processor registers are included as architectural features to aid in testability. IPR access to VIC tags and data is available through the VTAG and VDATA registers. See Section 7.2.1.16 for the implementation details of the these registers. IPR access to the branch history table and branch status is available through the BPCR register. See Section 7.5.1.8 for the implementation details of the BPCR.

7.12 Performance Monitoring Hardware

7.12.1 Signals

The Ibox provides two signals for performance monitoring: I%PMUX0_H asserts on every VIC access and I%PMUX1_H asserts on every VIC hit. These signals enable the Ebox performance monitoring hardware to gather statistics on VIC hits versus VIC accesses.

7.13 Revision History

Table 7–36: Revision History

Who	When	Description of change
John F. Brown	19-Feb-1991	Update following pass 1 tape out
John F. Brown, Ruben Castelino, Mary Field, Paul Gronowski, Jeanne Meyer	12-Jan-1990	Intermediate release.
John F. Brown, Paul Gronowski, Jeanne McKinley	06-Mar-1989	Release for external review.
John F. Brown	19-Dec-1988	Partial Update.
Shawn Persels	06-Oct-1988	Initial release.

Chapter 8

The Ebox

8.1 Chapter Overview

This chapter describes the Ebox section of the NVAX CPU chip. Only the major functional blocks, their interfaces to each other, and the interface to the rest of the NVAX system are described here. Circuit level implementation details are not of primary concern in this document.

8.2 Introduction

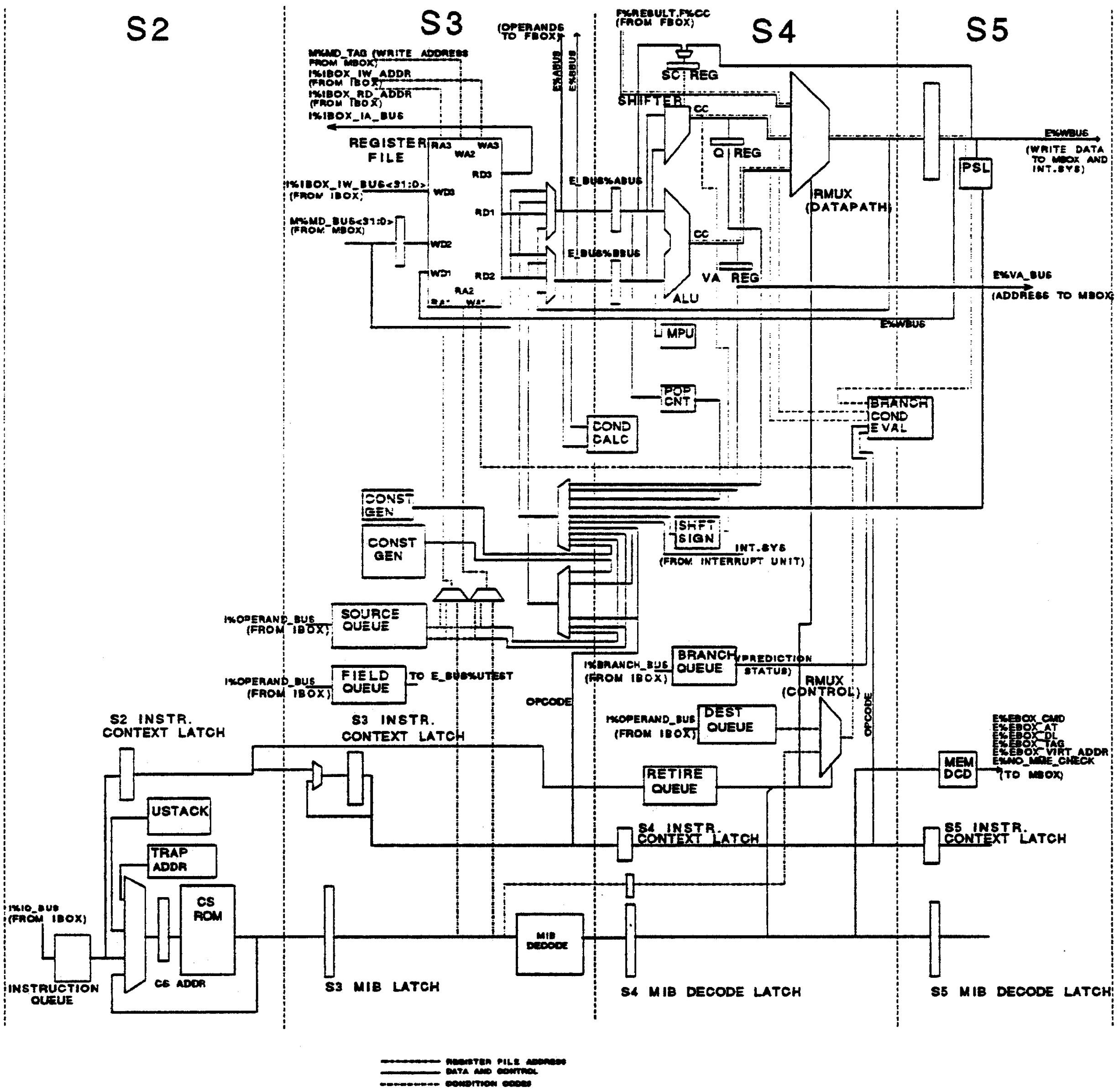
The Ebox is the instruction execution unit in the NVAX CPU chip. It is a 3 stage pipeline (S3..S5) which runs semi-autonomously to the rest of the NVAX chip and supports the following functions:

- **Instruction Execution**
  The Ebox is responsible for carrying out the execution portion of each VAX instruction under control of a microflow whose initial address is provided by the Ibox issue unit.
- **Instruction Coordination**
  The Ebox is a major source of control to coordinate instruction processing in the Ibox, Mbox, and Fbox. It ensures that Ebox and Fbox macroinstructions retire in the proper order, and it provides controls to the Mbox and Ibox which help manage certain inter-macroinstruction dependencies. The Ebox cooperates with the Ibox in handling mispredicted branches.
- **Trap, Fault and Exception Handling**
  The Ebox coordinates trap, fault, and interrupt handling. It delays the condition until all preceding macroinstructions complete properly. It then collects information about the condition and ensures that the correct architectural state is reached.
- **CPU Control**
  Most CPU control is provided by the Ebox. Ebox control functions include CPU initialization, controlling Ibox, Fbox, and Mbox activities, and setting control bits during major CPU state changes (e.g. taking an interrupt or executing a change mode instruction).

The Ebox accomplishes many of the above functions by executing the NVAX Ebox microcode. This chapter views the Ebox as the interpreter of microcode. Describing how microcode functions are used to correctly emulate the VAX architecture or the architectural motivation for Ebox hardware functions is generally outside the scope of this discussion.

Figure 8–1 at the end of this section is a top level block diagram of the Ebox showing all the major Ebox function units, their interconnections, and their place in the pipeline. The pipeline segments are shown in the diagram (S2, S3, S4, and S5). The sections following the diagram describe the function elements depicted and the Ebox pipeline.

Figure 8–1: Ebox Block Diagram



8.3 Chapter Structure

The Ebox is described from both an overall functional and individual function unit standpoint. The top level description is of the major Ebox functions. The next level consists of a detailed description of each of the Ebox function units.

The Ebox functions are described in the initial sections of this chapter. They are presented referring to the microcode fields which control the Ebox. Within each section the Ebox functions in question are discussed in detail and the Ebox function units which support that function are introduced. The functional overview is followed by a comprehensive description of the each of the Ebox function units.

The latter sections of this document describe Ebox initialization, timing, error handling, testability and other details not related to the main-line functionality of the Ebox.

8.4 Ebox Overview

8.4.1 Microword Fields

The Ebox is controlled by the data path control portion of the microword, which is either standard or special format. The other portion of the control word, the microsequencer control portion, controls the microsequencer which determines which microword is fetched in every cycle. The fields of the data path control portion of the microword and their effect within the Ebox are shown in Table 8–1. For more information on microword formats and field widths see Chapter 6.

NOTATION

> The notation FIELD/FUNCTION is used throughout this chapter to mean that microword field FIELD specifies FUNCTION.

Table 8–1: Data Path Control Microword Fields

Microword Field	Microword Format	Description
FORMAT	Both	This one-bit field determines whether the microword is in the special format. If it is 1, the MISC1, MISC2, and D fields exist. If it is 0, the Q, SHF, and VAL fields exist instead.
LIT	Both	This one-bit field determines whether the microword is the constant generation variant (format). If it is 1, the POS and CONST fields exist. If it is 0, the VAL and B fields exist instead in standard format, and the MISC2, D, and B fields exist instead in special format.
ALU	Both	Sets the ALU function, including typical ALU operations, and others.
MRQ	Both	Controls initiation of Ebox memory accesses and other Mbox control functions. The Ebox decodes the field and sends the corresponding request to the Mbox.
SHF	Standard	Sets the shifter function. The W and Q fields control how the shifter output is used. Some settings of this field specify a pass operation instead of a shift.

Table 8–1 (Cont.): Data Path Control Microword Fields

Microword Field	Microword Format	Description
VAL	Standard[1]	Specifies the shift amount (1 to 31) or, if VAL = 0, specifies to shift the amount in the SC register.
A	Both	Specifies the source of E_BUS%ABUS_L<31:0> for this microword. The A field can select any element in the register file or one of several of Ebox sources. E_BUS%ABUS_L<31:0> is one of the two sources for the ALU and the shifter.
B	Both[1]	When the source of E_BUS%BBUS_L<31:0> is a register this field specifies the source of E_BUS%BBUS_L<31:0>. The B field can select from some of the elements in the register file or from a small number of other Ebox sources. E_BUS%BBUS_L<31:0> is one of the two sources for the ALU and the shifter.
POS	Both[2]	When the source of E_BUS%BBUS_L<31:0> is from the constant generator this field specifies which byte the constant value is in. Bytes 0 through 3 may be specified. The other bytes are forced to 0.
CONST	Both[2]	This field contains the literal byte value which is sourced to one of the bytes of E_BUS%BBUS_L<31:0> as specified by the POS field. (The other E_BUS%BBUS_L<31:0> bytes are forced to 0.)
CONST.10[3]	Both[2]	This field contains the literal 10-bit value which is sourced to E_BUS%BBUS_L<9:0>. (E_BUS%BBUS_L<31:10> are forced to 0.)
DST	Both	This field specifies the destination of E_BUS%WBUS_L<31:0>. The possible destinations include a subset of the register file and a number of other Ebox destinations.
Q	Standard	Controls whether or not the Q register is loaded with the shifter output for this microword.
W	Both	Selects the driver of E_BUS%WBUS_L<31:0>. Either the ALU or the shifter output is driven on E_BUS%WBUS_L<31:0>.
L	Both	This field controls whether the Ebox operations are done with a data length of longword or the length specified in the DL register. The Ebox operations affected are condition code calculation, size of memory operations, zero extending of E_BUS%WBUS_L data, and bytes affected by register file writes.
V	Both	Controls updating of the VA register. Either the VA register is updated with the value from the ALU, or it is not changed from its previous value.
MISC	Both	This field has many uses. Only one use can be selected at a time. This field can control PSL condition code alterations, set the DL register, set or clear state flags, or invoke a box coordination or control function.
MISC1	Special	This field can specify one of a few Ibox or Fbox coordination or control functions, and can be used to set or clear state flags.
MISC2	Special[1]	One Mbox control function and one to add an Fbox destination scoreboard entry.
DISABLE.RETIRE	Special[1]	This field is used to disable retire of macroinstructions and retire queue entries

[1] Not constant generation microword variant.

[2] Constant generation microword variant.

[3] The CONST.10 field is actually the POS field bitwise concatenated with the CONST field, with the POS field in the more significant position. It is simply a way of treating these two microword fields as one. CONST.10 is only used when MISC/CONST.10.BIT is specified.

When a microword field is not present in all formats, it defaults to NOP (no operation) when a microword format without that field occurs. More specifically, standard format microwords effectively specify MISC1/NOP, MISC2/NOP, and DISABLE.RETIRE/NO by default. Special format microwords effectively specify Q/HOLD.Q, SHF/NOP, and VAL/0. When the microword is the constant generation variant of the standard format microword, VAL/0 is effectively specified, and the B field is ignored since this microword variant sources a constant onto **E_BUS%BBUS_L<31:0>**. In the constant generation variant of the special format microword, MISC2/NOP and DISABLE.RETIRE/NO are effectively specified, and the B field is ignored because this microword variant also sources a constant onto **E_BUS%BBUS_L<31:0>**.

8.4.1.1 Microsequencer Control Fields

In addition to decoding the datapath control portion of the microword, the Ebox decodes a part of the Microsequencer control portion of the microword. Specifically, it detects when the SEQ.FMT and SEQ.MUX fields (see Chapter 9 and Chapter 6) specify LAST.CYCLE or LAST.CYCLE.OVERFLOW. The Ebox fault detection logic and the RMUX control logic use these decodes.

8.4.2 The Register File

The register file contains four kinds of registers: MD (memory data), GPR, Wn (working), and CPUSTATE registers. The MD registers receive data from memory reads initiated by the Ibox, and from direct writes from the Ibox. The Wn registers hold microcode temporary data. They can receive data from memory reads initiated by the Ebox and receive result data from ALU, shifter, or Fbox operations, and from the Ibox in the case of Ibox IPR reads. The GPRs are the VAX architecture general-purpose registers (though R15 is not in the file) and can receive data from Ebox initiated memory reads, from the ALU or shifter, or from the Ibox. The CPUSTATE registers hold semipermanent architectural state (e.g. KSP, SCBB). They can only be written by the Ebox.

8.4.3 ALU and Shifter

Each microword specifies source operands for the ALU or shifter (A, B, POS, and CONST fields), operations for these function units to perform (ALU, SHF, and VAL fields), and a destination (or possibly two destinations if Q or VA is updated) for the result(s) (DST, Q, W, and V fields). Note that in special format microwords no shifter operation can be specified and the Q register can't be altered. In the course of executing the microword, the Ebox will fetch the source operands onto **E_BUS%ABUS_L<31:0>** and **E_BUS%BBUS_L<31:0>**, carry out the specified ALU and shifter functions, and store the result in the specified locations (if any).

8.4.3.1 Sources of ALU and Shifter Operands

In general the sources of **E_BUS%ABUS_L<31:0>** and **E_BUS%BBUS_L<31:0>** (the inputs to the ALU and shifter) are either a constant, a register from the register file, an Ebox register (e.g. PSL, Q, or VA), an Ebox source value calculated by a special function unit, a hardware status provided via a special path from outside the Ebox (e.g., interrupt status), or an entry from the source queue. **E_BUS%BBUS_L<31:0>** sources are limited to a subset of the register file, certain Ebox registers, and an entry from the source queue. The source queue is introduced in Section 8.4.4.

8.4.3.2 ALU Functions

The ALU is capable of standard operations on byte, word, and longword size operands. It can pass either input to the output and is capable of a number of arithmetic and logical operations on one or two operands, producing condition codes based on data length and operation. It also has specialized functions which are discussed in Section 8.5.3.

8.4.3.3 Shifter Functions

The shifter does longword and quadword shift operations and certain pass-thru operations, always producing a longword output. The shifter treats the two sources as a single quadword, with E_BUS%ABUS_L<31:0> as the more significant longword. The longword output is this quadword shifted right 0 to 32 bits and truncated to longword length. The shifter produces condition codes based the longword output data.

8.4.3.4 Destinations of ALU and Shifter Results

The output of the shifter and the output of the ALU can drive E_BUS%WBUS_L<31:0>. The shifter output is also directly connected to the Q register so that the Q register can be loaded with the shifter output regardless of the source of E_BUS%WBUS_L<31:0>. In the same way, the ALU output is directly connected to the VA register. E_BUS%WBUS_L<31:0> data is the input to one of the write ports on the register file and can be used to update any register file entry except an MD register. Certain other Ebox registers (e.g. SC, PSL) can be loaded from E_BUS%WBUS_L<31:0>.

The destination of E_BUS%WBUS_L<31:0> can be specified by the current destination queue entry, when the microword so specifies. The destination queue is introduced in the following section.

8.4.4 Ibox-Ebox Interface

The Ibox-Ebox interface is made up of a number of FIFO queues. The purpose of these queues is to allow the Ibox to fetch and decode new instructions before the Ebox is ready to execute them. The Ibox adds entries as it decodes instructions, and the Ebox removes them from the other end as it executes them. For each opcode, there is a predetermined number of entries added to the various queues by the Ibox. Ebox execution microflows remove exactly the right number of entries from each queue.

The queues which interface the Ibox to the Ebox directly are the source queue, the destination queue, the branch queue, and the field queue. The instruction queue, the PA queue, and the retire queue are introduced here for completeness.

The source queue holds source operand information. Entries are added by the Ibox as it decodes the source type operand specifiers of each instruction. The entry is either a pointer into the register file or the data from a literal mode operand specifier. The Ebox accesses and removes an entry each time a microword specifies a source queue access in either the A or B fields. If the entry is literal data, it is used as an ALU and/or a shifter operand. Otherwise the register file is accessed using the pointer in the entry.

The destination queue holds result destination information. Entries are added by the Ibox as it decodes the destination type operand specifiers of each instruction. A destination queue entry is either a pointer to a GPR in the register file or a flag indicating that the result destination is memory. The Ebox accesses and removes an entry each time a microword specifies a destination queue access in the DST field or the Fbox supplies a result which specifies a destination queue access. If the entry is a pointer to a GPR, the Ebox writes the ALU, shifter, or Fbox data into the register file. Otherwise the data is stored in memory at the address found in the PA queue.

The PA queue is in the Mbox. Each time the Ibox adds an entry indicating a memory destination to the destination queue it also sends the Mbox a virtual address to be translated. When the Mbox has translated the address it puts it in the PA queue. If the current destination queue entry indicates a memory destination, the Ebox sends the result data to the Mbox to be written to the physical address found in the PA queue. The Mbox removes the PA queue entry as it uses it.

The branch queue holds status bits for each branch instruction processed by the Ibox. The Ibox adds an entry to the branch queue each time it finishes processing a conditional or unconditional branch. The Ebox references and removes the current branch queue entry in the execution microflow for the branch. This allows the Ebox to synchronize with the Ibox so that the branch does not finish executing until the Ibox has successfully fetched the branch displacement specifier. It also allows the Ebox to check for an incorrect branch prediction by the Ibox.

Each time the Ibox decodes a branch it calculates the branch address. For unconditional branches it simply begins fetching from the new instruction stream immediately. For conditional branches the Ibox predicts whether the branch will be taken or not. The branch queue entry added by the Ibox indicates the branch prediction. When the Ebox executes an unconditional branch, it references the branch queue simply to ensure that the Ibox was able to fetch the displacement specifier without a fault or error. For conditional branches the Ebox also checks that the branch prediction was correct and initiates a microtrap if it wasn't. If the branch wasn't correct, the Ebox notifies the Ibox, which uses the alternate path PC (which it had kept) to begin fetching along the correct path.

The retire queue holds status for each macroinstruction currently being executed in the Ebox or the Fbox. The status indicates which unit will execute the instruction, the Ebox or the Fbox. The Ebox adds an entry each time the Microsequencer dispatches to a macroinstruction execution microflow. The Ebox references the retire queue when the macroinstruction execution is complete in order to ensure that instructions finish executing in the proper order. A certain amount of concurrent execution in the Fbox and Ebox is possible. The retire queue is used to prevent one box from altering any architecturally visible state before the other box's execution for preceding macroinstructions finishes. The Ebox references and removes a retire queue entry each time an Fbox or Ebox instruction is retired.

The field queue holds a one-bit type status for variable-length bit field base address operands processed in the Ibox. (Note that some operands are treated as variable-length bit field base address operands internally by the NVAX CPU even though the operand is not really the base address of a variable-length bit field. These operands, including the true bit field base address operands, are collectively referred to as field operands.) The field queue entry indicates whether the field operand was register mode. The Ibox adds an entry when it processes operands which it knows by context require an entry. The Ebox retires an entry after it has used the information in a microcode conditional branch. Very different execution microflows are required for some instructions, particularly bit field instructions, depending on whether a particular operand is register mode or specifies a memory address. In the latter case the information sent by the Ibox is a memory address, while in the first case the source and destination queue entries point to the register in the register file. See Section 8.5.15.8 for more information.

The instruction queue is part of the Ibox-Microsequencer interface. It holds information derived from the VAX instruction opcode. The Ibox adds an entry as it decodes each instruction. An entry contains the opcode, data length, the microcode dispatch address for execution, and a flag indicating whether the macroinstruction is for the Fbox. The Microsequencer references and removes an entry at the start of execution of each VAX instruction. It uses the dispatch address to fetch the first microword of the macroinstruction execution microflow. At the same time it passes the opcode, data length, and the Fbox execution flag to the Ebox. The Ebox adds an entry to the retire queue at that time. That entry is simply the Fbox execution flag (except if the Fbox is disabled, see Section 8.5.15.7). See Section 9.2.3.3.4 for more on the instruction queue.

8.4.5 Other Registers and States

The Ebox contains several special purpose registers, the SC, VA, and Q registers, and the PSL.

The SC register holds a shift count for use in some shift operations.

The VA register can hold a virtual address or a microcode temporary value. The VA register is directly readable by the Mbox and is the address source for all Ebox initiated memory operations. The VA register is loaded directly from the ALU output.

The PSL is the VAX architecture program status longword register. It is loaded from E_BUS%WBUS_L<31:0> and can be used as a source operand by the ALU or shifter. Its bits are used in many places in the Ebox and elsewhere in the CPU where required by the VAX architecture.

The Q register is loaded from the output of the shifter. It holds shifter results for later use.

8.4.6 Ebox Memory Access

Through the mechanism of the source queue and the destination queue, the Ibox initiates most memory accesses for the Ebox. In certain cases the Ebox must carry out memory accesses on its own. The MRQ field of the microword specifies the Mbox operation. The virtual or physical address is provided from the VA register. If the VA is being updated in this microword, the address is bypassed directly from the output of the ALU. For writes, the data is taken from E_BUS%WBUS_L<31:0>, so it can be the output of the shifter or the ALU. For reads, the DST field of the microword specifies the register file entry which is to receive the data. This register must be a GPR or a working register.

8.4.7 CPU Control Functions

Most control functions are invoked through one of the MISC fields, but some of the MRQ field functions are Mbox control functions or miscellaneous control functions rather than memory access commands. The control functions generally act to reset a function unit (Fbox, Ibox, or Mbox), synchronize Ebox operation with a function unit, or restart semiautonomous operation of the Mbox or Ibox when either of them has stopped for some reason.

S3 S4 S5

FETCH EXEC STORE

W1

W1

8.4.8 Ebox Pipeline

Execution of microwords in the Ebox is pipelined with three pipe stages (S3..S5). These stages are shown in Figure 8–1. In the first stage (S3), the **E_BUS%ABUS_L<31:0>** and **E_BUS%BBUS_L<31:0>** sources are fetched or prepared. In the second (S4) the ALU and shifter operate on the data. In the third (S5) the result is written into the register file or to some other destination. Stages S3 and S4 can stall for various reasons. Stage S5 cannot stall. Once a particular microword's execution has advanced into S5, it is going to complete. Various stalls occur in S4 in order to ensure that a particular microword's effects do not change any architectually visible state (e.g., GPRs, PSL) before proper completion without memory management faults is guaranteed.

The Microsequencer fetches the microword and delivers it to the Ebox in S3. If the Ebox's S3 stage is stalled, the Microsequencer's S2 activity is stalled as well. See Chapter 9 for more detail.

Even though the operand fetch, function execution, and result store take place in different cycles, the microword specifies the operation as if it all took place in one cycle. The Ebox has bypass paths which allow a microword to use a register as a source even it it is updated by one of the two preceding microwords. For example, if the immediately preceding microword updates W1 in the register file and the current microword specifies W1 as a source to the ALU, the Ebox hardware detects the condition and muxes the data into the staging latch before the ALU at the same time as it forwards the data to the latch which sources **E_BUS%WBUS_L<31:0>** in stage S5.

Bypass paths are only implemented where performance considerations warrant. Also bypassing isn't the solution to every problem pipelining introduces. For example, after the PSL is updated the microcode allows 2 cycles before a microword specifying SEQ.MUX/LAST.CYCLE or SEQ.MUX/LAST.CYCLE.OVERFLOW because the PSL is not actually updated until S5. The Microsequencer uses the FPD, T, and TP bits in the PSL to determine the proper new microflow dispatch. It would make the decision based on old PSL information if the microcode didn't allow the 2 cycles.

One place where the effect of pipelining is particularly apparent is in microcode conditional branches. For example, a microcode branch based on **E_BUS%BBUS_L<31:0>** data must immediately follow the microword which sources the relevant data onto **E_BUS%BBUS_L<31:0>**. Similarly, a microcode branch based on the ALU condition codes must be the second microword after the one which specified the ALU operation. See Chapter 9 for more detail on microcode branches.

8.4.9 Pipeline Stalls

The Ebox pipeline is controlled by the stall and fault logic. This function unit supplies stall signals which are used to gate clocking of control and data latches in each stage. It also controls insertion of effective no-ops into S4 when S3 is stalled and into S5 when S4 is stalled.

The Ebox pipeline stalls in S3 when it is accessing a source operand in the register file or the source queue which is not valid. Many register file entries have a valid bit associated with them. A register file entry is not valid, and its valid bit is not set, if a memory read has been initiated for that entry and hasn't yet completed. A source queue entry is not valid if the Ibox hasn't added that entry yet.

The Ebox stalls in S4 if the current destination queue entry is not valid and the microword in S4 references a destination queue entry. A destination queue entry is not valid if the Ibox hasn't added that entry yet.

The Ebox stalls in S4 if the current destination queue entry is valid but specifies a memory destination for the data and the current PA queue entry is not valid. A PA queue entry is not valid if the Mbox hasn't added that entry yet.

The Ebox stalls in S4 if the microword in S4 requests a memory operation and the Mbox is already working on an Ebox initiated memory operation (that is, the previous request is still in the EM_LATCH).

The Ebox stalls in S4 if the microword in S4 synchronizes with the branch queue and the branch queue entry is not valid. A branch queue entry is not valid if the Ibox hasn't added that entry yet.

The Ebox stalls in S4 if the current retire queue entry specifies that an Fbox instruction must retire before the instruction associated with the microword in S4 and the Ebox is requesting the use of the RMUX to store result data. (The Ebox requests the use of the RMUX if the microword in S4 specifies anything other than NONE in the DST field.)

If the Ebox stalls in S3, the S4 and S5 stages of the pipeline can continue execution. If S4 doesn't stall when S3 does, then an effective no-op is inserted into S4 after the current S4 operation advances into S5. The no-op is necessary so that the stalled S3 microword isn't advanced to S4 and S5 while an S3 stall is in effect. See Section 8.5.20 for more detail.

If the Ebox stalls in S4 then S3 stalls as well. (Microwords can't pass each other in the pipeline.) During S4 stalls, an effective no-op is inserted into S5 after the operation in S5 completes. This is necessary so that the operation in S4 isn't advanced into S5 while an S4 stall is in effect. See Section 8.5.20 for more detail.

In any cycle that the Ibox has not made a microstore dispatch address available to the Microsequencer and a dispatch is needed (i.e., during the last cycle of any microflow), the microsequencer fetches the STALL microword. This microword specifies no Ebox operation and can't cause a stall anywhere in the pipeline (although it does specify SEQ.MUX/LAST.CYCLE). This allows the microwords already in the pipeline to continue even when the Ibox is temporarily unable to supply new instruction execution dispatches. See Chapter 9 for more detail.

A microcode loop which repeatedly accesses the field queue until the current field queue entry becomes valid is also very much like a stall, though the stall logic is not actually involved. This condition is referred to as a field queue stall. In this situation, the Ebox pipeline advances in each cycle (unless the microword in S4 is stalled also). However, the same microword is fetched out of the control store in every cycle. In typical microcode usage of the field queue conditional branch, this microword will not alter any state in S4 or S5. See Section 8.5.15.8 for more detail.

8.4.10 Microtraps, Exceptions, and Interrupts

The Ebox and Microsequencer together coordinate the handling of exceptions and interrupts. Most interrupts and some exceptions are handled by Microsequencer dispatching to a microcode exception handler routine at the end of the current VAX instruction. These dispatches do not affect the execution of microwords already in the pipeline. Other exceptions cause a microtrap. In a microtrap the Microsequencer signals the Ebox to cause stages S3, S4, and S5 of the Ebox control pipeline to be flushed. It also signals the Ebox to flush the retire queue. (Flushing of the other Ibox-to-Ebox queues, the Fbox pipeline, and the specifier queue in the Mbox is done by microcode, except in the case of a branch misprediction.) At the same time the Microsequencer fetches a new microword from a special dispatch address in the control store based on the particular microtrap condition. This microflow handles any other necessary state flushing. Because a microtrap affects microwords already in the pipeline, the Ebox delays handling most traps until the microword which incurred the fault has reached S4. The microtrap is taken at the time that microword would normally have entered S5. In certain cases, Ebox stalls delay a microtrap until the stall is ended. The purpose of this is to ensure that operations which are part of a preceding VAX instruction are allowed to complete properly.

Most of the microtraps which the Ebox delays until S4 are due to Ibox-initiated memory operations which had an access or translation fault. Faults due to Ibox-initiated reads are detected by the Ebox when it accesses a valid MD register from the register file, and the fault bit associated with that MD is set. Each MD register has a fault bit which is set by the Ibox or the Mbox when a fault occurs in the memory reads necessary to fetch the source data. When the Ebox accesses an MD register with its fault bit set in S3, it carries that fault status down the pipeline into S4.

All faults detected in S3 are piped to S4 before they cause a microtrap. Faults detected in S4 or piped to S4 will cause a microtrap only if the Ebox is next to retire a macroinstruction. Otherwise they are delayed until the Fbox retires an instruction and the retire queue entry indicates the Ebox.

Fault status signals are sent by the Ibox for entries in the instruction queue, source queue, field queue, destination queue, and branch queue. Entries in the PA queue have fault bits. The Ebox detects a fault when it accesses a PA queue entry with its fault bit set or when it finds the instruction queue, source queue, field queue, destination queue, or branch queue empty and one of the fault status signals from the Ibox asserted. In the case of the instruction queue, the fault is detected in S2 and carried into S3 only when there is no S3 stall. In the case of the source queue and field queue, the faults are detected in S3. Instruction queue, source queue, and field queue related faults are carried down the pipeline until they reach S4, where they cause a microtrap once the Ebox is next to retire a macroinstruction.

Faults encountered in Ebox-initiated memory operations cause the Microsequencer to trap immediately. Ebox memory accesses begin in S5 so these traps cannot affect microwords from preceding VAX instructions. It is up to microcode to make sure that the last Ebox memory access has completed properly before the Microsequencer dispatches to another VAX instruction execution microflow.

Hardware errors are essentially handled in the same way as faults. See Section 8.5.19.

8.5 Ebox Detailed Functional Description

8.5.1 Register File

The register file has 4 distinct groups of registers: MD (memory data), GPR, Wn (working registers), and CPUSTATE registers. There are a total of 37 registers in the file. There are 6 ports: 3 read ports and 3 write ports. The read ports are the A port, the B port, and the IA port. The write ports are the W port, the IW port, and the MD port. The result is UNPREDICTABLE if more than one write to the same location occurs at the same time. Section 8.5.1.4 explains why this never happens.

8.5.1.1 Register Groups

The MD registers are only written by the Ibox directly or by the Mbox in completing an Ibox-initiated memory read. They are only read by the Ebox, and only accessed using a pointer from the source queue. There are 6 MD registers, MD0-MD5.

The GPRs are all of the VAX general purpose registers, except R15 (PC). These are read and written by the Ebox in the course of instruction execution. The Mbox writes them to complete an Ebox-initiated memory read. The Ibox also reads and writes them. It reads them as it processes operand specifiers which use a GPR in an address calculation. It writes them as it processes autoincrement and autodecrement operand specifiers, and in unwinding the RLOG. There are 15 GPRs, R0-R14 (R14 is often referred to as SP).

Writes to GPRs can depend on the DL (data length) register. If the L field of the microword which caused the write specifies LONG , the full longword is written. If the microword specifies L/LEN(DL), only the appropriate bytes are written. The following table shows which bytes are written in all cases.

Table 8–2: GPR Write Length

		Write Byte ?			
DL Register	**L Field of Microword**	**3**	**2**	**1**	**0**
X	LONG	Y	Y	Y	Y
BYTE	LEN(DL)	N	N	N	Y
WORD	LEN(DL)	N	N	Y	Y
LONGWORD	LEN(DL)	Y	Y	Y	Y
QUADWORD	LEN(DL)	Y	Y	Y	Y

X means don't care

The Wn registers are used by microcode for temporary storage and to receive memory read data. They are only read by the Ebox using the A or B fields of the microword. They can be written by the Ebox, Mbox, or Ibox. The Mbox writes them in completing an Ebox memory operation. The Ibox only writes them when completing an Ebox-initiated read of an Ibox IPR. There are 6 Wn registers, W0-W5.

The CPUSTATE registers are used by the microcode to hold elements of architectural state. They are read and written only by the Ebox. There are 10 CPUSTATE registers: KSP, ESP, SSP, USP, ISP, ASTLVL, SCBB, PCBB, SAVEPC, and SAVEPSL.

8.5.1.2 Access Ports

The A port and B port of the register file are read ports which can supply data to **E_BUS%ABUS_L<31:0>** and **E_BUS%BBUS_L<31:0>**, respectively. These two ports are accessed in S3. The address can be supplied directly from the A and B fields of the microword or indirectly through the source queue. Source queue addressing is specified in the A and/or B microword fields. The A port can read any register in the file; the B port can read any register in the file except a CPUSTATE.

The W port is the write port connected to **E_BUS%WBUS_L<31:0>**. It receives a result from the Ebox or Fbox in S5. It can write to the GPRs, CPUSTATEs, and Wn registers. The address can be supplied directly by the microword in the DST field or (for GPRs only) indirectly through the destination queue. Destination queue addressing is used when the microword specifies DST/DST or when the Fbox writes a result to a GPR.

NOTE

> When the Ebox initiates a memory read by sending a request to the Mbox, it specifies the register which will receive the memory data in the DST field of the microword. This has the sides effect, when the microword is in S5, of writing that register with the value on **E_BUS%WBUS_L<31:0>**. Normally this register is written by the Mbox after this, before the particular register is read again. However, an exception can prevent the Mbox write and leave the register containing effectively garbage data.

The IA port is a read port used by the Ibox to read GPRs for use in general address calculation and for autoincrement and autodecrement operand specifier processing. It can only read the GPRs. The address is supplied by the Ibox.

The IW port is a write port used by the Ibox. It can write to the GPRs, the MD registers, and the Wn registers. The Ibox writes GPRs when it processes autoincrement and autodecrement operand specifiers and when unwinding the RLOG. It writes MD registers when operand specifier decoding requires passing a value (such as an address) to the Ebox. The Ibox writes the Wn registers only when responding to an Ebox-initiated IPR read. The address is supplied by the Ibox.

The MD port is used by the Mbox to write memory or IPR read data into Wn registers, MD registers, and GPRs. The Mbox writes MD registers to complete Ibox-initiated reads. It writes Wn registers or GPRs to complete Ebox-initiated reads. The register file address is supplied by the Mbox. (The Mbox received the register file address when the memory operation was initiated.)

8.5.1.3 Register File Bypass Paths

The Ebox implements bypass for data being written into the register file or scheduled to be written into the register file further down the pipeline. Two techniques are employed: actual bypass datapaths and flow-thru bypass. Actual bypass paths are datapaths and drivers which directly drive the data onto **E_BUS%ABUS_L<31:0>** or **E_BUS%BBUS_L<31:0>**. The register file **E_BUS%ABUS_L<31:0>** or **E_BUS%BBUS_L<31:0>** drivers are automatically disabled when bypassed data is driven. Flow-thru bypass is the technique in which a write to the register file occurs early in the cycle, well before the read. This way, reads see the result of writes which occur in the same cycle. This technique can only be used when the write data is available early enough and is scheduled to be written in that cycle. (For example, bypass of S4 Ebox results to **E_BUS%ABUS_L<31:0>**

or **E_BUS%BBUS_L<31:0>** can't be done with flow-thru bypass because the register file write isn't supposed to happen until S5.)

See Section 8.5.8 for a description of bypassing of Ebox or Fbox result data from S4 or S5.

The register file has actual bypass paths for bypassing IW port writes to **E_BUS%ABUS_L<31:0>** and **E_BUS%BBUS_L<31:0>**. The IW port write occurs too late in the cycle for flow-thru bypass to be used.

NOTE

> IW port bypass is necessary for the NVAX CPU to correctly handle some sequences of operand specifier decoding. Here is one example. (To understand this example, the reader may need to know things which haven't been explained before this point in this specification.) Assume the CPU has to execute the following sequence of macroinstructions:
>
> ADDL2 R0,(R0)+
> ADDL2 R0,R1
>
> If the Ibox is executing far enough ahead of the Ebox and the read of memory data at (R0) takes a long time (as it would if it neither the Pcache nor the Bcache contains the data), then at some point the Ebox is stalled waiting for that data to arrive in an MD and the source and destination queues contain all the entries generated by the two ADDL2 instructions. The Ebox microword which executes ADDL2 is:
>
> A/S1, B/S2, ALU/A.PLUS.B, L/LEN(DL), MISC/LOAD.PSL.CC.IIII, SEQ.MUX/LAST.CYCLE.OVERFLOW
>
> In S3 this microword accesses the first two entries in the source queue, which in this case point to R0 and some MD. The microword is stalled waiting for the memory read to complete (and the MD to become valid). The Ibox complex specifier unit (CSU) is stalled by the scoreboard unit (SBU) because it is just about to write R0+4 into the register file. For the Ebox must see the old value when it reads R0, the Ibox write to R0 must be stalled. Once the Ebox retires the source queue entry containing the pointer to R0, the Ibox knows it can write R0.
>
> In cycle N the memory data arrives and is written into the MD. This ends the S3 stall in the Ebox. The very next microword to enter S3 (in cycle N+1) is for the second ADDL2. It reads RO and R1, and must see the new (incremented) value of R0.
>
> In cycle N+1, the Ebox signals the Ibox of two source queue retires, the Ibox SBU ends the CSU's stall, and the CSU writes R0+4 on the IW port. The Ebox reads R0 in that cycle and, because of the IW port bypass, it sees the correct (autoincremented) value of R0.

When processing an autoincrement or autodecrement specifier for an address access type operand specifier, the Ibox does two sequential writes into the register file. The first writes the address into an MD register. the second writes the incremented or decremented register value back into the register. In some cases this can cause the Ebox to attempt to bypass both from the output of the RMUX in S4 and from the IW port to either or both of **E_BUS%ABUS_L<31:0>** and **E_BUS%BBUS_L<31:0>**. In these cases the bypass from the output of the RMUX overrides the IW bypass. See Section 8.5.8 for more on bypass from the output of the RMUX.

8.5.1.4 Write Collisions

The result is UNPREDICTABLE if more than one write to the same register file location occurs at the same time. To prevent this, writes to registers are controlled by certain hardware and microcode mechanisms.

The MD registers can only be written by the Ibox or the Mbox. The Ibox complex specifier unit has hardware which allocates and deallocates MD registers. The Mbox writes an MD only when returning data for an Ibox-initiated operand data read, and it writes to the particular MD specified by the Ibox. The Ibox writes an MD directly only when it knows that no outstanding reads to the same MD exist in the Mbox. Therefore, The Mbox and Ibox will never write an MD at the same time.

The GPRs can be written by the Ebox, Ibox, and Mbox. In many typical instruction execution situations, the Ebox never writes a GPR explicitly. It only writes them through destination queue accesses. The Ibox only writes GPRs to process autoincrement or autodecrement operand specifiers, so it always reads a given GPR prior to writing it. The Ibox scoreboard unit keeps track of which GPRs have been entered into the destination queue and allows Ibox complex specifier unit reads only when there are no Ebox writes outstanding. This means the Ibox will never write a GPR at the same time as the Ebox.

When execution of a particular macroinstruction requires the Ebox to directly write GPRs, the Ibox is always stopped (the Ibox stops itself after processing the macroinstruction's operand specifiers). In these cases, microcode can write to any GPR without colliding with an Ibox write. The Mbox only writes a GPR when returning data for an Ebox-initiated Mbox operation. Microcode doesn't issue such a memory read unless it knows the Ibox is stopped, and microcode doesn't write the GPR while such an operation is outstanding.

When unwinding the RLOG, the Ibox may write GPRs. The Ebox microcode knows this may be happening because the unwind was either initiated under microcode control or as a result of a branch mispredict. In either case the Ebox microcode doesn't write GPRs while the unwind is occurring.

The Ebox, Ibox, and the Mbox can write the Wn registers. The Mbox only writes a Wn register when returning read data for an Ebox-initiated Mbox operation. The Ibox only writes Wn registers to return IPR data at the Ebox's request. Microcode never writes a Wn if there is an Mbox or Ibox operation outstanding which will write the same register.

Only the Ebox can write CPUSTATE registers, so there is no possibility of a write collision on those registers.

8.5.1.5 Valid, Fault, and Error Bits

Some of the registers in the register file have valid bits and/or fault and error bits associated with them. There is one valid bit, one fault bit, and one error bit associated with each MD register. The Wn registers each have a valid bit, but no fault or error bits.

Valid bits are used to allow synchronization with memory reads. Whenever a memory read to a Wn register is initiated, the associated valid bit is cleared. The valid bit for an MD register is cleared as a side effect of reading it, so it is already cleared when a memory read to it is initiated. (The MD valid bits are also cleared in exception cases, by MISC/RESET.CPU.) When the Mbox supplies the data, the valid bit is set. If the microword in S3 reads from an MD or Wn register whose valid bit is not set, the pipeline stalls in S3.

Fault and error bits are used to indicate that some sort of exception occurred with the memory read. Fault bits indicate memory management exceptions, while error bits indicate hardware errors. When the microword in S3 reads an MD register whose fault or error bit is set, a microtrap is scheduled for this microword. The microtrap is delayed in the pipeline as is discussed in Section 8.5.19. Fault and error bits are needed to delay Ebox detection of memory exceptions until the Ebox is processing the associated VAX instruction. A set fault or error bit indicates an Ibox or Mbox detected exception condition related to source operand specifier processing. If the Mbox was unable to complete an Ibox-initiated memory operation targeted to MD, it sets the fault or error bit. If the Ibox encountered any sort of fault or error before initiating the final memory read necessary to process an operand specifier, it sets the fault or error bit directly. In either case the Ebox will not detect the fault until it is executing the associated VAX instruction. There is no need for Wn register fault bits because microtraps due to Ebox memory reads are taken as soon as they are reported by the Mbox.

All the Wn register valid bits are set unconditionally in S3 of each new macroinstruction execution microflow. The Microsequencer signals the Ebox at start of these microflows. This is done to prevent errors from causing the pipeline to stall waiting for a condition which will never be true. If an error causes an Ebox memory read to a particular Wn register to fail to complete it leaves the valid bit cleared. If a new microflow references the same working register, it will stall. Since the memory operation will never complete, the stall will never end.

All Wn register valid bits are set unconditionally when the MISC field of the microword specifies RESET.CPU.

Wn register valid bits are normally set. A Wn register's valid bit is cleared in S4 if the microword specifies a memory read which will deliver data to that register. The bit is set when the Mbox or Ibox writes to that register. It is not altered by Ebox (A, B, or W port) accesses. The S4 clear of a Wn valid bit will cause the current S3 microword to stall if it references Wn.

All the MD valid bits are cleared when the microword MISC field specifies RESET.CPU. MD valid bits are not normally set. In normal operation, an MD register's valid bit is set when the Mbox or Ibox writes that register, and is cleared as a side effect of the Ebox reading the register.

8.5.2 Constant Generation

There are two constant generators, an extremely simple **E_BUS%ABUS_L<31:0>** constant source and a more complicated **E_BUS%BBUS_L<31:0>** source. The **E_BUS%ABUS_L<31:0>** constant source is specified in the A field of the microword. It can produce the following longword constants: 0, 1. To source these constants to **E_BUS%ABUS_L<31:0>**, the microword specifies K0 or K1,, respectively, in the A field.

The **E_BUS%BBUS_L<31:0>** constant generator builds a longword constant by placing a byte value in one of the four byte positions in the longword. The POS and CONST fields of the microword specify the value. The CONST field contains a byte value, while the POS field specifies the byte in the longword in which the value appears. The other bytes are zero. It is as if the POS field specified a left shift with zero fill of the CONST value.

The POS and CONST fields are part of the constant generation variant of the microword. In this variant the VAL and B fields of the standard format microword, or the MISC2, DISABLE.RETIRE, and B fields of the special format, are replaced by the POS and CONST fields. In the constant generation variant, **E_BUS%BBUS_L<31:0>** receives the constant so the B field is unnecessary. Also, the shifter uses the SC register for the shift amount so the VAL field is not needed (put another way, VAL/0 is effectively specified by the constant generation variant). Similarly, MISC2/NOP and DISABLE.RETIRE/NO are effectively specified by constant generation variant microwords.

Under control of the MISC field, the E_BUS%BBUS_L<31:0> constant generator can also provide a constant in which the low order 10 bits are specified by microcode and the high order 22 bits are all zero. This mode of constant generation occurs when the MISC field specifies CONST.10.BIT. In this case the 10 bit constant is sourced from the CONST.10 field of the microword. (The CONST.10 field is formed by concatenating the two-bit POS field with the 8-bit CONST field, with the POS field more significant.) The microword format must be the constant generation variant, if MISC/CONST.10.BIT is specified.

The E_BUS%BBUS_L<31:0> constant generator can also provide the constant 0000FFFF#16. It is produced when the B field of the microword specifies K.FFFF.

8.5.3 The ALU

The ALU is a 32-bit function unit capable of arithmetic and logical operations. Its inputs are E_BUS%ABUS_L<31:0> and E_BUS%BBUS_L<31:0>. Its output drives E_ALU%RESULT_H<31:0> which can be muxed onto E_BUS%WBUS_L<31:0> and is directly connected to the VA register (see Section 8.5.6). It also produces condition codes (ALU<C>, ALU<N>, ALU<V>, ALU<Z>) based on the results of its operation. The ALU condition codes are data length dependent, with the data length coming from the DL register or defaulting to longword depending on the microword L field. The ALU operation is specified by the ALU field of the microword.

The following table shows the ALU operations by name, and gives a description of each operation.

Table 8–3: ALU Operations

ALU Operation Name	Operation Description
PASS.A	E_ALU%RESULT_H <-- A
PASS.B	E_ALU%RESULT_H <-- B
A.AND.B	E_ALU%RESULT_H <-- A .AND. B
A.AND.NOT.B	E_ALU%RESULT_H <-- A .AND. (.NOT. B)
A.OR.B	E_ALU%RESULT_H <-- A .OR. B
A.XOR.B	E_ALU%RESULT_H <-- A .XOR. B
NOT.A.AND.B	E_ALU%RESULT_H <-- (.NOT. A) AND B
A.PLUS.1	E_ALU%RESULT_H <-- A + 1
A.PLUS.B	E_ALU%RESULT_H <-- A + B
A.PLUS.B.PLUS.1	E_ALU%RESULT_H <-- A + B + 1
B.MINUS.A	E_ALU%RESULT_H <-- B - A = B + (.NOT. A) + 1
A.MINUS.B	E_ALU%RESULT_H <-- A - B = A + (.NOT. B) + 1
A.MINUS.B.MINUS.1	E_ALU%RESULT_H <-- A - B - 1 = A + (.NOT. B)
A.MINUS.1	E_ALU%RESULT_H <-- A - 1
A.PLUS.4	E_ALU%RESULT_H <-- A + 4
A.MINUS.4	E_ALU%RESULT_H <-- A - 4

Table 8–3 (Cont.): ALU Operations

ALU Operation Name	Operation Description
NEG.B	E_ALU%RESULT_H <– –B (minus B)
NOT.B	E_ALU%RESULT_H <– .NOT. B (ones complement of B)
SMUL.STEP	E_ALU%RESULT_H <– A .SMUL. B (Q register is affected, see text)
UDIV.STEP	E_ALU%RESULT_H <– A .UDIV. B (Q register is affected, see text)

The following signals are used in functional descriptions below:

- E_ALU%RESULT_H<N> is the nth bit of the ALU result.
- E_ALU%CI_H<N> is the nth carry-in bit in the ALU. It is the carry **into** the nth bit slice. The carry-in to the ALU is E_ALU%CI_H<0>, while the carry out for longword data length is E_ALU%CI_H<32>.

8.5.3.1 ALU Condition Codes

The four condition codes calculated by the ALU are:

- **ALU<V>—Integer Overflow**
  This bit indicates an integer overflow from the operation. It is the XOR of the carry in to the most significant bit with the carry out of the same bit. The calculation depends on the data length in effect for the operation. It is E_ALU%CI_H<N> XOR. E_ALU%CI_H<N+1> where n is 7, 15, or 31 for byte, word, or longword data length, respectively.
- **ALU<C>—Carry Out**
  This bit is the carry out from the operation. It is E_ALU%CI_H<8>, E_ALU%CI_H<16>, or E_ALU%CI_H<32> for byte, word, or longword data length, respectively.
- **ALU<Z>—Zero**
  This bit indicates that the ALU result was zero. It is the logical NOR of E_ALU%RESULT_H<7:0>, E_ALU%RESULT_H<15:0>, or E_ALU%RESULT_H<31:0> for byte, word, or longword data length, respectively.
- **ALU<N>—Negative**
  This bit indicates that the ALU result was negative. It is simply E_ALU%RESULT_H<7>, E_ALU%RESULT_H<15>, or E_ALU%RESULT_H<31> for byte, word, or longword data lengths, respectively. length, respectively.

For logical and PASS operations the ALU<C> and ALU<V> condition code bits are always zero.

The ALU condition codes are available on the microtest bus and can be used to update the PSL. If the microword following the one setting the ALU condition codes is stalled, the Ebox control logic holds the ALU condition code bits constant until the microword branching on them is ready to use them. The effect is the same as if no stall had occurred. See Section 8.5.14 and Chapter 9 for more about the microtest bus and see Section 8.5.5 and Section 8.5.10.1 for more detail on setting PSL condition code bits.

If the ALU operation is SMUL or UDIV, the ALU condition codes correspond to the ALU result before the one-bit shift is done on the result.

8.5.3.2 SMUL Step Definition

The signed multiplication step is used to implement the sequential add and shift multiplication algorithm. It allows microcode to implement byte, word, and longword multiplication of two operands. The SMUL step uses the single bit left or right shifter at the output of the ALU, the Q register, and two microcode working registers.

The operation of a single SMUL step is described in Figure 8–2. The proper number of SMUL steps is controlled by the microcode and depends upon the data length of the operation.

The SMUL step operation selects the ALU operation (either PASS.A or A.PLUS.B) based on the least significant bit of the Q register. However the Q register must not have been loaded by the previous microword unless that microword specified an SMUL step. This is because that bit of the Q register is not ready in time to control the ALU operation if the Q register was loaded from the output of the shifter in the previous cycle.

Figure 8–2: SMUL Step Operation

```
At start:
   Q register   = multiplier
   Wb           = multiplicand
   Wa           = 0

Operation:  For Wa <-- Wa .SMUL. Wb

   o  If Q<0> = 1

      THEN E_ALU%RESULT_H<31:0> <-- Wa + Wb (Partial Product + Multiplicand)
      ELSE E_ALU%RESULT_H<31:0> <-- Wa      (Partial Product)

   o  WBUS<31:0> <-- (E_ALU%RESULT_H<31> .XOR. E_ALU%CI_H<31> .XOR. E_ALU%CI_H<32>) ' E_ALU%RESULT_H<31:1

   o  Q<31:0> <-- E_ALU%RESULT_H<0> ' Q<31:1>

At end:
   Wa ' Q = product

   NOTE: E_ALU%RESULT_H is the value of the ALU before the single-bit shift.

Description: The lsb of the Q register is tested for a 0 or 1.  If Q<0> EQL 0, then
             the partial product is passed through the ALU unmodified.  If Q<0> EQL 1, then
             partial product and the multiplicand are added together.  Then the output of the
             ALU and the Q register is shifted right one bit. The shift into the msb of WBUS is
             the exclusive-or of the ALU's output sign and the arithmetic overflow out of the ALU
             (arithmetic overflow is the exclusive-or of the carry-in and carry-out of the msb).
             The shift into the msb of Q comes from E_ALU%RESULT_H<0>.
```

8.5.3.3 UDIV Step Definition

The unsigned division step is used to implement the sequential shift and subtract non-restoring division algorithm. It allows microcode to implement byte, word, and longword division of two operands, and to produce the remainder. The UDIV step uses the single bit left or right shifter at the output of the ALU, the Q register, and two microcode working registers.

The operation of a single UDIV step is described in Figure 8–3. The proper number of UDIV steps is controlled by the microcode and depends upon the data length of the operation. The unsigned divide algorithm using the UDIV step requires microcode to shift the remainder one bit to the right after the final UDIV step.

Figure 8–3: UDIV Step Operation

```
Note that non-restoring division use the fact that
2 * (Partial Remainder - Divisor + Divisor) - Divisor =
2 * (Partial Remainder - Divisor) + Divisor

At start:
   Q register  = dividend
   Wb          = divisor
   Wa          = 0 (except during an extended divide when
                  Wa contains the high-order longword of
                  the dividend)

Operation:  For Wa <-- Wa .UDIV. Wb

This operation results in the Q register containing the quotient and
Wa containing the remainder.

   o  If ALU_CC.C EQL 1

      THEN E_ALU%RESULT_H <-- Wa - Wb (Partial Remainder/Quotient - Divisor)
      ELSE E_ALU%RESULT_H <-- Wa + Wb (Partial Remainder/Quotient - Divisor)

   o  WBUS<31:0> <-- E_ALU%RESULT_H<30:0> ' Q<31>

   o  Q<31:0> <-- Q<30:0> ' (.NOT.E_ALU%RESULT_H<31>)

   o  ALU_CC.C <-- E_ALU%CI_H<32>

At end:
   Q register = quotient
   Wa         = remainder


   NOTE: E_ALU%RESULT_H is the value of the ALU before the single-bit shift.

Description: ALU_CC.C is tested for a 0 or 1. If ALU_CC.C EQL 1, then Wb is subtracted from Wa.
             If ALU_CC.C EQL 0, then the Wa and Wb are added together. The output of the ALU is
             then rotated to the left one-bit and driven onto the WBUS with WBUS<0> being driven
             by Q<31>.  Additionally, the Q register is rotated left one bit with the
             complement of the bit shifted out of the ALU result becoming Q<0>.  The new
             ALU_CC.C condition flag comes from the carry out of the ALU (or E_ALU%CI_H<32> here).
```

8.5.4 The Shifter

The shifter is a right shift network with 64-bits of input and 32-bits of output. The input is **E_BUS%ABUS_L<31:0>** and **E_BUS%BBUS_L<31:0>** concatenated to form a 64-bit word with **E_BUS%ABUS_L<31:0>** in the more significant longword. The output is **E_SHF%SHF_RESULT_H<31:0>** which can be muxed onto **E_BUS%WBUS_L<31:0>** and is directly connected to the Q register (see Section 8.5.7).

The shifter produces two condition code bits, SHF<N> and SHF<Z>. These are available on the microtest bus and can be used to update the PSL. See Chapter 9 for more about the microtest bus and see Section 8.5.5 and Section 8.5.10.1 for more detail on setting PSL condition code bits.

The shifter shifts its input right by 0 to 32 bits. A shift amount of 0 selects the **E_BUS%BBUS_L<31:0>** and a shift amount of 32 selects **E_BUS%ABUS_L<31:0>**. The equivalent of a left shift of N is accomplished by shifting left justified data (32-N) to the right.

The shift operation is specified in the SHF field of the microword. The following table shows the shifter operations by name and gives a description of each operation. If the microword is in the special format, the shifter function defaults to NOP since the SHF field is not present.

Table 8–4: Shifter Operations

Shifter Operation Name	Operation Description
NOP	E_SHF%SHF_RESULT_H <-- UNPREDICTABLE
PASS.A	E_SHF%SHF_RESULT_H <-- A
PASS.B	E_SHF%SHF_RESULT_H <-- B
PASS.Z	E_SHF%SHF_RESULT_H <-- 0
LEFT.DOUBLE	E_SHF%SHF_RESULT_H <-- A'B rsh 32 - count (the effect is LSH count)
LEFT.SINGLE	E_SHF%SHF_RESULT_H <-- A'0 rsh 32 - count (the effect is LSH count)
RIGHT.DOUBLE	E_SHF%SHF_RESULT_H <-- A'B rsh count
RIGHT.SINGLE	E_SHF%SHF_RESULT_H <-- 0'B rsh count

' is the bitwise concatenation operator.

For the SHF/LEFT.SINGLE and SHF/RIGHT.SINGLE operations the shifter masks off **E_BUS%BBUS_L<31:0>** or **E_BUS%ABUS_L<31:0>**, respectively. This guarantees that the bits shifted into the result are 0.

The shift amount comes from the VAL field of the microword or from the SC register. The SC register is the source of the shift amount if the VAL field is 0 or if the VAL field is not present because the microword is in the constant generation variant format.

The SC register can specify an actual shift amount in the range of 0 to 31, and the VAL field can specify a shift amount of 1 to 31 (0 in VAL implies SC contains the shift amount).

Neither the SC nor the VAL field can specify a shift of 32. However, since the SHF/LEFT.SINGLE and SHF/LEFT.DOUBLE operations differ from the corresponding right shift operations only in that the actual shift amount is the amount in the SC register or VAL field subtracted from 32 (32-N), the shifter shifts right by 32 when a left shift of 0 is specified.

8.5.4.1 Shifter Condition Codes

The shifter condition codes are not dependent on the instruction data length. They are calculated always for longword data length. The two condition codes calculated by the shifter are:

- **SHF<Z> - Zero**
  This bit indicates that the shifter result was zero. It is the logical NOR of **E_SHF%SHF_RESULT_H<31:0>**.

- **SHF<N> - Negative**
  This bit indicates that the shifter result was negative. It is simply **E_SHF%SHF_RESULT_H<31>**.

The shifter condition codes are available on the microtest bus and can be used to update the PSL. If the microword following the one setting the shifter condition codes is stalled, the Ebox control logic holds the shifter condition code bits constant until the microword branching on them is ready to use them. The effect is the same as if no stall had occurred. See Section 8.5.14 and Chapter 9 for more about the microtest bus and see Section 8.5.5 and Section 8.5.10.1 for more detail on setting PSL condition code bits.

8.5.4.2 Shifter Sign

The shifter sign, SHF<N>, is saved after each shifter operation including pass operations. A constant based on this saved value is available as an input to **E_BUS%ABUS_L<31:0>**. It is accessed by specifying SHIFT.SIGN in the A field of the microword. The constant is 0 or FFFFFFFF#16 for Saved-SHF<N> equal 0 or 1, respectively. Saved-SHF<N> is updated after each shifter operation and is held in each shifter NOP cycle. If microword N specifies a shifter operation, and microword N+1 sources this constant, the new value is used to form the constant. However, the Saved-SHF<N> may be destroyed by executing a special format microword. The bit is UNPREDICTABLE after executing such a microword.

8.5.5 RMUX and E_BUS%WBUS_L

The RMUX coordinates Fbox and Ebox result storage and macroinstruction retiring. It is a large selector which selects the source of Ebox memory requests and the source of the next **E_BUS%WBUS_L<31:0>** data and associated information. The RMUX selection takes place in S4, as does the driving of the memory request to the Mbox. The new **E_BUS%WBUS_L<31:0>** data is not used until S5.

The RMUX is controlled by the retire queue. See Section 8.5.15.7 for detail on the retire queue. The retire queue output is a status which indicates whether the next macroinstruction to retire is being executed in the Ebox or the Fbox. Based on this status, the RMUX selects one of the two boxes to drive **E_BUS%WBUS_L<31:0>** and to drive the memory request signals. The box not selected will stall if it has need to drive **E_BUS%WBUS_L<31:0>** or memory request signals. The retire queue read pointer is not advanced, and therefore the RMUX selection cannot change, until the currently selected box indicates that its macroinstruction is to be retired (except that the retire queue read pointer is not advanced when MISC1/RETIRE.INSTRUCTION is specified).

NOTE

> The Ebox stalls when the microword does not specify NONE in the DST field and the retire queue selects the Fbox. It does not stall if the microword specifies DST/NONE, even if the same microword specifies a memory request. This is the reason for the microcode restriction that any microword specifying a memory operation must also specify DST/WBUS or something other than none in the DST field. See Section 8.5.27.15.

The source (Ebox or Fbox) indicated by the retire queue is always selected to drive the RMUX. If the Ebox is selected, the W field of the microword in S4 selects either the ALU or the shifter as the source of the RMUX. (Note that **E_BUS%WBUS_L<31:0>** is always driven, even if the Ebox specifies DST/NONE.)

8.5.5.1 RMUX Produced Memory Request Signals

The RMUX produced memory request signals are:

- a memory command,
- a status indicating a destination queue indirect memory store,
- a tag giving a register file address in case a memory read is specified,
- and the data length for the operation.

This information is processed slightly further in the Ebox's Mbox interface logic to produce a memory request about halfway through S4. See Section 8.5.17 for more on Ebox memory requests.

The only memory operation the Fbox can initiate is a destination queue indirect store (a memory store). If the Fbox is selected as the RMUX source, the memory request information comes from the Fbox and the destination queue. The destination queue is only accessed if the Fbox requests it. If it does not request a destination queue access, the memory information output by the RMUX indicates no operation. The Fbox also provides the data length if there is a store.

If the Ebox is selected as the RMUX source, the memory request information comes from the microword. However, the DST field can cause a memory store request if it specifies a destination queue indirect store. The data length is from the DL register unless the microword L field overrides it to longword. The register file address for memory reads always comes from the DST field.

8.5.5.2 RMUX Produced E_BUS%WBUS_L Related Information

E_BUS%WBUS_L<31:0> carries result data from the Ebox and Fbox and is the only path by which macroinstruction results are written to memory or registers. The RMUX produced **E_BUS%WBUS_L<31:0>** related information is:

- the **E_BUS%WBUS_L<31:0>** (a longword of data),
- the **E_BUS%WBUS_L<31:0>** destination address or specification,
- the data length associated with **E_BUS%WBUS_L<31:0>**,
- the S5 condition codes,
- and an indication of which condition code map is to be used.

The above control information is driven into S5 provided there is not an S4 stall. If there is an S4 stall, S5 control information specifying no operation is driven into S5 instead.

If the Fbox is selected, **E_BUS%WBUS_L<31:0>** data comes from the Fbox. The **E_BUS%WBUS_L<31:0>** destination address comes from the destination queue. The condition code bits and map specification come from the Fbox. The Fbox sets map specification code to specify no change of the condition code bits, except in the last cycle of an instruction retire when the map specifier specifies a particular condition code update. See Section 8.5.10.1.1 for more detail on condition code alteration.

If the Ebox is selected, **E_BUS%WBUS_L<31:0>** data comes either from **E_ALU%RESULT_H<31:0>** or **E_SHF%SHF_RESULT_H<31:0>**. The condition codes come from the same source (ALU or shifter). Since the shifter only produces N and Z condition code bits, the RMUX substitutes 0 for S5 C and V bits if the shifter is selected. The **E_BUS%WBUS_L<31:0>** destination address comes from the DST field of the microword or from the destination queue. The status indicating whether the condition code bits are to be updated and the condition code map to be used are both decoded from the MISC field of the microword.

In S5, E_BUS%WBUS_L<31:0> drives the W port of the register file and is the input to several miscellaneous registers in the Ebox. The condition codes and the map are used to update the PSL condition code bits if the map and associated status indicate this should happen. E_BUS%WBUS_L<31:0> is also the source of write data for any memory write request which was sent by the Ebox to the Mbox in the previous cycle. In other words E_BUS%WBUS_L<31:0>, in S5, is the source of write data for the memory operation selected by the RMUX in S4.

In S5, E_BUS%WBUS_L<31:0> is zero extended according to the data length. The data length is from the DL register unless the microword L field overrides it to longword. E_BUS%WBUS_L<31:0> data is zero extended from the effective data length to longword.

8.5.6 VA Register

The 32-bit VA register is the source for the address on all Ebox memory requests, except destination queue based stores which use the current PA queue entry for an address. Unlike the entry in the PA queue, the VA register address is not yet translated (though it may be a physical address). It is a virtual address except when the memory operation doesn't require translation (as in IPR references or explicit physical memory references) or when memory management is off.

The VA register can be used to latch a temporary ALU output value without driving the ALU result onto E_BUS%WBUS_L<31:0>.

The VA register can be loaded only from the output of the ALU, E_ALU%RESULT_H<31:0>. It is loaded when the microword V field specifies to load it. The load occurs at the end of S4, even when there is an S4 stall. If a given microword specifies a memory operation in the MRQ field and loads the VA register, the new VA value will be received by the Mbox with the memory command. For more detail on Ebox-initiated memory operations, see Section 8.5.17.

NOTE

> The address for memory operations is part of the data latched in the EM_LATCH in the Mbox. This is why the Ebox can overwrite the VA value during S4 stalls even though the stall might be because the EM_LATCH is full.

The VA register is one of the possible E_BUS%ABUS_L<31:0> sources. The microword specifies VA in the A field to use it.

8.5.7 Q Register

The 32-bit Q register is closely associated with the shifter. It can be loaded directly from the shifter output without driving that data onto E_BUS%WBUS_L<31:0>. Microcode uses it to hold temporary data.

The Q register can only be loaded from the shifter output, E_SHF%SHF_RESULT_H<31:0>. It is loaded when the microword Q field specifies to load it. The load occurs at the end of S4, even when there is an S4 stall.

The Q register is one of the possible sources of both E_BUS%ABUS_L<31:0> and E_BUS%BBUS_L<31:0>. The microword specifies Q in the A or B field to use it.

The data in the Q register is shifted one bit to the left or right as a side effect of the ALU SMUL.STEP and UDIV.STEP operations. The shift is one bit to the left for UDIV.STEP and one bit to the right for SMUL.STEP.

8.5.8 Bypassing of Results

The Ebox implements bypass paths for result data from S4 or S5 to **E_BUS%ABUS_L<31:0>** or **E_BUS%BBUS_L<31:0>**. These paths allow microwords to use any register in the register file as a source of **E_BUS%ABUS_L<31:0>** or **E_BUS%BBUS_L<31:0>** even if the register has been updated by one of the two preceding microwords. The Ebox pipeline reads from the register file in S3, operates on the data in S4, and writes the register file in S5. Since adjacent microwords in the pipeline could be from entirely different macroinstruction execution microflows, it is necessary that the Ebox hardware detect and resolve cases where one microword alters a register and a subsequent microword reads that register before it is written.

NOTE

> The Fbox is one possible source of result data in S4, and any S5 operation may be a result store operation from the Fbox piped forward one stage. Bypassing of results destined for the register file from S4 or S5 works for Fbox result store operations in the Ebox pipeline in the same way as for microcode operations.

The Ebox monitors the register file addresses on the A and B ports of the register file in S3 and compares those to the RMUX register file address in S4. Whenever **E_BUS%ABUS_L<31:0>** and **E_BUS%BBUS_L<31:0>** are expecting data that is not yet in the register file, the data is steered directly from the output of the RMUX (at the end of S4).

NOTE

> The bypass path for register file entries from **E_BUS%WBUS_L<31:0>** in S5 to **E_BUS%ABUS_L<31:0>** or **E_BUS%BBUS_L<31:0>** is implemented by register file flow-thru writes. **E_BUS%WBUS_L<31:0>** data is written into the register file early in the cycle and read after the write. So reads see the result of writes from the same cycle.

The S3 A and B port addresses can come from the microword or the source queue. Similarly the RMUX address in S4 can come from the microword, the destination queue, or the Fbox. The W port address in S5 has already been determined by the RMUX in the previous cycle. The Ebox bypass path control logic compares the final S3 read addresses to the final S4 write addresses and enables the appropriate bypass path when there is a match. (As noted above, S5 to S3 register file bypass is a flow-thru path.)

Data length has an effect on bypass operations for GPRs. When a pending GPR write is to less than a full longword, only the bytes which are going to be updated are bypassed. The other bytes are read from the register file. Effectively, an independent bypass check is made for each of the following: byte 0, byte 1, and the upper word.

In the event that the W port and the RMUX update the same register, the bypass logic chooses the RMUX data as the source of **E_BUS%ABUS_L<31:0>** or **E_BUS%BBUS_L<31:0>**.

NOTE

> Note it would be possible for a value to be constructed of data from the register file, the RMUX, and the W port all at once, because of differing data lengths.

In the event that the IW port (from the Ibox) and the RMUX update the same register, the bypass logic chooses the RMUX data as the source of **E_BUS%ABUS_L<31:0>** or **E_BUS%BBUS_L<31:0>**. See Section 8.5.1.3 for more on IW port bypass.

The Q and VA registers updates are also effectively bypassed. Microcode can depend upon the new data being available to **E_BUS%ABUS_L<31:0>** and **E_BUS%BBUS_L<31:0>** when the preceding microword updated these registers. However, the Q register contents are not bypassed if the Q register was updated by a shift caused by an ALU/SMUL.STEP or ALU/UDIV.STEP ALU operation.

NOTE

> The bypass mechanisms for the VA and Q registers are based on a flow-thru latch updated in S4 (and not stalled) rather than actual bypass paths. Neither bypass is data length dependent, as writes to these registers always load the entire longword.

Bypassing for other registers and states in the Ebox generally does not make sense, and therefore is not implemented. For example, there is no bypass associated with the INT.SYS register or the PSL.

8.5.9 Result Destinations

Most of the Ebox result destinations receive their data from **E_BUS%WBUS_L<31:0>** in S5. Destinations specified in the DST field of the microword are updated in S5 from **E_BUS%WBUS_L<31:0>**. Possible **E_BUS%WBUS_L<31:0>** destinations are any register file entry, the PSL and SC registers, and the MMGT.MODE and INT.SYS special registers. More detail on the miscellaneous registers is given in the next section.

A number of special capabilities for loading registers are available through the MISC field of the microword.

- The DL (data length) register can be altered in S3, affecting the next microword but not the current one.
- The SC register can be updated directly from **E_BUS%ABUS_L<4:0>** in S4 (overriding an S5 update from the preceding microword).
- The MPU (mask processing unit, see Section 8.5.10.7) can be updated directly from **E_BUS%BBUS_L<29:16>** in S4.

8.5.10 Miscellaneous Ebox Registers and States

There are a number of states and registers in the Ebox with special purposes. Some, like the DL register, provide control information. Some provide status signals used by Microsequencer conditional branches. They also vary in how and when they are loaded.

8.5.10.1 PSL

The PSL is the VAX architecture PSL register. Its bits are used in several places within the Ebox. The Microsequencer uses a number of the bits to make dispatching decisions. Additionally the current mode is used by the Mbox and the IPL level is used by the interrupts section (see Chapter 10 for more on interrupts).

The PSL can be loaded as a longword or byte destination of **E_BUS%WBUS_L<31:0>** in S5. There are two different decodes of the DST microword field which load the PSL, DST/PSL and DST/PSL.B0. The first loads the entire PSL. The second loads only the low-order byte of the PSL.

8.5.10.1.1 Condition Code Alteration

The condition code bits of the PSL can be altered independently. This occurs when the MISC field of the microword specifies one of the six possible PSL condition code update functions. Condition code update also occurs when the Fbox retires a macroinstruction. The update occurs at the end of S5. The resulting bits can be used in the next cycle (for example, the second following microword can source the PSL).

The new condition codes are a logic function (called a map) of the current PSL condition codes and the new S5 condition codes. The S5 condition codes in any cycle were selected in the previous cycle by the RMUX from the shifter, ALU, and Fbox condition codes. The map specifier is an output of the RMUX. It is either supplied by the Ebox or the Fbox. The six different condition code update functions available through the MISC field of the microword indicate six different maps. The Fbox derives its map from the opcode of the macroinstruction it is executing.

The following tables show all the different condition code alteration maps. Table 8–5 shows the microcode specified maps used for macroinstructions executed in the Ebox. Table 8–6 shows the maps used for macroinstructions executed in the Fbox.

Table 8–5: Condition Code Alteration Maps Specified By Microcode

MISC Field Specification	Map Function
LOAD.PSL.CC.IIIP	PSL<N,Z,V> <– S5 Condition Codes <N,Z,V> PSL<C> <– PSL<C> (unchanged)
LOAD.PSL.CC.JIZJ	PSL<N> <– S5 Condition Code <N> XOR S5 Condition Code <V> PSL<Z> <– S5 Condition Code <Z> PSL<V> <– 0 PSL<C> <– NOT S5 Condition Code <C>
LOAD.PSL.CC.IIII	PSL<N,Z,V,C> <– S5 Condition Codes <N,Z,V,C>
LOAD.PSL.CC.IIIJ	PSL<N,Z,V> <– S5 Condition Codes <N,Z,V> PSL<C> <– NOT S5 Condition Code <C>
LOAD.PSL.CC.IIIP.QUAD	PSL<Z> <– PSL<Z> AND S5 Condition Code <Z> PSL<N,V> <– S5 Condition Codes <N,V> PSL<C> <– PSL<C>(unchanged)
LOAD.PSL.CC.PPJP	PSL<V> <– NOT S5 Condition Code <Z> PSL<N,Z,C> <– PSL<N,Z,C>(unchanged)

Table 8–6: Condition Code Alteration Maps Used By The Fbox

Map Specifier Value	Map Function
0	No change to the PSL condition code bits.
1 (used for MOVF, MOVD, MOVG)	PSL<N,Z> <– S5 Condition Codes <N,Z> PSL<V> <– 0 PSL<C> <– PSL<C> (unchanged)
2 (used for most floating point instructions)	PSL<N,Z> <– S5 Condition Codes <N,Z> PSL<V,C> <– 0

Table 8–6 (Cont.): Condition Code Alteration Maps Used By The Fbox

Map Specifier Value	Map Function
3 (used for MULL and some convert instructions)	PSL<N,Z,V> <– S5 Condition Codes <N,Z,V> PSL<C> <– 0

8.5.10.1.2 Trace and Trace Pending Bits

When the first microword of a macroinstruction execution microflow reaches S5, the PSL<T> bit is copied into the PSL<TP> bit. (Macroinstruction execution microflows are distinguished from other microflows by a status bit sent from the Microsequencer. See Section 8.5.14.3.) The Microsequencer receives both these bits and causes a trap fault dispatch when necessary. The Microsequencer anticipates the setting of PSL<TP> when it dispatches a macroinstruction execution microflow so that it will dispatch to the trace fault handler on the next SEQ.MUX/LAST.CYCLE or SEQ.MUX/LAST.CYCLE.OVERFLOW. (See Section 9.2.3.3.2.)

8.5.10.2 SC

The SC register is a 5-bit register which holds a shifter shift amount. The microword can specify left and right shifts of the amount in the SC register. A microword specifies this one of two ways. If the constant generation variant of the microword is used, the SC register is always the source of the shift amount. Also, the SC register is the shift amount source if the microword is not a constant generation variant and the VAL field is zero.

The SC register can be loaded in two different ways. One way is to specify DST/SC, specifying the SC as the destination of **E_BUS%WBUS_L<4:0>**. The other way is to specify MISC/LOAD.SC.FROM.A. In this case the SC register is loaded from **E_BUS%ABUS_L<4:0>**.

The **E_BUS%WBUS_L<4:0>** load into SC occurs at the end of S5. The **E_BUS%ABUS_L<4:0>** load occurs at the end of S4. In either case, the new value is not seen by the shifter until the next cycle. The shifter can use the old SC value during the current cycle. The SC control logic ensures that the following case works the same way with and without a stall on the second microword. If microword N loads the SC register off **E_BUS%WBUS_L<4:0>**, and microword N+1 shifts some data by the amount in the SC register, the data will be shifted according to the value in SC as microword N began.

If two different microwords each specify a load of the SC in the same cycle, the **E_BUS%ABUS_L<31:0>** data is loaded. This can only happen if one microword specifies DST/SC and the following microword specifies MISC/LOAD.SC.FROM.A. The more recently executed microword wins. (Note that this means the result when a stall delays the second microword is the same as if there is no stall.)

NOTE

If an Ebox pipeline abort occurs, it does not necessarily prevent the modification of the SC register by a microword in the pipeline. If a microword which would alter the SC in S5 (i.e., specifies DST/SC) enters S5 in a pipeline abort cycle, the SC is loaded despite the abort. Effectively, the SC register is UNPREDICTABLE after a pipeline abort (though if a particular case is analyzed carefully, it may be possible to determine that the SC is predictable in that case).

8.5.10.3 INT.SYS

INT.SYS is a possible **E_BUS%ABUS_L<31:0>** source and a possible **E_BUS%WBUS_L<31:0>** destination. It is microcode's interface to the interrupt section. Both as a source and as a destination, INT.SYS is a longword. For information on the format and use of the register, see Chapter 10. The register is read in S3 and written in S5.

8.5.10.4 MMGT.MODE

The MMGT.MODE register is a 2-bit **E_BUS%WBUS_L** destination. It is loaded from **E_BUS%WBUS_L<3:2>** early in S5. Its value is used in memory management probe accesses (MRQ field specifies PROBE.V.RCHK, PROBE.V.RCHK.NOFILL or PROBE.V.WCHK). The Ebox drives this mode directly to the Mbox. For more detail on Ebox-Mbox interaction see Section 8.5.17.

8.5.10.5 State Flags

There are 6 1-bit state flags: 0 through 5. Microcode can conditionally branch on these bits. They can be set and cleared by microcode, and some are cleared automatically at the start of each macroinstruction execution microflow. The state flags are used as microcode flags for loops and shared microcode paths.

The state bits are maintained in S3. If the state bits are altered in a microword, a branch based on the new state may be specified in the next microword. It is possible to set or clear state flags and branch on the previous value in the same microword.

The following table shows the microword fields and specifications used to set and reset state flags.

Table 8–7: Setting and Clearing State Flags

MISC Field in Standard Format Microword

Mnemonic	Operation
MISC/CLR.STATE.3-0	Clear State Flags 0-3
MISC/SET.STATE.0	Set State Flag 0
MISC/SET.STATE.1	Set State Flag 1
MISC/SET.STATE.2	Set State Flag 2

MISC1 Field in Special Format Microword

Mnemonic	Operation
MISC1/CLR.STATE.5-4	Clear State Flags 4 and 5
MISC1/SET.STATE.3	Set State Flag 3
MISC1/SET.STATE.4	Set State Flag 4
MISC1/SET.STATE.5	Set State Flag 5

At the start of each macroinstruction (macroinstruction and FPD dispatches in the microsequencer), in S3, state flags 0 through 3 are reset. If the first microword of the macroinstruction execution microflow sets any of the state flags, it will override the automatic reset for the particular state bit(s) specified; the others are still cleared.

The state flag bits may be selected onto the microtest bus for use in microcode branches. See Section 8.5.14 and Chapter 9 for more on microcode branches.

8.5.10.5.1 E%MACHINE_CHECK_H

If state flag 5 is 1 and state flag 4 is 0, the signal E%MACHINE_CHECK_H is asserted. This causes pin **P%MACHINE_CHECK_H** to be asserted.

8.5.10.5.2 State Flags and Pipeline Abort

The state flags are maintained in S3. If a microword which specifies to set or clear state flags enters S3, the flags are altered. Also, the automatic reset of state flags 0 through 3 at the start of a new macroinstruction execution flow occurs when the associated microword is in S3. In either of these cases, a pipeline abort (due to a microtrap) in S4 for the associated microword will not prevent the state flag modification. When microcode intends that the state flags not be altered by a specific flow if it is aborted by a microtrap, special rules must be followed.

There are two cases. If the anticipated microtrap can only occur with microword N in S3, microword N+1 can specify an alteration of a state flag and it will not happen if the microtrap occurs. If the anticipated microtrap can only occur with microword N in S4, and microword N+1 alters a state flag, that state flag will be affected even if the microtrap occurs. In this case, microword N+2 may alter a state flag and it will not happen if the microtrap occurs.

If it is not predictable whether microword N will be in S3 or S4 when the anticipated microtrap occurs, then the obvious extrapolation of the above explanation determines the result.

Here is an example case in which microword N is guaranteed to be in S3 when an anticipated microtrap occurs:

- Microcode issued an explicit memory read to a Wn register and microword N sources Wn to the E_BUS%ABUS_L<31:0> to synchronize the operation. The anticipated microtrap is associated with the memory read to Wn.

Here are some example cases in which microword N is guaranteed to be in S4 when an anticipated microtrap occurs:

- Microword N sources an MD to the E_BUS%ABUS_L<31:0> (through the source queue) to synchronize to an operand prefetch issued by the Ibox. The microtrap is associated with the operand which is to be returned to the MD.
- Microword N synchronizes to an explicit memory reference in microword N-1 by specifying MRQ/SYNC.MBOX. The microtrap is associated with the memory reference issued by microword N-1.

Microcode which intends to avoid the side effect in which state flags 0 through 3 are cleared in the first cycle of a macroinstruction microflow if a microtrap occurs may have to add a microword after the one synchronizing to the anticipated microflow before specifying SEQ.MUX/LAST.CYCLE or SEQ.MUX/LAST.CYCLE.OVERFLOW. Specifically if microword N synchronizes to an anticipated microtrap in S4 and microword N+1 specifies SEQ.MUX/LAST.CYCLE, then state flags 0 through 3 will not be cleared if the microtrap occurs. However, if microword N specifies SEQ.MUX/LAST.CYCLE, the state flags could be cleared (though it would depend on the detailed timing of the events).

8.5.10.6 DL Part of the Instruction Context Register

The DL is one field of the instruction context register. It contains the initial data length for the macroinstruction which is being executed in the Ebox. The data length is determined by the Ibox and passed to the Microsequencer in the instruction queue. The Microsequencer enters the DL into the instruction context register, along with other instruction context information. It is used by the Ebox as the default data length for each microword. Each microword specifies use of the data length in the DL or use of a data length of longword. The L field of the microword determines this. The operations affected by data length are:

- **Calculation of the ALU condition codes.**
  The four condition codes are determined according to the data length. (For example, the ALU<N> is bit <31>, <15>, or <7> for longword, word, or byte length operations, respectively.)
- **Zero Extending of E_BUS%WBUS_L<31:0> data.**
  E_BUS%WBUS_L<31:0> data is zero extended from the specified data length to longword.
- **The size of a memory operation initiated by this microword.**
  This affects all memory operations except result stores to the current PA queue entry address. (PA queue entries contain the data length used for the store operation.)
- **Register File GPR Writes.**
  GPR writes from E_BUS%WBUS_L<31:0> are gated by the data length such that only the bytes in that data length are affected by the write and others are unchanged. (Writes from the MD and IW ports to the GPRs are not affected by the DL.)

The DL field in the instruction context register can be modified by specifying DL.BYTE, DL.WORD, or DL.LONG in the MISC field of the microword. The effect is to set the DL to byte, word, or longword data length, respectively. The old DL value applies to operations in the current microword. The new DL value applies to the next microword.

See Section 8.5.14.1 for more on the instruction context register.

8.5.10.7 Mask Processing Unit

The mask processing unit (MPU) holds and processes a 14-bit value. The value is loaded from E_BUS%BBUS_L<29:16> when the microword specifies LOAD.MPU.FROM.B in the MISC field. The MPU outputs a set of bits with which the microcode can carry out an eight-way branch. They are MPU0_6<2:0> and MPU7_13<2:0>. The purpose of this is to allow microcode to quickly process bit masks in macroinstruction execution microflows for CALLG, CALLS, RET, FFC, FFS, POPR, and PUSHR.

The MPU unit loads a 14-bit value from E_BUS%BBUS_L<29:16> when the microword specifies it. This occurs in S4. The MPU evaluates the input producing the values on MPU0_6<2:0> and MPU7_13<2:0> shown in the table below. MPU0_6<2:0> depends only on mask bits <6:0> and MPU7_13<2:0> depends only on mask bits <13:7>.

Table 8–8: MPU Calculation

MPU0_6<2:0> Truth Table

Mask<6:0>							MPU0_6<2:0>
X	X	X	X	X	X	1	000

All values shown in binary. X = don't care

Table 8–8 (Cont.): MPU Calculation

MPU0_6<2:0> Truth Table

Mask<6:0>							**MPU0_6<2:0>**
X	X	X	X	X	1	0	001
X	X	X	X	1	0	0	010
X	X	X	1	0	0	0	011
X	X	1	0	0	0	0	100
X	1	0	0	0	0	0	101
1	0	0	0	0	0	0	110
0	0	0	0	0	0	0	111

MPU7_13<2:0> Truth Table

Mask<13:7>							**MPU7_13<2:0>**
X	X	X	X	X	X	1	000
X	X	X	X	X	1	0	001
X	X	X	X	1	0	0	010
X	X	X	1	0	0	0	011
X	X	1	0	0	0	0	100
X	1	0	0	0	0	0	101
1	0	0	0	0	0	0	110
0	0	0	0	0	0	0	111

All values shown in binary. X = don't care

Microcode can branch on the MPU7_13<2:0> or MPU6_0<2:0> values after they are loaded. The initial processing is done by the end of the S4 cycle which loaded the MPU. When microcode does branch on one of these values, the least significant bit which is 1 in the current mask value in the MPU is reset to 0 automatically. This occurs in S3, so that the next microword can branch on the new value of the mask. (The MPU bit clear does not occur in a cycle in which there is an S3 stall.) The MPU detects that the microword entering S3 specifies an eight-way branch on MPU7_13<2:0> or MPU6_0<2:0> by examining the **E_USQ%UTSEL_H<4:0>** and **E_USQ%UTSEL_L<4:0>** bits. If they specify a MPU branch, the appropriate bit is reset.

If a load of a new MPU mask value is simultaneous with a microcode MPU branch, the new data is loaded correctly without any side effect due to the branch. This occurs when a microword specifies LOAD.MPU.FROM.B and the immediately following microword does a branch on the previous mask value. The branch is an S3 operation of the second microword, while at the same time the load is an S4 operation of the first. (The branch outcome is guaranteed to reflect the MPU value before the load.)

8.5.11 Branch Condition Evaluator

The branch condition evaluator uses the macroinstruction opcode, the ALU condition code bits, the PSL condition code bits, and E_SHF%SHF_RESULT_H<0> to evaluate the branch condition for all macroinstruction conditional branches. The evaluation is done in every cycle but is only used if the microword specifies SYNC.BDISP.TEST.PRED in the MRQ field. The result of the evaluation is compared with the Ibox prediction for the branch. The Ibox prediction is indicated in the current branch queue entry. If the Ibox prediction was not correct, the Ebox signals the Ibox and sends a branch misprediction trap request to the Microsequencer.

The branch condition evaluation is begun late in S4 and finished early in S5. All the information needed to perform the evaluation is gathered late in S4. The PSL condition code bits used in the comparison are bypassed; they are the bits which will be latched into the PSL at the end of S4. The ALU condition code bits used are generated late in S4 and are dependent on the data length for the instruction. The shifter result bit is also generated late in S4. The opcode is available early in S4 and is used to set up the evaluation.

In S5, the result of the branch condition evaluation is compared with the Ibox prediction, and E%BCOND_RETIRE_L is asserted to tell the Ibox that a branch queue entry for a conditional branch was removed from the branch queue. If the prediction was not correct, the Ebox also asserts E%BRANCH_MISPREDICT_L which is received by the Ibox and Microsequencer. The Microsequencer forces a branch mispredict microtrap beginning in the next cycle when E%BRANCH_MISPREDICT_L is asserted. If E%BCOND_RETIRE_L is asserted and E%BRANCH_MISPREDICT_L is not, the Ibox releases the resource which is holding the alternate PC (the address which the branch should have gone to if the prediction was not correct). If E%BCOND_RETIRE_L and E%BRANCH_MISPREDICT_L are both asserted, the Ibox begins unwinding the RLOG and fetching instructions from the alternate PC. In this case, the microtrap in the Ebox will cause the Ebox and Fbox pipelines to be purged and the various Ibox-Ebox queues to be flushed. Also, E%FLUSH_MBOX_H is asserted, flushing Mbox processing of Ebox operand accesses other than writes. See Section 8.5.19 for more on Ebox handling of microtraps. See Chapter 9 for more on dispatching a microtrap. See Chapter 7 for more on activity surrounding branch misprediction.

The branch macroinstruction has entered S5 and is therefore retired even in the event of a misprediction. It is the macroinstructions following the branch in the pipeline which must be prevented from completing in the event of a misprediction trap.

The following shows all the cases the branch condition evaluator handles. The macroinstruction opcode and mnemonic are given along with the boolean equation used to determine if the branch is taken.

Table 8–9: Branch Condition Evaluation

Instruction	Opcode	Branch Taken Condition
BNEQ, BNEQU	12	NOT PSL<Z>
BEQL, BEQLU	13	PSL<Z>
BGTR	14	NOT (PSL<N> OR PSL<Z>)
BLEQ	15	PSL<N> OR PSL<Z>
BGEQ	18	NOT PSL<N>

Table 8–9 (Cont.): Branch Condition Evaluation

Instruction	Opcode	Branch Taken Condition
BLSS	19	PSL<N>
BGTRU	1A	NOT (PSL<C> OR PSL<Z>)
BLEQU	1B	(PSL<C> OR PSL<Z>)
BVC	1C	NOT PSL<V>
BVS	1D	PSL<V>
BGEQU, BCC	1E	NOT PSL<C>
BLSSU, BCS	1F	PSL<C>
SOBGEQ	F4	NOT ALU<N>
SOBGTR	F5	NOT (ALU<N> OR ALU<Z>)
AOBLSS	F2	ALU<N> XOR ALU<V>
AOBLEQ	F3	(ALU<N> XOR ALU<V>) OR ALU<Z>
ACBB	9D	(ALU<N> XOR ALU<V>) OR ALU<Z>
ACBW	3D	(ALU<N> XOR ALU<V>) OR ALU<Z>
ACBL	F1	(ALU<N> XOR ALU<V>) OR ALU<Z>
BBS	E0	E_SHF%SHF_RESULT_H<0>
BBC	E1	NOT E_SHF%SHF_RESULT_H<0>
BBSS	E2	E_SHF%SHF_RESULT_H<0>
BBCS	E3	NOT E_SHF%SHF_RESULT_H<0>
BBSC	E4	E_SHF%SHF_RESULT_H<0>
BBCC	E5	NOT E_SHF%SHF_RESULT_H<0>
BBSSI	E6	E_SHF%SHF_RESULT_H<0>
BBCCI	E7	NOT E_SHF%SHF_RESULT_H<0>
BLBS	E8	E_SHF%SHF_RESULT_H<0>
BLBC	E9	NOT E_SHF%SHF_RESULT_H<0>

8.5.12 Miscellaneous Ebox Operand Sources

Generally Ebox operand sources are registers in the register file or other registers. Certain sources are read type accesses to Ebox states, special results calculated automatically, or access to a data path not normally used as an operand source. In some cases data which can be accessed in another way is arranged in a special format as a source.

Figure 8–4: S+PSW Format

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0  0|  | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0|        | 0  0  0  0  0|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
      |                                                                  |
      |                                                                  +--- PSL<7:5>
      +--- OPCODE<0>
```

8.5.12.1 S+PSW_EX

The S+PSW_EX **E_BUS%ABUS_L<31:0>** source is simply a bit from the macroinstruction opcode and several bits from the PSL. It saves microcode steps in the CALLS and CALLG macroinstructions. Figure 8–4 shows the format of this longword source.

Bit <29> comes from the instruction context register (OPCODE<0>). Bits <7:5> come from the PSL register.

8.5.12.2 Population Counter

The Population Counter is an Ebox function unit which calculates four times the number of ones in **E_BUS%ABUS_L<13:0>** every cycle. Its result is available as a **E_BUS%ABUS_L<31:0>** source to the following microword. It saves microcode steps in the CALLS, CALLG, POPR, and PUSHR macroinstructions.

The Population Counter calculates a result in the range 0 to 14*4 equal to four times the number of ones in **E_BUS%ABUS_L<13:0>** early in S4. If microword N steers data to **E_BUS%ABUS_L<31:0>**, microword N+1 can access the Population Counter result for that data by specifying POP.COUNT in the A field. If microword N+1 is stalled in S3, Ebox control logic holds the Population Counter result until the stall ends. The effect is the same as if no stall had occurred.

The Population counter's result is used to calculate the extent of the stack frame which will be written by the macroinstruction. The two ends of the stack frame are checked for memory management purposes before any writes are done.

8.5.12.3 RN.MODE.OPCODE

RN.MODE.OPCODE is a longword composite source used when the microcode needs to access one of these data items. The four data fields in this register are RN<3:0>, CUR_MOD<1:0>, OPCODE<7:0>, and the VAX_RESTART_BIT. Figure 8–5 shows the position of these fields in the longword. This longword is one of the possible **E_BUS%BBUS_L<31:0>** sources. It is read in S3.

The RN<3:0> field is really a special data path. Its value is the GPR number in the current source queue entry. The following restrictions apply: The A field of the microword must specify S1 (the current source queue output), and the microcode must know from context that the source queue entry points to a GPR. If these restrictions are not met, the value returned in the RN field is UNPREDICTABLE.

Figure 8–5: RN.MODE.OPCODE E_BUS%BBUS_L<31:0> Source

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|      RN       | 0   0|       |            OPCODE             | 0   0   0   0   0   0   0   0|   | 0   0   0   0   0   0   0|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                            |                                                                 |
                            |                                                                 +-- VAX_RESTART_BIT
                            +-- CUR_MOD
```

The CUR_MOD<1:0> field is simply the access mode of the current process; it is taken directly from PSL<25:24>.

The OPCODE<7:0> field is the opcode from the most recent macroinstruction execution dispatch. It is taken from the instruction context register in S3. This instruction context register field has 9 bits. The 9th bit indicates the first byte of the opcode was FD#16. The opcode portion of the RN.MODE.OPCODE source does not include the 9th bit.

The VAX_RESTART_BIT field is the VAX Restart Bit which indicates that the most recently dispatched macroinstruction execution microflow has **not** altered a GPR or initiated a memory write operation of some kind. It is used to indicate to the operating system that a macroinstruction which encountered some error hasn't modified any architectural state. See Section 8.5.13 for more detail.

8.5.12.4 PMFCNT Register

The PMFCNT register which is part of the performance monitoring facility is available as an E_BUS%ABUS_L<31:0> source. See Chapter 18 and Figure 18–4.

8.5.13 VAX Restart Bit

The VAX Restart Bit is used to keep track of whether the currently executing macroinstruction has altered any architecturally visible state. It is only used by macrocode handling machine check exceptions. Conceptually, the Ebox hardware resets this bit anytime a GPR is altered or a memory write or store is initiated and sets it anytime a new macroinstruction begins. Often there is more than one macroinstruction in the NVAX pipeline, making maintenance of the VAX Restart Bit somewhat tricky.

As is described in Section 8.5.19, microtraps for faults are always taken at the end of S4, before the microword can advance to S5. The VAX Restart Bit is set reset only when operations advance to S5 and there is no pipeline abort in that cycle.

The VAX Restart Bit is reset each time a microword which alters a GPR or specifies any memory write is advanced into S5. The bit is reset in S5 when a read is sent to the Mbox and the read data is to be returned to a GPR, since that event actually writes the data on E_BUS%WBUS_L<31:0> into the specified GPR.

The memory operations specified in the MRQ field which cause the VAX Restart Bit to be reset are:

- WRITE.V.WCHK and
- WRITE.V.UNLOCK.

In addition, all microwords specifying DST/DST reset the VAX Restart Bit since destination queue indirect stores are either memory stores or GPR writes.

The VAX Restart Bit is set each time a microword which causes dispatch to an execution microflow is advanced into S5, or when microcode handles a trap exception by retiring the current instruction and dispatching to the exception handler in microcode. Specifically, it is set when MISC/RETIRE.INSTRUCTION or SEQ.MUX/LAST.CYCLE is specified by the microword in S5. The set always overrides the reset when both conditions exist in the same cycle. So the bit is reset when a microword which alters a GPR or writes or stores to memory is in S5 and that microword does not specify MISC/RETIRE.INSTRUCTION or SEQ.MUX/LAST.CYCLE.

When the Fbox is retiring results, the VAX Restart Bit is maintained properly. It is reset if the Fbox stores a result in memory or the register file (that is, it is reset on any destination queue indirect store from the Fbox). It is set when the Fbox asserts **F%RETIRE_H**, retiring the current Fbox instruction. (Note that this is not the cycle in which the microword which initiated the Fbox instruction is in S4; this is the cycle in which the Fbox sends the result of the operation to the Ebox.) As with Ebox retires, the set overrides the reset.

The VAX Restart Bit doesn't detect all changes to architecturally visible states. Microcode takes explicit action when it is about to alter some architecturally visible state other than memory or a GPR. It can, for example, copy a GPR to itself before changing the other state in question.

The VAX Restart Bit is read out in S3 but is maintained in S5. The value of this bit isn't useful if the pipeline is executing macroinstructions normally. It is useful only when a machine check exception has been detected. Since the VAX Restart Bit is updated in mid S5, it won't report a memory or GPR write until the second microword after the one which does the write.

The VAX Restart Bit is read through the RN.MODE.OPCODE **E_BUS%BBUS_L<31:0>** source. See Section 8.5.12.3.

8.5.14 Ebox-Microsequencer Interface

The Ebox receives the data path control part of the microword and the macroinstruction context information from the Microsequencer at the beginning of S3. It also receives a few signals indicating the circumstances accompanying the fetch of the microword. The Ebox sends many states which are needed for conditional branches to the Microsequencer from various points in the Ebox pipeline. The Microsequencer uses these states for conditional branch calculation.

8.5.14.1 Instruction Context Register

The Microsequencer latches macroinstruction information at the beginning of each macroinstruction execution microflow, including FPD microflows. This information was originally created in the Ibox and entered in the instruction queue. At some point the Microsequencer extracted that information along with a control store dispatch address. The Microsequencer pipelines this information so that it becomes visible to the Ebox at the same time as the microword from the dispatch address is clocked into the MIB Latch. The Microsequencer holds this data until the next time the first microword of a macroinstruction enters S3. See Section 9.2.3.3.4 and Section 9.2.3.3.4.1.

Except for the DL data, the Ebox simply carries the instruction context data down the pipeline. In the Ebox, the DL register is loaded with the DL data when the first microword of a macroinstruction is in S3. This latch can be altered under microcode control. See Section 8.5.10.6.

The information passed by the Microsequencer to the Ebox is made up of the following fields:

- **Macroinstruction Opcode; Instruction Context<OPCODE> = Instruction Context<12:4>**
  The ninth bit indicates FD#16 was the first opcode byte. This data is carried down the Ebox pipeline. It is used in S3 as a source of data and for microcode conditional branches. In S4/S5 it is used in the conditional branch evaluator.
- **Data Length; Instruction Context<DL> = Instruction Context<3:2>**
  The Ebox holds this initial instruction data length in the DL register.
- **Fbox Instruction Flag; Instruction Context<FI> = Instruction Context<1>**
  This bit is asserted if the opcode is for any macroinstruction which is normally executed in the Fbox. The Ebox enters it in the retire queue and uses to force a reserved opcode fault for Fbox instructions when the Fbox is disabled.

The Microsequencer signals that a new microflow begins with the accompanying microword and macroinstruction context information. If the new microflow is due to a macroinstruction, the Ebox latches the DL<1:0> data. The DL value can be altered by microcode, so a special latch is implemented in S3 for it. The opcode is simply carried along the pipeline. It remains latched in the Microsequencer until the next new macroinstruction flow is dispatched, so it is not latched explicitly in the Ebox. This instruction context information is available to any microword in the associated macroinstruction's execution microflow.

The floating point instruction flag is also entered in the retire queue when a new microflow is for a macroinstruction. For more detail on the retire queue see Section 8.5.15.7.

The macroinstruction context information is carried down the pipeline with each microword. The context information stalls when the microword stalls. The opcode is used in S4 and S5 to determine conditional branch results. The DL is used to control the ALU in S4, the size for any memory request in S4, **E_BUS%WBUS_L<31:0>** zero extension in S5, and GPR byte write-enables in S5. The floating point instruction flag is used in S3 to determine how to handle source operand faults.

The DL register can be altered by microcode. This occurs when the microword specifying the change is in S4. If new instruction context information enters S3 at the same time as a microword specified DL alteration occurs, the instruction context load overrides the microword specified alteration. This is because the instruction context load is for the microword subsequent to the microword specifying the DL alteration.

8.5.14.2 Microtest Fields

The Ebox provides most of the information used by the Microsequencer for microcode branches. The condition bits are driven onto the microtest bus when the Microsequencer requests it by driving the select code on **E_USQ%UTSEL_H<4:0>** and **E_USQ%UTSEL_L<4:0>**. The condition data is driven early in the cycle after it is computed. The following table shows the information the Ebox can supply. It gives the source and pipeline segment in which the data is driven. This condition information is tested in S3, as specified by the SEQ.COND field in the Microsequencer control part of the microword. The S3 operation determines the address of the next microword. So data delivered by the Ebox when microword N is in S3 is used by microword N+1 to select microword N+2. If the data is driven while microword N is in S4 or S5, one or two more cycles of microbranch latency are required, respectively.

Table 8–10: Ebox Sourced Microbranch Conditions

Source	Pipeline Stage (condition bit driven at end of stage):
ALU<Z>, ALU<C>, ALU<V>, ALU<N>	S4
SHF<N>, SHF<Z>	S4
E_BUS%ABUS_L<31,15:14,7:5>	S3
E_BUS%ABUS_L<13> OR E_BUS%ABUS_L<12>	S3
E_BUS%BBUS_L<5:3,2:0>	S3
E_BUS%BBUS_L<2:0> EQ 0	S3
E_BUS%BBUS_L<15:8> NEQ 0	S3
MPU0_6<2:0>, MPU7_13<2:0>	S4
State Flags 0-5	S3
Opcode<2:0>	S2[1]
PSL<29,26:22>	S5
VECTOR_PRESENT	always stable; configuration status bit (not used by NVAX microcode, see Section 8.5.18)
FBOX_ENABLE	always stable; configuration status bit
Field queue status - valid, and reg_mode	always accessible; Ibox-Ebox queue
Fbox fault code (see Section 8.5.19.7)	effectively always stable; not valid except in microtrap for Fbox faults

[1] bypass or flow-thru design required so first microword of a macroinstruction execution flow can specify a conditional branch on its macroinstruction opcode.

See Chapter 9 for more on microbranches.

8.5.14.3 Miscellaneous Microsequencer Signals

The Microsequencer provides the Ebox with several control signals. They signal certain Microsequencer events which have Ebox side effects.

The Microsequencer signals **E_USQ%UTSEL_H<4:0>** and **E_USQ%UTSEL_L<4:0>** are used in early S3 by the Ebox to detect that one of the MPU conditional branches (MPU0_6 or MPU7_13) is decoded from the Microsequencer control part of the microword. The Ebox clears the appropriate bit in the mask stored in the MPU by the end of S3. See Section 8.5.10.7 for more detail.

The Microsequencer signals **E_USQ%UTSEL_H<4:0>** and **E_USQ%UTSEL_L<4:0>** are used early in S3 by the Ebox to detect that the field queue status conditional branch is decoded from the Microsequencer control part of the microword. The Ebox retires an entry from the field queue if the entry was valid at the time the branch was evaluated. See Section 8.5.15.8 for more detail.

NOTE

E_USQ%UTSEL_H<4:0> and **E_USQ%UTSEL_L<4:0>** are derived almost directly from the SEQ.COND field of the Microsequencer control part of the microword. See Chapter 9.

The Microsequencer asserts **E_USQ%MACRO_1ST_CYCLE_H** when the microword in S3 is the first microword of a macroinstruction execution microflow (including the microflow at the FPD dispatch). The Ebox sets all the Wn register valid bits and resets state flags 0-3 as a result of this signal. Both effects occur in S3. It also copies PSL<T> into PSL<TP> once the microword reaches S5. Also, the Ebox latches the new instruction context DL value at the beginning of S3.

The Microsequencer asserts **E_USQ%PE_ABORT_L** when a microtrap is initiated. In this cycle all the control latches in the Ebox pipeline are flushed. Also, the Ebox flushes the retire queue.

The Microsequencer asserts **E_USQ%IQ_STALL_H** when the microword in S2 is the STALL microword (see Section 8.5.20.1). This status is carried down the Ebox pipeline along with the microword. The status is asserted (and the microword is the STALL microword) only when the Microsequencer required an instruction queue entry but no entry was valid. When this status is true, and the Ibox is asserting one of its memory error signals, the Ebox assumes a memory error in fetching the opcode byte(s) occurred. This is piped forward to S3 and then treated like any other S3 detected fault. A microtrap is forced when the condition is clocked into S4. See (Section 8.5.19). The STALL microword status is also used by the Ebox S3 stall timeout logic (see Section 8.5.25.1).

Two fields from the Microsequencer control portion of the microword are decoded by the Ebox. These fields are SEQ.MUX and SEQ.FMT. The Ebox determines when these fields decode to the operation LAST.CYCLE or LAST.CYCLE.OVERFLOW. See Chapter 9 for more on the format of the Microsequencer control portion of the microword. The decoded status is carried down the Ebox pipeline with the other decodes of the microword. When a microword specifying SEQ.MUX/LAST.CYCLE or SEQ.MUX/LAST.CYCLE.OVERFLOW is advanced into S5, the Ebox signals the Ibox that a macroinstruction is retiring (except if the microword specifies DISABLE.RETIRE/YES). See Section 8.5.15.9 for more detail.

When a microword specifying SEQ.MUX/LAST.CYCLE.OVERFLOW is advanced into S5, and the PSL<IV> and PSL<V> bits are both set, the Ebox signals the Microsequencer that an integer overflow microtrap should occur.

8.5.14.4 Miscellaneous Ebox-to-Microsequencer Signals

The Ebox sends the Microsequencer several PSL bits which affect new microflow dispatching (dispatching in response to SEQ.MUX/LAST.CYCLE or SEQ.MUX/LAST.CYCLE.OVERFLOW). They are PSL<T, TP, and FPD>. When the Microsequencer next decodes a SEQ.MUX/LAST.CYCLE or SEQ.MUX/LAST.CYCLE.OVERFLOW operation, if PSL<FPD> or PSL<TP> is set, it dispatches to special microflows (a different microflow for FPD than for TP) instead of the next macroinstruction execution microflow. If it dispatches for FPD (first part done), the Microsequencer removes an entry from the instruction queue and sends the instruction context information to the Ebox. For TP (trace fault) dispatches, the instruction queue is not referenced and the instruction context register is not loaded.

When PSL<T> is set at instruction dispatch time (including dispatching for FPD), the Microsequencer sets a local copy of the PSL<TP> bit, called LOCAL_TP (see Section 9.2.3.3.2). If LOCAL_TP or PSL<TP> is set at the time of a dispatch for a macroinstruction, the instruction queue reference does not occur and a trace fault dispatch occurs instead. This could happen on the very next cycle after the macroinstruction dispatch with PSL<T> set and PSL<TP> not set. The Microsequencer sets LOCAL_TP during the first dispatch cycle so that it can affect the immediately subsequent dispatch.

The Ebox asserts the signal **E_PSL%PSL_IS_DST_S5_H** in S5 of any cycle in which the entire PSL is being updated (i.e., if only the low byte of the PSL is updated, **E_PSL%PSL_IS_DST_S5_H** is not asserted). The Microsequencer clears LOCAL_TP when this signal is asserted. Note that the Microsequencer will initiate a trace fault dispatch if the PSL<TP> bit is set or LOCAL_TP, or both. So if a new PSL with PSL<TP> set is loaded, the trace fault dispatch will occur at the correct point.

NOTE

> There is a microcode restriction which disallows specifying SEQ.MUX/LAST.CYCLE or SEQ.MUX/LAST.CYCLE.OVERFLOW in the two microwords following one which loads the PSL. An exception to this rule is made when none of the PSL bits which affect new microflow dispatching will be changed. Some microflows know from context that none of these bits will change in a given PSL write (for example, in the execution microflow for the CALL macroinstruction, several bits in the low byte of PSL are cleared, but <T, TP, and FPD> are unaffected).

8.5.15 Ebox-Ibox Interface

The Ibox to Ebox interface is made up of a number of FIFO queues which carry operand information to the Ebox. These are the source queue, destination queue, field queue, and branch queue, which carry source operand information, destination operand information, type information for bit field operands, and branch related information, respectively. These queues are part of the Ebox. The Ibox generally processes instructions ahead of the Ebox. As it processes operand specifiers it adds entries to one or more of the queues. Each specific macroinstruction execution microflow always removes the same number of entries from each queue as the Ibox adds (unless an exception occurs). With this buffering, the Ibox and Ebox operate independently enough that stalls or latencies in one box don't necessarily cause a stall in the other, resulting in greater overall execution speed.

See Chapter 7 for more detail on many of the topics in this section.

The Ebox maintains macroinstruction ordering information in the retire queue. This FIFO is not part of the Ibox to Ebox interface, but is closely related. The Ebox is both the supplier and the consumer of retire queue entries.

In any of the queues described in this section an entry which hasn't been added is said to be invalid. Except in the case of the field queue, a stall (S3 for source queue, S4 for destination queue and branch queue) results when the microword references a queue entry which isn't valid. This stall ends when the Ibox adds enough entries to fulfill the microword's request.

In any of the queues described here, adding an entry means writing an entry, and moving the write pointer to the next entry in the queue. Accessing or referencing an entry means reading an entry, and moving the read pointer to the next entry in the queue. Where it is needed, status information concerning the number of valid entries in a queue is generated by examining the read and write pointers of that queue.

8.5.15.1 Ibox Counters

The Ibox has three counters which prevent queue overrun. Two counters are used to keep track of the number of entries in the source and destination queues, one for the source queue (allowing 12 entries) and one for the destination queue (allowing 6 entries). The Ibox increments these counters when it adds entries. The Ebox notifies the Ibox when it retires entries from the source or destination queue, and the Ibox decrements the counters in response.

Another counter in the Ibox keeps track of the number of macroinstructions which have been sent to the Ebox but have not been retired. This limits the number of entries in the retire queue, branch queue and field queue because there can be no more than one entry in each of these queues for any given macroinstruction. The counter allows up to 6 instructions in the Ebox/Fbox at a time. The Ibox increments this counter when it adds an entry to the instruction queue. When the Ebox signals the Ibox that a macroinstruction is retiring, the Ibox decrements the counter. This happens in S5 of the Ebox pipeline, one or two stages after the stage in which entries are removed from these queues. Note that this same mechanism limits the number of instruction queue entries to 6.

NOTE

> The limit of one field queue entry per macroinstruction is simply an NVAX convention. The VAX Architecture does not include instructions which have more than one bit field base address operand specifier, but NVAX defines other operands as field type where it simplifies the implementation.

The Ibox also has a counter to keep track of the number of available MD registers. It increments this counter when it allocates an MD to hold operand data (e.g., when it initiates a read of operand data from memory to an MD). When the Ebox retires a source queue entry, it tells the Ibox whether the entry pointed to an MD. The Ibox decrements the counter when the Ebox retires a source queue entry which pointed to an MD. It is possible for the Ebox to retire two source queue entries in one cycle, and the Ibox decrements the counter by two when both source queue entries pointed to MDs.

8.5.15.2 Source Queue

The source queue carries source operand information. The information is either literal mode data (6 bits) or a pointer into the register file. If it is a register file pointer, it either points to a GPR or to an MD register. The Ebox accesses one or two source queue entries per cycle in S3. Source queue accesses always cause data to be sourced to **E_BUS%ABUS_L<31:0>** or **E_BUS%BBUS_L<31:0>**. literal mode data is zero extended and driven directly onto the specified bus. Otherwise the contents of the location in the register file pointed to by the source queue entry is fetched. If the register which is accessed is not valid or is marked for writing by the Fbox, then the appropriate S3 stall occurs.

Figure 8–6 shows a source queue entry. The VALUE field is either a register file address or a 6-bit literal data value. If it is a register file address, it points to either a GPR or MD register. SH_LIT indicates whether VALUE is short literal data (if SH_LIT is 1, VALUE is short literal data).

Source queue entries are made for read, modify, address, and field operands. Both a source queue and a destination queue entry is made for each modify operand.

Figure 8–6: A Source Queue Entry

```
 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+
|  |       VALUE     |
+--+--+--+--+--+--+--+
  |
  +--- SH_LIT
```

Field operands in NVAX are classified into read and modify types. Read and modify field operands both result in a source queue entry. Modify field operands also result in a destination queue entry if the operand specifier is register mode.

Two source queue entries are made for quadword length operands. If they are for registers, they point to registers N and N+1. If they are memory operands, they point to MD registers which will receive data from memory addresses A and A+4. For literal mode, the first value is the immediate data, and the second is 0.

Source queue access fulfills a necessary synchronization function. When microcode successfully accesses a source queue entry it knows that the Ibox was able to fetch the associated operand specifier. It also knows that there is no access violation or invalid translation condition associated with the operand. For modify type operands it also knows that the location will not give an access violation when written. Microcode for complex macroinstructions always references all source operands which might cause a memory management fault before altering any architecturally visible state.

The number of entries in the source queue is 12.

8.5.15.3 Destination Queue

The destination queue carries destination operand information. The information is either an address in the register file of a GPR or a status indicating a memory write to the address in the PA queue in the Mbox. The destination queue is accessed in S4 (no more than one entry per cycle is used). Its information is used to decide how to write the result which is being calculated by the ALU, shifter, or Fbox in the same cycle. If the destination queue entry indicates a memory store, the request is sent to the Mbox. An S4 stall occurs if the Mbox is already busy or the PA queue entry is not ready. If the destination queue entry indicates a GPR write, the register file will be written using the address from the destination queue. The GPR write occurs in the next cycle (S5).

Figure 8–7 shows a destination queue entry. The VALUE field is either a register file address or is unused. If it is a register file address, it points to a GPR. MDEST indicates whether the destination of the data is memory. If MDEST is 0, the result is destined for the register file and VALUE field indicates the destination address. If MDEST is 1, the destination of the data is memory and the VALUE field is unused.

Destination queue entries are made for modify and write access type operands. Also, modify field operands result in a destination queue entry if the operand specifier is register mode.

Figure 8–7: A Destination Queue Entry

```
  04|03  02  01  00
+---+---+---+---+---+
|   |     VALUE     |
+---+---+---+---+---+
  |
  +--- MDEST
```

Two destination queue entries are made for quadword length operands. If they are for registers, they point to registers N and N+1. For memory operands they point to addresses A and A+4.

Destination queue access fulfills a necessary synchronization function. If the destination queue entry is accessed and used successfully, microcode knows that the destination operand specifier was fetched successfully and that there will be no access violation when the destination location (if it is in memory) is written. In the case of quadword data length, successful use of the first destination queue entry guarantees that the second write will not incur a memory management exception either.

The destination queue contains the Fbox destination scoreboard function. See Section 8.5.16.4 for more information.

The number of entries in the destination queue is 6.

8.5.15.4 Miscellaneous Queue Retire Information

When an entry is retired from the source or destination queues, certain information is sent back to the Ibox. The Ibox uses this information to maintain three counter values and to maintain GPR scoreboard information in the scoreboard unit (SBU).

Zero or one destination queue entry can be retired in a given cycle. The retire information sent to the Ibox for the destination queue is:

- whether an entry is being retired,
- whether the entry being retired indicates a GPR write or a memory write, and
- the GPR number if it is a GPR write.

The Ebox signals the Ibox when a destination queue retire occurs early in the cycle in which the operation is advanced into S5.

Zero, one, or two source queue entries can be retired in a given cycle. Similar information is sent for each of the two source queue read ports. The retire information sent to the Ibox for each source queue read port is:

- whether an entry is being retired,
- whether the entry being retired indicates a GPR read, an MD read, or is short literal data, and
- the GPR number if it is a GPR read.

The Ebox signals the Ibox when one or two source queue retires occur. It does this early in the cycle in which the microword retiring the source queue entries is advanced into S4.

8.5.15.5 Branch Queue

The branch queue carries information for conditional and unconditional branches. The information is a one-bit prediction status. The prediction status is only used by conditional branches. It indicates which way the Ibox predicted the conditional branch would go. The Ebox references the branch queue for two reasons: to synchronize with the Ibox fetch of the branch displacement and to compare the Ibox branch prediction to the actual branch result.

The Ebox accesses the branch queue in S4 when the microword specifies SYNC.BDISP, SYNC.BDISP.RETIRE, or SYNC.BDISP.TEST.PRED in the MRQ field. SYNC.BDISP.RETIRE is used in unconditional branches. SYNC.BDISP.TEST.PRED is used in all conditional branches. SYNC.BDISP is used in some complex conditional branches. The Ibox doesn't add an entry to the branch queue until it has successfully fetched the displacement. When the Ebox accesses the branch queue, it will stall until there is an entry. This stall occurs in S4 and prevents the branch macroinstruction from retiring before the displacement has been successfully fetched.

For conditional branches, the Ebox waits for the Ibox to add the entry to the branch queue and then compares the Ibox prediction to the actual result of the branch which is calculated in the Ebox. If the branch was mispredicted, the Ebox initiates a microtrap in S5. Because the microtrap is in S5, the branch macroinstruction retires but subsequent macroinstructions are prevented from completing.

In some complex conditional branches, the Ebox microcode waits for the branch queue entry to become valid before it stores a result calculated by the instruction. This allows the microcode to be sure the branch displacement was fetched without a memory management fault or hardware error before modifying state. The microcode may have to delay retiring the branch queue entry and checking the branch prediction. So SYNC.BDISP accesses the branch queue, and causes an S4 stall if the entry is not valid, but does not cause the entry to be retired.

The Ebox signals the Ibox whenever a microword which retires a conditional branch queue entry advances into S5 (that is a microword specifying SYNC.BDISP.TEST.PRED). This causes the Ibox to release the alternate branch path PC (the PC of the path not taken by the Ibox prediction). The Ebox signals a mispredicted branch at the same time, if there is one. If there is a mispredicted branch, the Ibox responds by unwinding the RLOG and resuming macroinstruction fetching at the alternate PC address.

Due to complexity in the branch queue bypass logic, it may happen that one cycle of "unnecessary" stall occurs in cases where there back-to-back branches are executed. The extra cycle of stall happens only if the two branches are in adjacent stages of the Ebox pipeline and the Ibox writes the second branch queue entry one cycle before the the second branch is in S4, ready to retire (i.e., it wouldn't be stalled except for the branch queue stall). In this case the branch queue read pointer is being advanced and another branch queue entry is being written. Bypass is not implemented for the second branch in this specific case.

The number of entries in the branch queue is 6.

8.5.15.6 Operand and Branch Buses

The transmission of operand information for the source queue, destination queue, and field queue occurs via the operand bus. This bus is described in Chapter 7. It carries all the information which might be entered into any of these queues, and it has valid bits which tell the Ebox when to add entries.

The operand bus caries information derived from decoding a single operand specifier. Zero, one or two source and/or destination queue entries are specified, and zero or one field queue entry. Only when the operand is quadword length can more than one source or destination queue entry be made. Whether a source or destination queue entry is made depends on whether the operand is read, write, or modify access type. (Note that the access type referred to here might not be identical to the true access type given in the VAX Architecture Standard, for various reasons.)

A field queue entry is made for each field operand. The Ibox instruction decode logic determines if a particular operand is a field operand. Only certain macroinstructions have a field operand, and no macroinstruction has more than one field operand.

The branch queue receives its information via the branch bus. This bus has one bit of data (a prediction status) and a valid bit. A branch queue entry containing the prediction status is added in every cycle in which the valid bit is asserted. See Chapter 7 for more information.

8.5.15.7 Retire Queue

The retire queue is used by the Ebox to force macroinstructions to retire in order. It contains one bit of information, a status indicating whether the Ebox or Fbox is the source of the next macroinstruction to retire. The Ebox adds an entry to the retire queue in S3 each time a new macroinstruction execution microflow begins. (If there is an S3 stall, the entry is added to the retire queue in the first cycle of the stall. Exactly one entry is made whether or not an S3 stall occurs for one or more cycles.) The retire queue entry is the FI bit from the instruction context register (see Section 8.5.14.1). However, if the FBOX_ENABLE bit in ECR (IPR 125) is not set, the retire queue entry is forced to indicate Ebox retire regardless of the FI bit. Similarly, if PSL<FPD>, PSL<27>, is set, the retire queue entry is forced to indicate Ebox retire regardless of the FI bit.

The the retire queue is forced to indicate that the Ebox is next to retire when ECR<FBOX_ENABLE> is not set because the Fbox will not receive an operation dispatch from the Ebox (F%FBOX_1ST_CYCLE_H will never be asserted). ECR<FBOX_ENABLE> also disables microcode sending of operand data, overriding microcode. The Ebox generally forces a reserved instruction microtrap when Fbox instructions are in S4 (see Section 8.5.16.8 for more detail). This microtrap flushes the retire queue (and, because the retire queue is empty, the Ebox is automatically selected as the RMUX source).

If the Fbox instruction is MULL, a reserved instruction microtrap does not occur (see Section 8.5.16.8). Instead the Ebox microcode executes the MULL. This requires that the Ebox be selected as next to retire and is the reason ECR<FBOX_ENABLE> forces the retire queue entry to select the Ebox.

When PSL<FPD> is set SEQ.MUX/LAST.CYCLE and SEQ.MUX/LAST.CYCLE.OVERFLOW causes the microsequencer to dispatch to a specific microcode entry point regardless of the instruction queue contents. Since this dispatch is to an Ebox microcode flow which will not send operands to the Fbox, the Ebox must be selected in the retire queue (though any previous instruction is not affected and retires normally). Otherwise, the Ebox could stall waiting for the Fbox to retire an instruction while the Fbox waited for source operands to be sent. That deadlock would only end on S3 stall timeout.

The Ebox examines (without retiring an entry) the retire queue in S4 to determine whether the Fbox or the Ebox is the next source of a retiring macroinstruction. Based on the retire queue output, the RMUX is set to select either the Fbox or the Ebox as the source of control for S4-initiated memory references and most S5 operations. This selection remains in effect until the retire queue entry is retired. See Section 8.5.5 for more on how this status is used to control the RMUX.

If the Ebox is the next to retire a macroinstruction, the retire queue entry is retired in S4 when the microword advancing into S5 specified SEQ.MUX/LAST.CYCLE or SEQ.MUX/LAST.CYCLE.OVERFLOW and did not specify DISABLE.RETIRE/YES. If the Fbox is the next to retire a macroinstruction, the retire queue entry is retired in S4 when the Fbox asserts **F%RETIRE_H**. In either case the retire queue entry is not retired unless the selected operation advances into S5 (i.e., there is no S4 stall). (Note that a retire queue entry is not retired by the MISC1/RETIRE.INSTRUCTION operation.)

The retire queue is flushed when a microtrap occurs as well as when the MISC field function RESET.CPU is specified. Anytime the retire queue is empty, the Ebox is automatically selected as the source of the RMUX.

Note that it is not possible for the retire queue to have less than the necessary number of entries in it, except after a microtrap, because each entry is added before it is required.

The number of entries in the retire queue is 6.

8.5.15.8 Field Queue

The field queue carries information about field type source operands for bit-field macroinstructions and some other macroinstructions. The information is one bit which indicates whether the operand was register mode or not. Two different execution microflows are required for bit-field macroinstructions and certain other macroinstructions depending on whether a particular operand is register mode. The Ibox provides this information when it adds a source queue entry for the operand. Microcode is able to branch conditionally on the status of the field queue. This allows execution microflows to decide how to execute the instruction.

Each entry in the field queue is a one-bit status which indicates whether the associated field operand is register mode. Microcode branches on a field queue entry are four way branches, though only three of the four outcomes are possible. The following table shows the possible branch outcomes.

Table 8–11: Field Queue Branch

Condition	Resulting Microtest Bus Value
Field queue empty	11 (can be execution dispatch target)
Field queue not empty—register mode	01 (start of execution for register mode case)
Field queue not empty—not register mode	00 (start of execution for address mode case)

A branch on the field queue when it is not empty causes the current field queue entry to be retired.

The field queue has 6 entries.

When the Ebox is branching on the field queue, it may have to wait for the Ibox to make an entry, in which case it loops repeatedly testing the field queue. This condition is similar to a stall, but no Ebox stall is involved. When microcode is branching on the field queue and it is empty, the signal **E_FLQ%FQ_STALL_H** is asserted. This tells the S3 stall timeout logic that the Ebox is looping on the field queue. If this continues for a long time, a machine check occurs. See Section 8.5.25.1 for more detail.

E_FLQ%FQ_STALL_H is also used by the fault logic. If **E_FLQ%FQ_STALL_H** is asserted and one of **I%IMEM_MEXC_H**, **I%IMEM_HERR_H**, or **I%RSVD_ADDR_FAULT_H** is asserted, then a S3 fault condition is detected. After a cycle in which there is no S4 stall (and given that the Ebox is next to retire), the fault condition advances into S4 and the appropriate microtrap is requested. See Table 8–12 and Section 8.5.19 for more information.

8.5.15.9 Retiring Instructions

Retiring a macroinstruction is an important synchronization point between the Ebox and the Ibox. When a macroinstruction is retiring, the last of its operations is in S5 and cannot be stalled or aborted. The Ebox signals the Ibox so that it can free up certain resources associated with the retiring instruction. The Ebox usually retires a retire queue entry at the same time as it retires the macroinstruction (the exception is MISC1/RETIRE.INSTRUCTION which doesn't affect the retire queue).

The resources in the Ibox which are freed up by retiring a macroinstruction are a backup PC queue entry and a group of RLOG entries associated with that macroinstruction.

When the retire queue indicates the Ebox is next to retire a macroinstruction, the set of conditions required for retiring to occur are:

- the microword in S5 specifies SEQ.MUX/LAST.CYCLE or SEQ.MUX/LAST.CYCLE.OVERFLOW, and not DISABLE.RETIRE/YES, or
- the MISC1 field function, MISC1/RETIRE.INSTRUCTION, is specified (though the retire queue is not affected in this case).

The Fbox determines its own retire instruction status which it sends through the RMUX when the retire queue indicates the Fbox is next to retire a macroinstruction. If the Fbox operation request in S4 is advanced to S5 with this condition asserted, the Ebox retires an instruction.

8.5.15.10 First Part Done

The Ebox sends the current state of the PSL<FPD> bit to the Ibox on **E%FPD_SET_L**. If the Ibox fetches an opcode and this bit is set, the Ibox stops operation as soon as the opcode has been completely fetched. If the instruction is an interrupted instruction that is being resumed, then the operand specifiers mustn't be processed again since they may have side effects or may depend on data which has been altered by the instruction's execution.

8.5.15.11 Ebox to Ibox Commands and IPR Accesses

The Ebox is the source of two signals which immediately affect Ibox operation, and three others which cause IPR read and write operations or a load-PC operation.

The two signals which immediately change Ibox operation are: **E%STOP_IBOX_H** and **E%RESTART_IBOX_H**. **E%STOP_IBOX_H** is asserted in S5 when the microword specifies MISC/RESET.CPU. **E%RESTART_IBOX_H** is asserted when the microword in S5 specifies MISC/RESTART.IBOX.

E%STOP_IBOX_H is used to cause the Ibox to stop processing instructions and clear the Ibox GPR scoreboard. It does not clear the RLOG or backup PC queue, so the Ibox is still able to restore state to that required for a fault. See Chapter 7 and Section 8.5.19.

E%RESTART_IBOX_H is used to restart the Ibox when it put itself in the stopped state after processing the operands for certain complex instructions.

The Ebox detects its own accesses to Ibox IPRs in S5 just after issuing the request to the Mbox. It also decodes MRQ/LOAD.PC to detect a load-PC operation in S5. At that time it asserts one of three command strobes to the Ibox. They are **E%IBOX_IPR_READ_H**, **E%IBOX_IPR_WRITE_H**, and **E%IBOX_LOAD_PC_L**. The Ebox drives the signal fields **E%IBOX_IPR_TAG_H<2:0>** and **E%IBOX_IPR_NUM_H<3:0>** with the Wn register file destination for IPR read data and the IPR number, respectively. (The full register file address for the destination is 6 bits, but the Ibox appends the prefix for Wn registers since all Ibox IPR reads are sent to Wn registers.) For IPR writes and load-PC operations the Ibox receives the data when the Mbox forwards it on **M%MD_BUS_H<31:0>** in a later cycle. For read accesses the Ibox returns the data to the designated Wn register.

Microcode synchronizes load-PC operations by issuing an Mbox operation (possibly MRQ/SYNC.MBOX). This synchronization is necessary because the Ibox will not be ready to accept the new PC data if a MISC/RESET.CPU occurs before the new PC data is forwarded by the Mbox. Any interrupt or exception which occurs after the load-PC will cause the Ebox to read the backup PC from the Ibox, and that value must have resulted from the load-PC operation. Once the synchronizing Mbox operation is complete, the microcode knows the Ibox has the data.

Ibox IPR writes are synchronized by issuing a MRQ/SYNC.MBOX (or another Mbox operation) after the operation. Once the MRQ/SYNC.MBOX (or other Mbox operation) is complete, the microcode knows the Ibox has the data.

8.5.15.12 Loading The PC

The Ibox maintains all PC information for the NVAX CPU. When microcode executing in the Ebox determines that instruction fetching should begin at some address, it sends the starting PC value to the Ibox. Conceptually, this is equivalent to loading the PC register. However, the Ibox keeps track of a number of PC values, and there isn't really a current PC register. See Chapter 7 for more on how PC values are maintained.

The Ebox sends a new PC value to the Ibox in S5 when the microword specifies LOAD.PC in the MRQ field. The PC data is sent via the Mbox. Microcode first ensures that the Ibox is stopped and, if necessary, flushes appropriate queues. Note that the RLOG should have been unwound beforehand.

8.5.15.13 Ebox to Ibox Flush Signals

Microcode is able to flush several entities in the Ibox: the virtual instruction cache (VIC), the branch prediction cache (BPC), and the backup PC queue (PCQ). In S5, the Ebox drives **E%FLUSH_VIC_H**, **E%FLUSH_BPT_H**, and **E%FLUSH_PCQ_H**, when it decodes MISC1/FLUSH.VIC, MISC1/FLUSH.BPC, and MISC1/FLUSH.PCQ, respectively.

8.5.15.14 Detecting Ibox Incurred Faults and Errors

There are two kinds of faults which can occur due to Ibox processing. Also, hardware errors can occur. When a fault or error occurs, the status is latched. The Ebox effectively detects the fault or error when it executes a microword which uses the result of the operation which incurred the fault or error. The Ebox causes a microtrap to occur when that same microword is about to be advanced from S4 into S5. See Section 8.5.19 for more on microtrap management.

Some Ibox incurred faults and errors are initially detected by the Ibox, while others are first detected by the Ebox. When the Ibox detects a fault or error, it halts operation and asserts one of two fault indication signals or an error indication signal which are all received by the Ebox. These signals are **I%IMEM_MEXC_H** (which indicates a memory management fault), **I%IMEM_HERR_H** (which indicates a hardware error), and **I%RSVD_ADDR_FAULT_H** (which indicates a reserved addressing mode). The Ibox only asserts **I%RSVD_ADDR_FAULT_H** for one cycle, so the Ebox has a latch which is set when it is asserted. This latch is reset by MISC/RESET.CPU and by branch mispredict microtraps.

The Ebox ignores Ibox fault conditions until it determines that they applies to the current microword. This is done by associating some queue empty condition with the fault status. See Table 8–12.

Faults and errors not detected by the Ibox are reported by the Mbox. For reads, the Mbox sets the fault or error bit associated with the target MD register in the register file. For writes, it sets the fault or error bit in the appropriate PA queue entry. When the Ebox references the MD register or tries to use the PA queue entry with a fault bit set, it detects the fault.

Faults in memory reads issued by the Ibox as an intermediate step in processing an operand specifier (as in register deferred mode) are handled in a special way. When the memory read fault or error is detected in the Mbox, it returns a fault/error status instead of data. The Ibox latches this fault/error status. If the Ibox was going to use this data as an address (deferred mode), it sends the fault/error status with the next specifier related memory request. The Mbox, seeing the fault/error status associated with the operation, sends the result to the MD register (for reads) or PA queue (for writes) with the same fault/error status.

Detecting faults in memory reads issued by the Ibox as an intermediate step in processing an operand specifier can also occur another way. In the case where the Ibox will not have to issue a memory request using the result of the failed request (as in address access type with a deferred mode operand specifier), the Ibox reports the error by writing the MD fault or error status bit directly. The fault/error status latched in the Ibox is written into the MD fault/error status bits when the Ibox writes the MD.

The table below lists the faults and indicates how each is detected.

Table 8–12: Detection of Ibox Incurred Faults and Errors

Fault	How Detected
Instruction stream read fault/error on opcode	Instruction queue empty AND (I%IMEM_MEXC_H OR I%IMEM_HERR_H)

Table 8–12 (Cont.): Detection of Ibox Incurred Faults and Errors

Fault	How Detected
Instruction stream read fault/error on source operand (including modify type)	(Source queue empty[1] OR E_FLQ%FQ_STALL_H asserted) AND (I%IMEM_MEXC_H OR I%IMEM_HERR_H)
Instruction stream read fault/error on destination operand (write type)	Destination queue empty AND (I%IMEM_MEXC_H OR I%IMEM_HERR_H)
Instruction stream read fault/error on branch displacement	Branch queue empty AND (I%IMEM_MEXC_H OR I%IMEM_HERR_H)
Memory access fault/error encountered in processing a source operand (including modify type)	Attempt to read an MD register with a fault bit set
Memory access fault/error encountered in processing a destination operand (write type)	Attempt to use a PA queue entry with a fault bit set
Reserved addressing mode on source operand	(Source queue empty[1] OR E_FLQ%FQ_STALL_H asserted) AND I%RSVD_ADDR_FAULT_H
Reserved addressing mode on destination operand	Destination queue empty AND I%RSVD_ADDR_FAULT_H
Reserved opcode	Microsequencer Dispatch

[1] In this context, source queue empty includes the case where the microword in S3 requires two source queue entries to advance, but only one entry is present in the source queue.

It is not possible for the Ibox to assert both **I%RSVD_ADDR_FAULT_H** and either of **I%IMEM_MEXC_H** or **I%IMEM_HERR_H** at the same time. The Ibox stops operation as soon as it encounters one of these two faults, so the other cannot occur after one is detected.

8.5.16 Ebox-Fbox Interface

The Fbox executes independently of the Ebox but is dependent on the Ebox for delivery of source operands and storing of results. Floating point macroinstructions are decoded by the Ibox exactly like any other macroinstruction. The Ebox is dispatched to an execution microflow. This microflow delivers the source operands to the Fbox in S3 of the pipeline. Once the operands are delivered, the microflow is done. The Fbox returns the result in S4, along with any faults it might have detected. The Ebox keeps track of whether the Fbox macroinstruction is next to retire using the retire queue (see Section 8.5.15.9 and Section 8.5.15.7). Once the Fbox is next to retire, the Ebox may, at the Fbox's request, access the destination queue for the Fbox to determine where the Fbox results are to be written. When the Fbox indicates its last execution cycle, the Ebox retires a retire queue entry and updates the PSL with an Fbox supplied condition code.

8.5.16.1 Fbox Opcode and Operand Delivery

The Ebox prepares to deliver operands during S3 when the microword specifies FOP.VALID in the MISC1 field. The opcode<8:0> for the instruction is delivered from the Microsequencer late in S2, so that the Fbox can decode the opcode before the operands arrive. The operands are available at the beginning of S4. They come from the output of the bypass muxes so that result data from the most recent S4 (Ebox or Fbox) operation is bypassed if necessary. Anything which stalls S3 in the Ebox, stalls Fbox operand delivery (this includes S4 stalls). Along with the operands, the Ebox sends the current value of PSL<FU>.

If the Ebox detects a fault or error associated with an Fbox source operand, it indicates this to the Fbox. The Fbox carries this information along its pipeline and indicates the fault and/or error when the Ebox is retiring the Fbox operation. This is how Fbox source operand fault microtraps are delayed until all preceding macroinstructions have retired. The Ebox ignores source operand faults (which proceed down the pipeline to S4) when the Fbox is next to retire.

8.5.16.2 Fbox Result Handling

The Ebox handles writing of Fbox results in S4 and S5. When the current retire queue entry indicates the next macroinstruction to retire is to come from the Fbox, the Ebox waits for the Fbox to assert **F%STORE_H** or **F%RETIRE_H**. Either or both may be asserted. If **F%STORE_H** is asserted, the Ebox accesses the destination queue and issues a memory store or a GPR write, depending on the MDEST bit in the current destination queue entry. (See Section 8.5.17 for the exact definition of memory store.)

The Fbox indicates it is retiring an instruction by asserting the signal **F%RETIRE_H**. In response to this signal, the Ebox retires the current retire queue entry. The Fbox sends a map specifier which tells the PSL logic in S5 of the Ebox pipeline how to set the PSL condition code bits based on the Fbox condition code. There may be an Fbox result store at the same time as a retire.

The storing of Fbox results is handled exactly like the storing of Ebox results in the pipeline. The request is made in S4, through the RMUX. The Fbox supplies the data length for the store. (It derives the data length from the opcode.) If there is no stall or fault, the operation is advanced into S5 where the write is done unconditionally. Condition code updates are done in S5, too. The stalls which apply to this operation are the same as for an Ebox microword doing a stall. The destination queue and PA queue must have valid entries and the Mbox must be ready, if the Fbox is doing a store. The retire queue must indicate the Fbox for an Fbox store or retire to be allowed. Otherwise the Fbox store or retire is stalled.

8.5.16.3 Fbox Store Stall

In some cases the Fbox asserts **F%STORE_H** to indicate it has result data to store and then asserts **F%STORE_STALL_H** to abort the store. This is done because certain Fbox operations may take an extra cycle, depending on the actual data pattern. **F%STORE_STALL_H** is asserted too late for the Ebox to not send a store request to the Mbox (if the result is supposed to be stored to memory). If a store is forwarded to the Mbox and is then revoked by **F%STORE_STALL_H**, the Ebox asserts **E%EM_ABORT_L** early in the next cycle to abort the EM_LATCH operation and purge the EM_LATCH. This is the same mechanism used to abort EM_LATCH operations when an Ebox pipeline abort occurs (see Section 8.5.17.2).

Due to complexities in the Mbox, (see Section 8.5.17.2), the Ebox ignores **M%PA_Q_STATUS_H<0>** in cycles in which **E%EM_ABORT_L** is asserted because of previous **F%STORE_STALL_H** assertion. In this cycle, it behaves as if **M%PA_Q_STATUS_H<0>** is deasserted.

Ignoring **M%PA_Q_STATUS_H<0>** and behaving as if it is deasserted has the effect of unconditionally stalling the Fbox store (which is always ready in these cases in the current implementation). This means there is one cycle additional latency beyond that introduced by the Fbox aborting the store. Note this only occurs when **E%EM_ABORT_L** is actually asserted. If the abort store never was sent to the Mbox, **M%PA_Q_STATUS_H<0>** is not ignored.

8.5.16.4 Fbox Destination Scoreboard

The Ebox maintains state to detect pending Fbox stores to GPRs in the Fbox destination scoreboard. If any Ebox or Fbox operation attempts to source one of the GPRs which the Fbox is scheduled to update, the Ebox stalls and Fbox operand delivery is stalled. The Fbox destination scoreboard is implemented as part of the destination queue. This section describes the Fbox destination scoreboard functionality of the destination queue. See Section 8.5.15.3 for more on the main function of the destination queue.

The Fbox destination scoreboard consists of a pair of comparators and a write-pending bit associated with each destination queue entry. If an Fbox update of a particular GPR is pending, the write-pending bit in the destination queue entry for that store is set. The bit is set in S4, by specifying F.DEST.CHECK in the MISC2 field. If the Fbox source operands are all sent by one microword, that microword specifies MISC2/F.DEST.CHECK. If a sequence of more than one microwords sends the source operands to the Fbox, the MISC2/F.DEST.CHECK is in the last microword.

Whenever a GPR is accessed using the source queue (A/S1 and/or B/S2) in S3 , every destination queue entry with a set write-pending bit is compared with the two outputs of the source queue. A match, or hit, causes a stall if the source queue output which hits is actually specified by the microword in the A or B fields. For a hit to cause a stall, the write-pending bit in the destination queue must be set. Additionally, the source queue output which hits must specify a GPR access (i.e., it must not point to an MD register or contain literal data). If these conditions are met, the S3 operation is stalled.

Note that the above check includes destination queue entries with their MDEST bit set. So pending writes to memory (using PA queue addresses) may cause a scoreboard hit stall. This is not done to prevent the Ebox from reading a GPR before a pending Fbox write to the GPR completes. Instead, it is done to prevent the Ebox from reading a GPR when the Ibox must write an incremented or decremented value first. This occurs when the Ibox processes an autoincrement or autodecrement specifier with write access type for an Fbox instruction. In processing the specifier, the Ibox CSU can be stalled for some reason, and thus be delayed from writing the new value to the GPR. To handle this case, the Ibox sends the GPR number with ALL destination queue entries. If the Ebox reads a GPR which was used in a destination specifier, the scoreboard hit stall prevents the read until the destination queue entry is retired.

Because of the minimum latency in the Mbox in processing specifier accesses, it is known that the Ibox CSU will update the GPR before the associated PA queue entry becomes valid, and the destination queue entry will not be retired until the PA queue entry becomes valid. (Actually, the destination queue entry is effectively retired before the Ebox "knows" that the PA queue entry is not valid, but then an S4 stall exists which will last until the PA queue entry becomes valid. This stall will also stall S3, so the GPR access will be prevented until the GPR is valid. This is why all RMUX S4 stalls also stall S4 and S3 when the Fbox is next to retire an instruction.)

In the event that a modify access type specifier is processed, an entry is made in the source and destination queues for the same specifier. If it is a register mode specifier, it does not cause a deadlock because the MISC2/F.DEST.CHECK operation which sets the write pending bit in the destination queue for the entry is not done until the last microword of the execution microflow is in S4. By that time all the operands have been sent to the Fbox. If the addressing mode is some memory access mode, the operand bus bits which carry the GPR number when processing a write access type specifier are used instead to carry the index of the MD register which will hold the source data. Interpreting this MD index as a GPR number could cause lost performance if a subsequent instruction accesses the GPR with the same index as that MD. (Deadlock doesn't occur for the same reason as before.) To prevent possible loss in performance, the Ebox forces the index bits to 1 as they are written into the destination queue GPR field for modify access type operands only. This has the effect of converting specifying the PC in the destination queue. If a subsequent instruction does access the PC directly, then a stall will not hurt since this is an UNPREDICTABLE case. (The Ebox supplies the value 0 when the PC is specified in this way.)

NOTE

> When the Ebox is next to retire an instruction and is writing to a write access type destination operand, it will stall in S4 if the PA queue is not valid. This causes an S3 stall. Thus the case which motivated the above special scoreboarding case for Fbox destinations can not occur for Ebox instructions. In fact, the only reason it can occur for Fbox instructions is because there are several "hidden" pipeline stages between S3 and S4 when the Fbox processes an instruction. These extra pipeline stages allow the Fbox to accept new instructions and their associated source operands before it has retired the current instruction. This combined with the fact that the Ibox can process "simple" specifiers for new instructions even while the CSU is stalled processing a complex write access type specifier from a previous instruction is what leads to the need for the special scoreboard case described above.

The Ebox will access ahead of the current destination queue entry as part of the Fbox destination scoreboard function. A pointer called the FDest pointer is maintained which may point to an entry which is after the front entry in the FIFO queue. Normally, it points to the current entry. However, in circumstances where the Fbox is next to store a result, it is incremented ahead of the current destination queue entry pointer.

When the microword in S4 specifies MISC2/F.DEST.CHECK, the Ebox checks that the destination queue entry at the FDest pointer is valid. If it isn't, S4 stalls (stalling S3 as well). If the destination queue entry is valid, the associated write-pending bit is set. If the DL is quadword, then the bit associated with the next destination queue entry is also set. The FDest pointer is incremented by one, or by two if the DL is quadword. The write-pending bits are set in S4 even if there is an S4 stall. The FDest pointer is incremented as the operation advances into S5, when there are no S4 stalls.

NOTE

> The DL supplied in the instruction queue with Fbox instructions is the length of the result.

Flow-thru bypass ensures that the S3 microword is stalled if it is accessing a GPR and that GPR is specified by a destination queue entry whose write-pending bit is being set by the microword in S4.

Write-pending bits in the destination queue are reset in S4 as the Fbox writes results, even if the MDEST bit is set in the destination queue entry being retired. Flow-thru bypass ensures that an S3 stall due to the scoreboard is broken in the cycle in which the Fbox drives the result to the Ebox. This means the result in S4 (after the RMUX) is bypassed to **E_BUS%ABUS_L<31:0>** and/or **E_BUS%BBUS_L<31:0>** in these cases.

In S4, when the Fbox stores a result, the write-pending bit of the destination queue entry is reset. This means that destination queue entry can no longer cause a scoreboard hit stall. The bit is cleared even if RMUX S4 stalls. In all cases this is safe either because the destination queue entry has MDEST set or because the particular RMUX S4 stall also causes an S4 stall which in turn causes an S3 stall which prevents Fbox operand delivery.

The write-pending bits and all destination queue pointers are reset when **E_MSC%FLUSH_EBOX_H** is asserted. This happens in every microtrap, including the power-up microtrap.

8.5.16.5 Fbox Fault and Error Management

As mentioned above, the Fbox latches source operand fault and error information and carries it along with its other instruction related information. Also, the Fbox may encounter a fault in the course of computing the result. All these faults and errors are presented by the Fbox when it requests the RMUX. The Ebox responds by signaling a microtrap to the Microsequencer once the retire queue indicates the Fbox. Before the retire queue points to the Fbox, the Ebox ignores the fault status coming from the Fbox.

The Ebox detects Ibox incurred faults and errors for Fbox operands as described in Table 8–12, but instead of handling them directly, it passes the fault/error status to the Fbox. The Fbox doesn't wait for the operand valid signal when a fault or error status is asserted, even though there isn't valid data. This breaks a stall which might never end otherwise, since the Ibox stops processing operand specifiers when it encounters a fault or error.

NOTE

> The Fbox treats the data which comes with the fault/error status as UNPREDICTABLE. Also the Fbox breaks the stall on any operands which follow an operand with an associated fault or error. The Ibox stops processing operand specifiers when it encounters a fault or error. If the Fbox didn't break the stall and propagate the fault/error to the RMUX, the CPU would hang.

If there isn't a fault or error being signaled by the Fbox, there could still be a destination operand fault or error. If the Fbox is requesting the RMUX and indicating a destination queue indirect store, the Ebox checks for a destination operand fault or error (see Table 8–12). If there is one, the appropriate microtrap is forced.

Most Fbox faults, and all Fbox errors, result in VAX architecture exceptions of the fault type. This means most Fbox faults, and all errors, are taken in S4 when the operation is about to advance into S5. Integer overflow is a trap in the VAX architecture sense, and causes a microtrap late in S5.

Fbox operand faults and errors have higher priority in the Microsequencer than Fbox originated data faults. Fbox operand faults cause the same microtraps as would be taken if that fault or error was detected in an Ebox instruction. Fbox originated data faults cause a floating fault microtrap, provided there aren't any operand faults or errors. See Section 8.5.19.7 for more on how microcode determines the cause of the microtrap.

8.5.16.6 Ebox to Fbox Commands

The Ebox asserts the signal **E%FLUSH_FBOX_H** when the microword in S6 specifies RESET.CPU in the MISC field. This has the effect of reseting the Fbox and clearing its pipeline of all operations.

8.5.16.7 Summary of Fbox-Ebox Signals

The following signals are driven by the Ebox to the Fbox.

- **E%FOPCODE_H**
  This 9-bit bus carries the full opcode for Fbox operations. (This bus is actually driven by the Microsequencer.)
- **E%FBOX_1ST_CYCLE_L**
  This bit indicates there is a valid Fbox opcode on **E%FOPCODE_H**. (This signal is actually driven by the Microsequencer.)
- **E%ABUS_H** and **E%BBUS_H**
  These 32-bit busses carry the source operand(s).
- **E%FDATA_VALID_H**
  This signal tells the Fbox that all operands being sent to it are valid. The Fbox knows, from decoding the opcode, exactly what data is being sent on **E%ABUS_H<31:0>** and **E%BBUS_H<31:0>**.
- **E%A_SHLIT_H** and **E%B_SHLIT_H**
  These signals indicate the data on **E%ABUS_H<31:0>** and **E%BBUS_H<31:0>**, respectively, is a 6-bit short literal value extracted from the instruction stream. Special data formatting is required by the Fbox.
- **E%PSL_FU_H**
  The current PSL<FU> value for use by the Fbox in deciding whether to signal floating point underflow faults or not.
- **E%F_MMGT_FLT_H**, **E%F_MEM_ERR_H**, and **E%F_RSVD_ADDR_MODE_H**
  These signals tell the Fbox that there is a fault or error associated with the source operands. The Fbox carries this status down the pipeline so that it is handled after instructions which are already in the Fbox pipeline.
- **E%FLUSH_FBOX_H**
  This signal causes the Fbox to clear its pipeline of all operations.
- **E%RETIRE_OK_H** This signal tells the Fbox whether to stall if it has an instruction to retire. The Fbox stalls if it wants to retire an instruction and this signal is not asserted.
- **E%STORE_OK_H** This signal tells the Fbox whether to stall if it has a result to store. The Fbox stalls if it wants to write a result and this signal is not asserted, even if it also wants to retire an instruction and **E%RETIRE_OK_H** is asserted.

The following signals are driven by the Fbox to the Ebox.

- **F%INPUT_STALL_H**
  This signal causes the Ebox to stall in S3 if it is attempting to send operands to the Fbox.
- **F%STORE_STALL_H**
  This signal is asserted by the Fbox when it is asserting **F%STORE_H** but isn't able to supply valid data.
- **F%FBOX_RESULT_H**
  This 32-bit bus carries Fbox results to the Ebox.
- **F%CC_N_H, F%CC_Z_H, AND F%CC_V_H**
  These are the 3 the Fbox condition code bits. They are Negative, Zero, and Overflow.
- **F%RETIRE_H**
  This control signal tells the Ebox the Fbox is retiring an instruction in this cycle.

- **F%STORE_H**
  This control signal tells the Ebox the Fbox is storing a result in this cycle.
- **F%CC_MAP_H<1:0>**
  This is the map specifier which tells the Ebox how to update the PSL condition code bits.
- **F%FBOX_DL_H<1:0>**
  This is the data length used by the Ebox for an Fbox store.
- **F%MMGT_FAULT_H**
  Signals a memory management fault for one of the currently retiring instruction's source operands.
- **F%MERR_H**
  Signals a memory access hardware error for one of the currently retiring instruction's source operands.
- **F%RSVD_ADDR_MODE_H**
  Signals a reserved address mode fault for one of the currently retiring instruction's source operands.
- **F%RSV_H**
  Signals a reserved operand fault for one of the currently retiring instruction's source operands.
- **F%FOV_H**
  Signals a floating point overflow fault resulted from the currently retiring instruction.
- **F%FU_H**
  Signals a floating point underflow fault resulted from the currently retiring instruction.
- **F%FDBZ_H**
  Signals a floating point divide-by-zero fault resulted from the currently retiring instruction.

8.5.16.8 Fbox Disabled Mode

The ability to operate with the Fbox disabled is provided in the Ebox. When the Fbox is disabled, all floating point macroinstructions, including all floating point CVT macroinstructions, cause reserved instruction faults. MULL is handled in microcode.

The Fbox enable bit is in IPR 125, ECR (see Section 8.5.22). If it is not set, Ebox hardware functions are altered as follows:

- Assertion of **E%FBOX_1ST_CYCLE_L** to the Fbox is disabled (in the Microsequencer).
- The entry made in the retire queue is overridden to specify Ebox instruction retire.
- A reserved instruction fault is signaled to the Microsequencer when the first microword of any Fbox execution microflow is about to advance into S5, except if that microword specifies MISC/MULL.

With the Fbox disabled, each floating point macroinstruction causes a fault (a VAX architecture reserved instruction fault) when the first microword of its execution microflow is about to advance into S5. This occurs for all floating point macroinstructions, including floating point CVT instructions.

Microcode can branch conditionally on the Fbox disable bit. The first microword of the MULL execution microflow specifies MISC/MULL and branches conditionally on the Fbox disable status. If the Fbox is enabled, the branch is to a microflow which dispatches the operation to the Fbox. If the Fbox is disabled, the branch is to an Ebox execution microflow which completes the MULL.

8.5.17 Ebox-Mbox Interface

The Ebox to Mbox interface has a memory request function and a returned read result function. The Ebox issues memory requests by sending a command, address, and possibly write data to the Mbox. The Mbox returns read results by writing them directly into the register file. Faults and errors encountered by the Mbox in completing the operation are reported one of three ways depending on the operation.

NOTE

> When the Ebox initiates a memory read by sending a request to the Mbox, it specifies the register which will receive the memory data in the DST field of the microword. This has the sides effect, when the microword is in S5, of writing that register with the value on **E_BUS%WBUS_L<31:0>**. Normally this register is written by the Mbox after this, before the particular register is read again. However, an exception can prevent the Mbox write and leave the register containing effectively garbage data.

There are three kinds of memory access requests issued by the Ebox: reads, writes, and stores. Reads are requests for memory data to be returned to a Wn or GPR register in the register file. The Ebox supplies the address directly. Writes are requests that data be written to memory. The address and data are both supplied directly by the Ebox. Stores are requests that data be written to the address in the current PA queue entry in the Mbox. The Ebox only supplies the data for stores.

There are several control operations the Ebox can request of the Mbox. There are three kinds of TB invalidate requests. It can synchronize to the Mbox, causing a stall until the Mbox finishes memory management checks for the current request. Also, probe, write check, TB fill, and processor register read and write operations are available.

The Ibox issues operand data reads to MD registers on behalf of the Ebox as it processes operand specifiers. The Ebox simply uses the data when it is returned. The Ibox also issues a request that is the first half of a store. This supplies an address for the Mbox to translate and then enter into the PA queue. The Ebox eventually issues a store request which uses the address in the PA queue to do the write.

Memory management faults encountered in memory reads and writes (not stores) issued by the Ebox are reported by the Mbox asserting the signal **M%MME_TRAP_L** which is received by the Microsequencer. This causes an immediate microtrap and Ebox pipeline abort.

Memory management faults encountered in memory reads initiated by the Ibox on behalf of the Ebox result in the Mbox asserting **M%MME_FAULT_H** which sets the memory management fault status bit associated with the target MD register in the register file. The Ebox detects the fault when a microword sources that particular MD register.

Faults for stores are reported by the Mbox as soon as the PA queue entry is valid. The Ebox detects the fault when a microword attempts to issue a store request.

Hardware errors in memory reads issued by the Ebox are reported by asserting **M%HARD_ERR_H** in the cycle in which read data is written into the register file. The data is generally incorrect, since an error occurred. The register file write can't be to an MD register since it is issued directly by the Ebox. There aren't fault bits in the register file to receive the error status for registers other than the MD registers. So, when the Ebox detects a MD port write to a register other than an MD and the error status is asserted, the Ebox forces an immediate microtrap. This microtrap is not delayed by any S3 or S4 stalls.

Hardware errors in memory reads initiated by the Ibox on behalf of the Ebox result in the Mbox asserting **M%HARD_ERR_H** as it writes the target MD register in the register file. This sets the error status bit associated with the target MD register. The Ebox detects the error when a microword sources that particular MD register.

Hardware errors for stores are reported by the Mbox as soon as the PA queue entry is valid. The Ebox detects the error when a microword attempts to issue a store request.

TB parity errors are a special case. Whenever a TB parity error is encountered, the Mbox asserts **M%TB_PERR_TRAP_L**. The Microsequencer initiates an immediate asynchronous hardware error microtrap when **M%TB_PERR_TRAP_L** is asserted. This could happen as a result of Mbox processing of any Ebox memory reference, Ibox operand prefetch reference, or Ibox instruction fetch or prefetch which uses the TB.

All Mbox requests except store are specified in the MRQ field of the microword. The store request is implicit in Ebox or Fbox result storing through the RMUX. All Mbox requests are issued in S4. The table below shows the requests the Ebox can send to the Mbox. See Chapter 12 for more detail on each operation.

Table 8–13: Ebox Mbox Requests

Request Mnemonic	Addressing	Access Check	Mode Used[1]	Operation Description
MRQ/READ.V.RCHK	virtual	read	current	read virtual memory
MRQ/READ.V.WCHK	virtual	write	current	read virtual memory and check for write access
MRQ/READ.V.NOCHK	virtual	—	—	read virtual memory with no access check
MRQ/READ.V.LOCK	virtual	write	current	read-lock virtual memory
MRQ/READ.P	physical	—	—	read physical memory
MRQ/READ.PR	physical	—	—	read processor register
MRQ/PROBE.V.RCHK	virtual	read	mode	Probe byte address for read - return 3-bit probe status to register file
MRQ/PROBE.V. RCHK.NOFILL	virtual	—	—	Probe byte address for presence in TB - return 1-bit status to register file, but don't fill TB if entry is not already in TB
MRQ/WCHK	virtual	write	current	check that memory location can be written
MRQ/WRITE.V.WCHK	virtual	write	current	write virtual memory
MRQ/WRITE.V.NOCHK	virtual	—	—	write virtual memory without access checks
MRQ/WRITE.V.UNLOCK	virtual	write	current	write-unlock virtual memory
MRQ/WRITE.P	physical	—	—	write physical memory
MRQ/WRITE.PR	physical	—	—	write processor register

[1] Current means CUR_MOD from the PSL, mode means contents of MMGT.MODE.

Table 8–13 (Cont.): Ebox Mbox Requests

Request Mnemonic	Addressing	Access Check	Mode Used[1]	Operation Description
MRQ/PROBE.V.WCHK	virtual	write	mode	Probe byte address for write - return 3-bit probe status to register file
STORE [2]	virtual [3]	write [3]	current [3]	write to physical address in PA queue
MRQ/LOAD.PC	—	—	—	send the data to the Ibox to be used as the new PC
MRQ/SYNC.MBOX	—	—	—	synchronize with memory management check from previous Mbox request by issuing a form of NOP.
MRQ/TB.TAG.FILL	virtual	—	—	directly load TAG part of "current" TB entry
MRQ/TB.PTE.FILL	-	—	—	directly load PTE part of "current" TB entry
MRQ/TB.INVAL.SINGLE	virtual	—	—	invalidate single TB entry, if present
MRQ/TB.INVAL.PROCESS	—	—	—	invalidate all TB entries for current process
MRQ/TB.INVAL.ALL	—	—	—	invalidate all TB entries

[1] Current means CUR_MOD from the PSL, mode means contents of MMGT.MODE.

[2] This operation is not initiated through the MRQ field. It is issued by microwords specifying DST/DST and Fbox operations with F%STORE_H asserted, given that the destination queue entry indicates a memory destination.

[3] Translation and access check done previously by the Mbox.

The store operation in the above table is not specified in the MRQ field. Each destination queue indirect result store which is to memory (as opposed to a GPR) is turned into a Mbox store request. The Mbox writes the data received with this request to the address extracted from the PA queue. (Two address entries in the PA queue are needed for unaligned stores.)

The load-PC operation is accomplished with the aid of the Mbox (MRQ/LOAD.PC). The Mbox's part is to pass the data (PC) on **E%WBUS_H<31:0>** to the Ibox via **M%MD_BUS_H<31:0>**. The Ebox signals the Ibox that the new PC value is coming.

The information sent to the Mbox when the Ebox issues a command is shown in the following table. The information, except **E%WBUS_H<31:0>** data, is valid in S4. The command information is driven early in S4, while the address isn't valid until late in S4. **E%WBUS_H<31:0>** data is valid early in S5. The table shows the source of each item. See Chapter 12 for the encoding of these fields.

Table 8–14: Ebox Memory Request Information Busses

Signal	Source	Description
E%EBOX_CMD_H<4:0>	decoded from MRQ and DST microword fields	Request - command code
E%WBUS_H<31:0>	E_BUS%WBUS_L<31:0>	write data, not ready until S5. (only needed for write type and store operations)
E%VA_BUS_L<31:0>	VA register with bypass	address (or PTE in case of TB.PTE.FILL)
E%EBOX_TAG_H<4:0>	DST microword field	address in register file where read or probe result is to go
E%EBOX_AT_H<1:0>	decoded from MRQ microword field	access type for operation
E%EBOX_DL_H<1:0>	DL register	data length for access
E%EBOX_VIRT_ADDR_H	decoded from MRQ microword field	Indicates virtual address - translation needed
E%NO_MME_CHECK_H	decoded from MRQ microword field	Indicates no access check should be done

This information is all latched by the Mbox in the EM_LATCH. This latch can only hold one command. Once it is full the Mbox will ignore Ebox requests until it is empty again. It is emptied when the Mbox request completes.

To process requests from the Ebox and from the Ibox, the Mbox receives the CUR_MOD bits from the PSL and the MMGT.MODE register contents. The CUR_MOD bits are normally used as the access mode for a request's TB check. The MMGT.MODE bits are used only when the request is a PROBE.V.RCHK, PROBE.V.RCHK.NOFILL or PROBE.V.WCHK. Note that the Mbox uses the CUR_MOD field for all Ibox-initiated requests at all times, so it must receive both mode fields simultaneously.

The address for Ebox-initiated memory accesses comes from the VA register. The microword issuing the memory request may update the VA register. If it does, the new VA value is sent with the request. The write data for a memory request is the data put on **E%WBUS_H<31:0>**, a buffered copy of **E_BUS%WBUS_L<31:0>**, by the microword issuing the memory request.

The following table shows what the Ebox sends on each of the memory request information busses for each operation.

Table 8–15: Ebox Memory Request Information Truth Table

Request Mnemonic	E%EBOX_ CMD_H<4:0>	E%EBOX_ AT_H<1:0>	E%EBOX_ DL_H<1:0>[1]	E%EBOX_ TAG_H<4:0>[2]	E%EBOX_ VIRT_ ADDR_H	E%NO_MME_ CHECK_H	Addr/Data Sent?
READ.V.RCHK	DREAD	read	DL	DST	true	false	yes/no
READ.V.WCHK	DREAD	modify	DL	DST	true	false	yes/no
READ.V.NOCHK	DREAD	read	DL	DST	true	true	yes/no
READ.V.LOCK	DREAD_ LOCK	modify	DL	DST	true	false	yes/no

[1] DL means data length dictated by the microword; the DL register value unless the microword overrides the data length to longword.

[2] DST means the tag is the register specified in the DST field of the microword.

— means don't care, doesn't apply.

Table 8–15 (Cont.): Ebox Memory Request Information Truth Table

Request Mnemonic	E%EBOX_ CMD_H<4:0>	E%EBOX_ AT_H<1:0>	E%EBOX_ DL_H<1:0>[1]	E%EBOX_ TAG_H<4:0>[2]	E%EBOX_ VIRT_ ADDR_H	E%NO_MME_ CHECK_H	Addr/Data Sent?
READ.P	DREAD	read	DL	DST	false	—	yes/no
READ.PR	IPR_RD	—	DL	DST	false	—	yes/no
PROBE.V.RCHK	PROBE	read	Byte	DST	true	false	yes/no
PROBE.V.RCHK. NOFILL	PROBE	0[4]	Byte	DST	true	true	yes/no
WCHK	MME_CHK	write	DL	—	true	false	yes/no
WRITE.V.WCHK	WRITE	write	DL	—	true	false	yes/yes
WRITE.V.NOCHK	WRITE	write	DL	—	true	true	yes/yes
WRITE.V.UNLOCK	WRITE_ UNLOCK	write	DL	—	true	false	yes/yes
WRITE.P	WRITE	write	DL	—	false	—	yes/yes
WRITE.PR	IPR_WR	—	DL	—	false	—	yes/yes
PROBE.V.WCHK	PROBE	write	Byte	DST	true	false	yes/no
STORE	STORE	—	—	—	false	—	no/yes
LOAD.PC	LOAD_PC	—	—	—	false	—	no/yes
SYNC.MBOX	NOP	—	Byte	—	false	—	no/no
TB.PTE.FILL	TB_PTE_ FILL	—	Byte	—	false	true	yes[3]/no
TB.TAG.FILL	TB_TAG_ FILL	—	Byte	—	false	true	yes/no
TB.INVAL.SINGLE	TBIS	—	Byte	—	false	true	yes/no
TB.INVAL.PROCESS	TBIP	—	Byte	—	false	true	no/no
TB.INVAL.ALL	TBIA	—	Byte	—	false	true	no/no

[1] DL means data length dictated by the microword; the DL register value unless the microword overrides the data length to longword.

[2] DST means the tag is the register specified in the DST field of the microword.

[3] PTE data is sent on address bus through VA register.

[4] Special code—no access check is done. Only the presence of an entry in the TB is checked.

— means don't care, doesn't apply.

8.5.17.1 IO Read Synchronization

Because the Ibox issues operand reads before the Ebox executes the associated macroinstruction, there is a possibility that an exception or branch will result in an operand read occurring even though the associated macroinstruction is never executed. This is not a problem if the read is to memory space, but it might be if the read is to IO space. Many IO space reads have side effects, so some mechanism is required which postpones an Ibox issued IO space read until the Ebox is actually executing the macroinstruction which requires the IO space read. The Mbox delays all IO space reads issued by the Ibox until the Ebox asserts the signal **E%START_IBOX_IO_RD_H**.

The Ebox asserts **E%START_IBOX_IO_RD_H** when the following are all true:

1. The Ebox is stalled in S3 waiting for a register file entry indexed through the source queue (i.e., A/S1, A/S2, B/S1, or B/S2) to become valid, or **E_FLQ%FQ_STALL_H** is asserted,
2. there is exactly one entry in the retire queue,
3. there is no stall of S4 of the RMUX part of the Ebox pipeline,
4. conditions 1, 2, and 3 were true in the previous cycle,
5. there is no MD fault for any of the MD registers currently being accessed in (stalled) S3,
6. and the Ebox pipeline is not being flushed by a microtrap this cycle.

The Mbox processes specifier queue entries one at a time (the specifier queue is the queue in the Mbox which receives all operand data references issued by the Ibox). If the specifier queue entry is an IO space access, the Mbox will not process it unless S6 in the Mbox is idle (not processing any reference) and S6 was idle in the previous cycle and **E%START_IBOX_IO_RD_H** is asserted. (Note that a one cycle delay occurs in the Mbox on **E%START_IBOX_IO_RD_H**. This is why the current cycle and the previous cycle are checked for NOP in S6 in the Mbox.) If the Ebox is stalled waiting for read data to be put in an MD by the Mbox, and the Mbox is waiting for **E%START_IBOX_IO_RD_H** to be asserted (because the specifier queue entry is an IO space read) then the Ebox must be waiting for the result of that IO space read.

The Ebox only asserts **E%START_IBOX_IO_RD_H** when it is certain that the macroinstruction which will use the result of the IO space read is going to execute. If the retire queue contains more than one entry, other instructions are in the Ebox or Fbox pipeline so **E%START_IBOX_IO_RD_H** is not asserted in case one of them incurs an exception. If the Ebox is stalled in (RMUX) S4, it doesn't assert **E%START_IBOX_IO_RD_H** because the previous macroinstruction's result store may incur an exception when it advances to S5. (Note that the retire queue entry is removed from the queue before the RMUX S4 stall status is known so that the RMUX S4 stall status has to be examined as well.)

If the Ebox is being flushed by a microtrap in the current cycle, it doesn't assert **E%START_IBOX_IO_RD_H** because the previous macroinstruction actually had a trap.

If there is an MD fault being reported in S3 of the Ebox, then the Ebox will take a microtrap after one cycle with no S4 stalls has passed. In the interim, **E%START_IBOX_IO_RD_H** must not be asserted.

Assertion of **E%START_IBOX_IO_RD_H** when field queue stall is present is necessary to avoid deadlock, however it will cause the CPU to start an IO space operand prefetch even when a memory management fault will cause the instruction to be fault. For example, this might occur with ADAWI if the second operand is in IO space and the first can incur a memory management fault.

8.5.17.2 Mbox-Ebox signals

The Mbox drives the following control signals for Ebox use: **M%EM_LAT_FULL_H** and **M%PA_Q_STATUS_H<2:0>**. **M%EM_LAT_FULL_H** tells the Ebox the EM_LATCH is full. **M%PA_Q_STATUS_H<2:0>** gives the status of the current PA queue entry. **M%PA_Q_STATUS_H<0>** indicates that sufficient entries are valid in the PA queue to accept a store request. Multiple PA queue entries are needed for a store when the store will access multiple longwords in memory (as in quadword length stores and unaligned stores which cross a longword boundary). **M%PA_Q_STATUS_H<1>** indicates that the relevant PA queue entries have a memory management fault associated with them. The Ebox will not issue the store; it will microtrap when the microcode attempts it. **M%PA_Q_STATUS_H<2>** indicates the relevant PA queue entries have a hardware error associated with them. The Ebox will not issue the store; it will microtrap when the microcode attempts it.

In one case Ebox logic ignores **M%PA_Q_STATUS_H<0>** and behaves as if it is deasserted. Due to complexities in the Mbox, **M%PA_Q_STATUS_H<2:1>** are not logically correct in the cycle in which the Ebox aborts a EM_LATCH operation by asserting **E%EM_ABORT_L**. This happens when the Ebox aborts an Fbox result store operation because of **F%STORE_STALL_H** (see Section 8.5.16.3).

Due to complexities in the Mbox, **M%PA_Q_STATUS_H<2:1>**, which signal memory management exceptions and hardware errors associated with the PA queue entry, are not always correct in a cycle in which an EM_LATCH operation is aborted by assertion of **E%EM_ABORT_L**. In this cycle, the Ebox ignores **M%PA_Q_STATUS_H<0>** and behaves as if it is deasserted. **M%PA_Q_STATUS_H<0>** qualifies every use of **M%PA_Q_STATUS_H<2:1>**, so the Ebox can't incorrectly take or not take an exception because of incorrect **M%PA_Q_STATUS_H<2:1>** values.

The Ebox ignores **M%PA_Q_STATUS_H<0>** only in cycles in which a store of Fbox data was sent in the previous cycle and was aborted in this cycle by asserting **E%EM_ABORT_L** because **F%STORE_STALL_H** was asserted. This be coincident with an actual pipeline abort (which also causes assertion of **E%EM_ABORT_L** if a request was sent to the Mbox in the previous cycle). In this case the Ebox will ignore **M%PA_Q_STATUS_H<0>** in a cycle in which the microword in S4 is effectively a NOP, and no change in behavior will result.

The Ebox stalls the microword in S4 if it specifies an Mbox request and the EM_LATCH is full. Also, S4 is stalled if the microword specifies a store and **M%PA_Q_STATUS_H<0>** is not asserted.

The Mbox drives several signals and busses used in writing the data into the register file. These are **M%EBOX_DATA_H**, **M%MD_BUS_H<31:0>**, and **M%MD_TAG_H<4:0>**. When **M%EBOX_DATA_H** is asserted, the data on **M%MD_BUS_H<31:0>** is written into the register addressed by **M%MD_TAG_H<4:0>**. Note that **M%MD_TAG_H<4:0>** is 5 bits; it can address up to 32 locations. The organization of the register file is such that the MD, Wn, and GPR registers (a total of 27 registers) are in the first 32 locations in the register file. This means they can be addressed with a 5-bit tag (which is mapped into the full 6-bit address by zero extension).

The Mbox drives fault and error flags which are associated with the data on **M%MD_BUS_H<31:0>**: **M%MME_FAULT_H** and **M%HARD_ERR_H**. If **M%MME_FAULT_H** or **M%HARD_ERR_H** is asserted when **M%EBOX_DATA_H** is asserted, then a fault or error is being reported to the Ebox for some previously initiated read operation. This is handled in one of several ways, depending on the case, as is shown in Table 8–16.

Table 8–16: Ebox Response to M%MME_FAULT_H and M%HARD_ERR_H

M%MD_TAG_H<4:0> Addresses:	Signal Asserted	Response
Wn or GPR	M%MME_FAULT_H	The Ebox ignores this case. M%MME_TRAP_L would have been asserted for the same fault in a previous cycle.
Wn or GPR	M%HARD_ERR_H	In this case the Ebox forces an immediate hardware error microtrap.
MD	M%MME_FAULT_H	In this case the fault bit for the particular MD is set in the register file.

Table 8–16 (Cont.): Ebox Response to M%MME_FAULT_H and M%HARD_ERR_H

M%MD_TAG_H<4:0> Addresses:	Signal Asserted	Response
MD	M%HARD_ERR_H	In this case the error bit for the particular MD is set in the register file.

The Mbox drives **M%MME_TRAP_L** and **M%TB_PERR_TRAP_L** to force immediate microtraps. **M%MME_TRAP_L** causes a memory management exception microtrap, while **M%TB_PERR_TRAP_L** causes an asynchronous hardware error microtrap.

The Ebox asserts certain Mbox control signals under the control of the MISC and MISC2 fields of the microword. These signals are **E%FLUSH_MBOX_H**, **E%FLUSH_PA_QUEUE_H**, and **E%RESTART_SPEC_QUEUE_H**. **E%FLUSH_MBOX_H** is asserted when MISC/RESET.CPU is specified. It causes the Mbox to flush ongoing Ebox reads, including those initiated by the Ibox. It also flushes the specifier queue. It does not flush the PA queue, so writes and stores already issued by the Ebox are not affected.

E%RESTART_SPEC_QUEUE_H is asserted when MISC/RESTART.MBOX is specified. It restarts Mbox processing of specifier queue references. Mbox specifier queue processing is stopped by Ibox request when certain complex macroinstructions are encountered.

E%FLUSH_PA_QUEUE_H is asserted when MISC2/FLUSH.PAQ is specified. It causes the PA queue in the Mbox to be flushed. MISC2/FLUSH.PAQ should always be sepcified with a MRQ field request which causes an EM latch command (i.e., other than MRQ/SYNC.BDISP, MRQ/SYNC.BDISP.RETIRE, MRQ/SYNC.BDISP.TEST.PRED, or MRQ/NOP).

When a pipeline abort occurs, the Ebox asserts **E%EM_ABORT_L**, conditionally. It is asserted because the abort is recognized too late to prevent the Ebox from issuing an Mbox request in S4. **E%EM_ABORT_L** is asserted in S5 and signals the Mbox to disregard the command just sent in S4. It is only asserted if the Ebox actually made an Mbox request in S4 and the EM_LATCH wasn't full. Even stores and write requests are aborted in this case.

8.5.17.3 Ibox IPR Access and LOAD PC

The Ebox detects Ibox IPR access requests in S5. At that time it asserts a command strobe to the Ibox. The Mbox will also detect that the IPR access is to the Ibox. It will treat an Ibox IPR read as a NOP. For IPR writes the Mbox forwards the data on **M%MD_BUS_H<31:0>** in a later cycle. Microcode synchronizes with Ibox IPR writes by issuing a MRQ/SYNC.MBOX after the operation. Once the MRQ/SYNC.MBOX is complete, the microcode knows the Ibox has the data.

In detecting Ibox IPRs,. the Ebox treats the entire range of normal IPR addresses from D0 to DF (hex)as Ibox IPRs. The exact test used by the Ebox is: VA<9:6>=D (hex) and VA<24>=0. The low four bits (VA<5:2>) are sent to the Ibox so it can determine which of its IPRs is specified.

The Ebox requests a load-PC Mbox operation in S4 when the microword specifies LOAD.PC in the MRQ field. In S5 of that microword it asserts a command strobe to the Ibox informing it that the Mbox will soon forward the new PC value. Microcode synchronizes with the load-PC operation by specifying a SEQ.MUX/LAST.CYCLE. The instruction queue must be empty at this time. Once the Ibox adds a new instruction queue entry, a macroinstruction dispatch occurs. While waiting, the Ebox executes a continuous stream of "STALL" microwords (see Section 8.5.20.1).

8.5.18 Ebox Vector Support

The Ebox supports potential future vector architecture integration by providing a configuration status bit which is available for microcode conditional branches.

VECTOR_UNIT_PRESENT is a configuration status bit for vector support in the IPR ECR. See Section 8.5.22. Microcode can branch conditionally on the VECTOR_UNIT_PRESENT status.

8.5.19 Fault and Trap Management

There are three kinds of VAX Architecture exceptions: faults, aborts, and traps. In all cases the PC, PSL, and other data is pushed on the stack, and the address of an exception handling routine is fetched from the SCB. For a trap, the instruction which caused the trap has finished completely, and the PC on the stack points to the next instruction to execute. For a fault, the PC on the stack points to the instruction which caused the exception. For an abort, the PC, PSL, and other state are UNPREDICTABLE; however, whenever possible the NVAX CPU tries to turn aborts into faults. The difference between an abort and a fault is that no important architectually visible state was modified by the instruction if it was a fault, while some important architecturally visible state may have been modified if it was an abort. (Certain state, for example, the memory location which is pointed to by the stack pointer, can be modified in the case of a fault. Generally speaking, aborts are cases where restarting the instruction may not work because some state which the instruction depended on may have been altered.) The VAX Restart Bit in the machine check stack can be used in determining whether it is safe to treat an abort as a fault.

To cleanly support the concepts described in the previous paragraph, the NVAX CPU has a macroinstruction commit point in the pipeline. Once any microword of the execution microflow has passed this point, the macroinstruction may have modified architectural state. Until the first microword of the microflow passes the commit point, the instruction cannot have modified any architectural state. This point is the boundary between S4 and S5 in the Ebox pipeline. No architecturally visible state is ever modified in S3 or S4 of the pipeline. For example, the PSL and all registers in the register file are written only in S5. Also, memory requests are not issued until a microword specifying one is about to advance into S5, and it is certain there are no S4 stalls.

Each macroinstruction execution microflow obeys the restriction that no microword in that flow modifies any architectural state before it is certain that all the operand specifiers for the instruction have been properly fetched and decoded and that all the memory accesses which this microflow will request are not going to encounter a memory management violation. This does not mean that no microword of the microflow passes the S4/S5 boundary before all this is checked. It only means that the microwords in the microflow don't write memory or any other architecturally visible state until these things are verified. The net result is that macroinstructions which encounter a memory management violation are restartable once the condition has been corrected. (Note that the string instructions don't quite follow these simple rules. Instead, they use a more elaborate set of rules to ensure that they can be restarted after any memory management fault.)

Microflows for macroinstructions which might encounter any kind of fault other than a memory management exception specifically test for the fault condition(s) before modifying any architectural state. This is in addition to checking for memory management faults, as described above.

Ebox hardware forces a reserved opcode fault for Fbox instructions (except MULL) when the Fbox is disabled, in S4 of the first microword of Fbox macroinstruction execution flows. Because this fault is requested in S4 of the first microword, it prevents any architectural state from being altered by these flows.

Hardware errors are handled differently. They generally can't be checked for, and the architecture doesn't require any such checks. Generally aborts occur as a result of a hardware error encountered in an macroinstruction after all memory management checks have been done. In these cases, some architecturally visible state may have been modified before the macroinstruction has completed.

8.5.19.1 Faults and Errors Detected in S4

When the Ebox detects a fault or error condition in S4 associated with an operation that is about to advance into S5, it signals the Microsequencer to cause a microtrap. The microtrap will cause the Ebox pipeline to abort before it advances. Any operation which was in S5 already completes normally, but the operation in S4 is purged before it enters S5. The operation in S5 may be part of a previous macroinstruction microflow. That macroinstruction is not affected by the microtrap. The microword in S4 may be the first microword to modify architecturally visible state in a given execution microflow so it must be prevented from advancing into S5.

8.5.19.1.1 Coordinating Ebox and Fbox Faults and Errors

It is necessary that macroinstructions retire in order, even when there is a fault or error detected in S4. The microtrap for the fault or error must be delayed until the macroinstruction connected to the fault or error is next to retire. The current retire queue entry is used by the Ebox to decide whether a microtrap should be signaled. For example, if a branch displacement access fault or error is detected by the Ebox in S4 but the retire queue indicates the Fbox is next to retire a macroinstruction, then the branch macroinstruction came after the one being executed in the Fbox. The branch's fault or error must not cause a microtrap until the Fbox has retired its macroinstruction. Then the microtrap is forced, given that the next entry in the retire queue indicates the Ebox is next to retire a macroinstruction. The microtrap occurs in S4 after the Fbox's last operation advances into S5. The branch is prevented from retiring by the microtrap, since it incurred a fault or error.

The Fbox reports a number of faults and one error to the Ebox. The Ebox ignores them until the retire queue indicates the Fbox is next to retire a macroinstruction. The reason is the same as in the previous paragraph. The microtrap has to be delayed until the logically preceding macroinstructions are advanced into S5.

Destination queue and PA queue faults and errors can be connected either to the Ebox or the Fbox. It depends on whether the box selected by the retire queue is requesting a destination queue indirect store. If the destination queue is empty and **I%IMEM_MEXC_H**, **I%IMEM_HERR_H**, or **I%RSVD_ADDR_FAULT_H** is asserted and the box indicated by the retire queue is requesting a destination queue store, then a microtrap is signaled immediately. Also, if a destination queue store is requested while the current destination queue entry is valid and **M%PA_Q_STATUS_H<1>** or **M%PA_Q_STATUS_H<2>** is asserted, a microtrap is taken (see Section 8.5.17).

8.5.19.1.2 Breaking the S4 Stall

Other than the requirement that instructions retire in order, S4 stalls do not delay microtraps for faults or errors which are in S4. In other words, any S4 stall is broken once a fault or error in S4 is due to the next macroinstruction to retire.

8.5.19.2 Faults and Errors detected in S3

When the Ebox detects a fault or error condition in S3, it latches it in order to carry it down the pipeline to S4. Unlike most control signals propagating down the pipeline, these fault and error conditions are not forced off when S3 is stalled and S4 isn't stalled. So the S3 stall doesn't have to end for the fault/error condition to propagate to S4. However, the fault/error conditions do stall in S3 if there is an S4 stall. This is because the microword in S4 may be from a previous macroinstruction. That instruction must be allowed to complete normally before the microtrap. Once the fault or error status has advanced into S4 and the retire queue indicates the Ebox is next to retire a macroinstruction, the Ebox signals the Microsequencer to microtrap.

8.5.19.3 Integer Overflow and Branch Mispredict Traps

There are two traps handled in Ebox hardware. They are integer overflow traps, and branch misprediction traps. Integer overflow traps are VAX Architecture exceptions, while branch misprediction traps are not part of the VAX architecture. Both of these traps are handled in the Ebox by causing a microtrap once the last microword of the macroinstruction's execution microflow has entered S5. The microtrap prevents the next microword (which is the first microword of a new microflow) from advancing into S5. This means that the macroinstruction in question completes properly but its successors are not allowed to execute. This is done for integer overflow because this is the effect required by the VAX Architecture. It is done for branch misprediction because this is the effect required to recover from an incorrectly predicted conditional branch.

Integer overflow traps occur when a microword which specifies SEQ.MUX/LAST.CYCLE.OVERFLOW is in S5 and PSL<IV> and PSL<V> are both set. If a microtrap is signaled, it prevents the next microword (or Fbox operation) from advancing into S5; the current operation in S5 completes regardless of whether the microtrap is signaled.

Of the VAX architecture instructions which can cause integer overflow, MULL and all the CVT instructions are executed in the Fbox (except that MULL is executed in the Ebox when the Fbox is disabled). Integer overflow is detected in the Fbox for these instructions. The Ebox determines that an integer overflow occurred by examining the new PSL<V> bit for every Fbox retiring instruction. To distinguish instructions which can incur integer overflow traps from others the Fbox might retire, the Ebox checks the map specifier supplied by the Fbox. MULL and CVTs with integer destinations all use the same map specifier, and no other Fbox executed instruction uses that particular specifier. When the instruction being retired by the Fbox uses that particular map specifier, and PSL<IV> and PSL<V> are both set, the Ebox forces the microtrap for integer overflow.

Branch misprediction traps are taken in S5 when the microword specifies SYNC.BDISP.TEST.PRED and the branch condition evaluator determines that the branch was incorrectly predicted. The Ibox prediction is read from the branch queue in S4. The branch condition evaluator result is available in S5. If the prediction doesn't match the actual result, a branch misprediction microtrap is signaled. The microtrap will prevent the microword in S4 from completing. That microword may have been the first microword of the execution microflow for the next macroinstruction. It is not supposed to be executed because the Ibox incorrectly predicted the outcome of the conditional branch. For more on mispredicted branches see Section 8.5.15.5.

If a branch mispredict is detected at the same time as an integer overflow, the integer overflow microtrap is taken. See Section 8.5.19.5.

8.5.19.4 Ebox Microtrap Handling

The Ebox makes a microtrap request by asserting one of a number of microtrap request signals. The Microsequencer causes a microtrap at the end of the current cycle. The Microsequencer has a priority encoder which it uses to decide which microtrap dispatch should be taken when more than one microtrap request is asserted (see Chapter 9). Regardless of which microtrap is taken, the signal **E_USQ%PE_ABORT_L** is asserted, causing an effective no-op to be inserted into all the control latches in S3, S4, and S5. The result is a pipeline abort.

Early in a pipeline abort cycle (the cycle in which all the control latches in the pipeline are flushed), the Microsequencer signals asserts **E_USQ%PE_ABORT_L**. The Ebox responds by flushing the retire queue and the Ebox pipeline. Also, if in the last cycle a new command had been accepted by the Mbox, the Ebox asserts **E%EM_ABORT_L** which aborts that command. (**E%EM_ABORT_L** will abort any EM_LATCH entry.)

In the case of a branch mispredict microtrap, the Ibox has already been signaled by the Ebox that a mispredict occurred. The Ibox has the alternate PC latched, and it will begin fetching from that location as soon as it has unwound the RLOG. See Chapter 7 for more detail.

All microtrap flows except branch mispredict execute a RESET.CPU. This causes a flush or reset of the Ebox queues and register file valid bits, the Fbox, and the Mbox (except the PA queue and EM_LATCH). It also causes **E%STOP_IBOX_H** to be asserted. These microtrap flows then read the Ibox IPR which causes the RLOG to be unwound and returns the backup PC.

The branch mispredict microflow doesn't execute a RESET.CPU because the Ibox automatically recovers from the branch mispredict and begins fetching instructions from the correct memory location. For the same reason, it does not read the Ibox IPR which causes the RLOG to be unwound and returns the backup PC. For branch mispredict, Ebox hardware asserts all the flush or reset signals that MISC/RESET.CPU would have caused except that **E%STOP_IBOX_H** is not asserted.

All microtrap flows synchronize with the Mbox by executing MRQ/SYNC.MBOX. Then they execute a MISC2/FLUSH.PAQ which causes the PA queue in the Mbox to be flushed. This allows any stores which were pending in the EM_LATCH to be finished before the PA queue is flushed.

Certain microcode rules and restrictions apply to the process of gathering state and flushing the various boxes and function units within boxes. See Section 8.5.27.18.

8.5.19.5 Coincidence of Branch Mispredict Trap with other Traps

It is possible for a branch mispredict trap to happen at the same time as an integer overflow trap. When this occurs, the integer overflow trap is taken because it has higher priority than branch mispredict. However, the Ibox is still signaled that a branch mispredict took place. In the few cycles that it takes for the MISC/RESET.CPU in the integer overflow microflow to arrive at S5 in the Ebox pipeline, the Ibox has begun unwinding the RLOG and correcting the backup PC queue. Once the Ibox starts this process, it delays its own response to the **E%STOP_IBOX_H** signal (which is asserted by MISC/RESET.CPU) until it has completed the correction process for the mispredicted branch. In this way, the correct backup PC is made available to the integer overflow microflow.

It is also possible for a trace fault to follow a mispredicted branch. In this case, the branch mispredict trap flushes the pipeline (purging the microflow for a trace fault which is following it down the pipeline) and the Ibox unwinds the RLOG and corrects the backup PC queue. Then the branch mispredict microflow executes a LAST.CYCLE which causes the Microsequencer to dispatch to the trace fault handler. Early in the trace fault microflow, RESET.CPU will be executed, and so **E%STOP_IBOX_H** may be asserted to the Ibox before it has finished correcting for the mispredicted branch. The Ibox's ability to delay its response to **E%STOP_IBOX_H** is what allows the Ibox to finish its corrective action.

8.5.19.6 Possible Microtrap Requests

The following table lists the microtrap requests the Ebox can make.

Table 8–17: Ebox Microtrap Requests

Microtrap	When Asserted	Sources	Signal
Memory management fault	S4	Ibox signal, MD fault status bits, PA queue fault bit, or indicated by Fbox signal, F%MMGT_FAULT_H	E_FLT%MME_ERR_H
Memory access error	S4	Ibox signal, MD fault status bits, or indicated by Fbox signal, F%MERR_H	E_FLT%HW_ERR_H
Reserved addressing mode	S4	Ibox signal or indicated by Fbox	E_FLT%RSVD_ADDR_MODE_H
Reserved operand fault	S4	Indicated by Fbox	E_FLT%FLOATING_FAULT_H
Reserved instruction fault	S4	For floating point macroinstructions when the Fbox is not enabled.	E_FLT%RSVD_INSTR_L
Branch misprediction trap	S5	branch result mismatch	E%BRANCH_MISPREDICT_L
Integer overflow trap	S5	PSL<V> and PSL<IV> both set, and SEQ.MUX/LAST.CYCLE.OVERFLOW or Fbox map specifier indicates integer result	E_FLT%IOVFL_L
Floating overflow fault	S4	Indicated by Fbox	E_FLT%FLOATING_FAULT_H
Floating underflow fault	S4	Indicated by Fbox	E_FLT%FLOATING_FAULT_H
Floating divide-by-zero fault	S4	Indicated by Fbox	E_FLT%FLOATING_FAULT_H

8.5.19.7 Fbox Fault Reporting

The four Fbox faults, reserved operand, floating overflow, floating underflow, and floating divide-by-zero all cause the same dispatch in the Microsequencer. The Ebox latches a priority encoded status when one of these faults is reported by the Fbox. This status is available to the trap handler via a microbranch. The priority order, from highest to lowest, is reserved operand, floating divide-by-zero, floating overflow, and floating underflow. Table 8–18 shows the code for each of the four fault conditions.

Table 8–18: Fbox Fault Codes

Fault	Priority	Code
Reserved operand	1	0
Floating divide-by-zero	2	1
Floating overflow	3	2
Floating underflow	4	3

8.5.20 Ebox Stalls

The Ebox pipeline is controlled by the Ebox stall logic. It supplies stall signals which gate clocking of data information into each pipeline stage. The Ebox stall logic stalls only the segments which must be stalled. S5 is never stalled. S3 stalls if S4 is stalled. If S3 is stalled but S4 is not stalled, a NULL microword (or, more generally, an effective no-op) is injected into S4 after the control information in S4 advances into S5.

The clock for each pipeline latch in S3 and S4 is Φ_1 gated by a stall control signal. The stall control signals are **E%S3_STALL**, **E%S4_STALL**, and **E%RMUX_S4_STALL** for stages S3, S4, and RMUX S4 respectively. These signals determine whether the corresponding latches are opened in Φ_1.

The stall control signals are used to stall a pipe stage. A stage is stalled when it cannot complete its operation for some reason. Generally data needed by the stage is not yet valid, but is expected to become valid after some time. Also stage N will be stalled when stage N+1 is not ready to receive the output of stage N.

The Ebox pipeline can be stalled while the Fbox uses the RMUX portion of the pipeline to store results. When the Fbox is next to retire an instruction, **E%RMUX_S4_STALL**, **E%RMUX_S4_FLUSH**, and **E%RMUX_S5_FLUSH** depend on the progress of Fbox result store operations. When the Ebox is next to retire, these signals are driven to the same logic level as **E%S4_STALL**, **E%S4_FLUSH**, and **E%S5_FLUSH**, respectively.

The clock for S5 pipeline latches is not gated. However there is an S5 flush signal for control information and another flush signal for the output of the RMUX.

The S3, S4, and S5 pipeline latches which hold control information also have an asynchronous reset input signal: **E%S3_FLUSH**, **E%S4_FLUSH**, **E%RMUX_S4_FLUSH**, **E%S5_FLUSH**, and **E%RMUX_S5_FLUSH**. These signals clear (flush) the control information to an effective no-op. They are asserted after the clock which loads the latch but before the control information is used to alter any state in the Ebox or anywhere else in the NVAX CPU.

The flush control signals are used to insert effective no-ops into a particular stage. This is done for two distinct reasons. First, when pipeline stage N is stalled but stage N+1 is not stalled, an effective no-op is inserted into stage N+1 as its current operation advances to stage N+2. Secondly, when a pipeline flush is needed, the flush signals are all asserted, so every stage of the pipeline has an effective no-op inserted. The Ebox flushes the pipeline when the Microsequencer asserts **E_USQ%PE_ABORT_L** (which indicates that a microtrap dispatch has been initiated).

Figure 8–8 shows control and data path latches and how the various pipeline control signals are typically connected.

Figure 8–8: Ebox Pipeline Latches

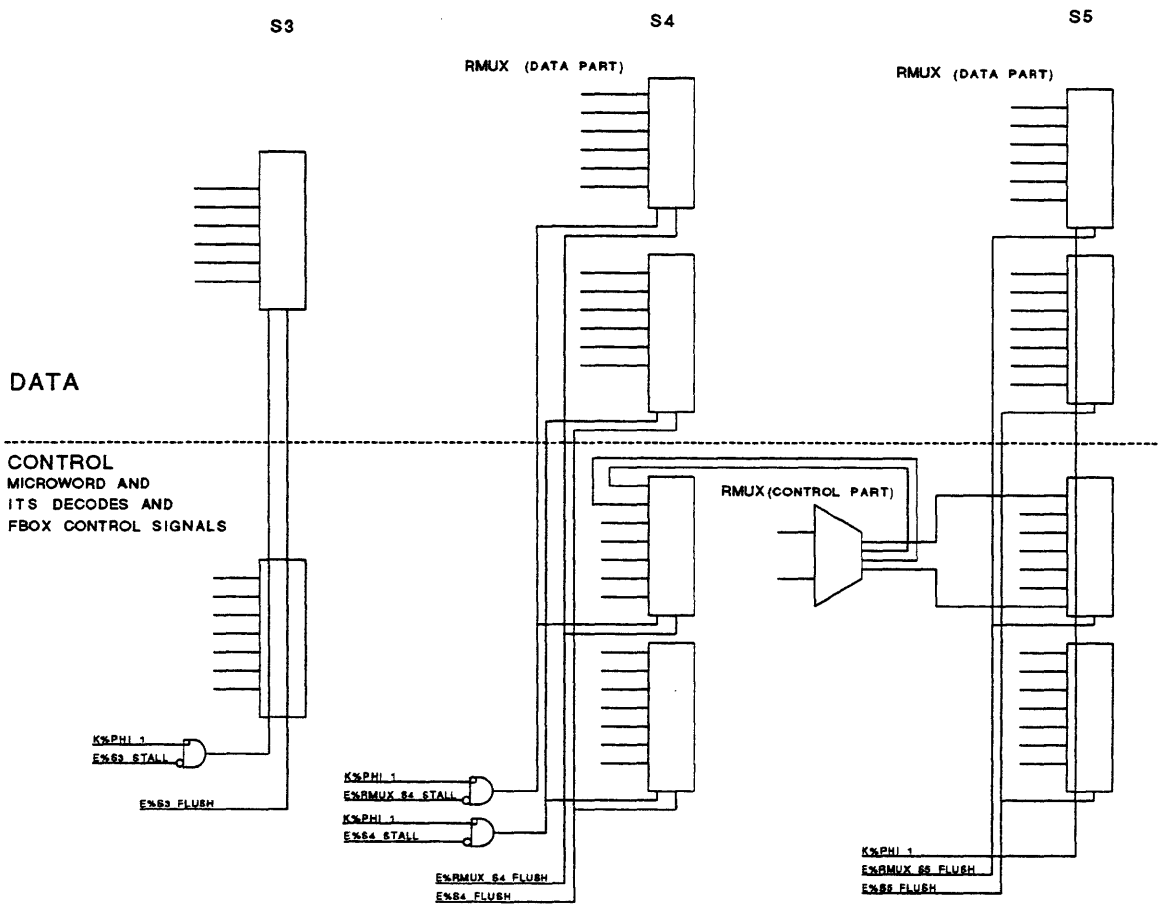


Table 8–19 shows the various pipeline stall and flush combinations which can occur. An important factor in determining the pipeline controls is whether the Ebox or Fbox is next to retire a macroinstruction. This status is given by the current retire queue entry.

Table 8–19: Ebox Pipeline Stall and Flush Cases

	Ebox Next to Retire a Macroinstruction			
Pipeline Control Case	S3 Clock/ S3 Flush	S4 Clock/ S4 Flush	RMUX S4 Clock/ RMUX S4 Flush	S5 Flush/ RMUX Flush
No Stalls	run/don't flush	run/don't flush	run/don't flush	don't flush/don't flush
S3 Stall (with no S4 stall)	stall/don't flush	run/flush	run/flush	don't flush/don't flush
S4 Stall	stall/don't flush	stall/don't flush	stall/don't flush	flush/flush

Table 8–19 (Cont.): Ebox Pipeline Stall and Flush Cases

Ebox Next to Retire a Macroinstruction

Pipeline Control Case	S3 Clock/ S3 Flush	S4 Clock/ S4 Flush	RMUX S4 Clock/ RMUX S4 Flush	S5 Flush/ RMUX Flush
Pipeline Abort	run/flush	run/flush	run/flush	flush/flush

Fbox Next to Retire a Macroinstruction[1]

Pipeline Control Case	S3 Clock/ S3 Flush	S4 Clock/ S4 Flush	RMUX S4 Clock/ RMUX S4 Flush	S5 Flush/ RMUX Flush
Ebox requesting RMUX (in S4)[2], or Ebox S4 stall	stall/don't flush	stall/don't flush	see note [3]/ don't flush	flush/see note[4]
Ebox not requesting RMUX (in S4) and no S3 stall[2]	see note[5]/don't flush	see note[5]/don't flush	see note [3]/ don't flush	don't flush/see note[4]
Ebox not requesting RMUX (in S4) with S3 stall[2]	stall/don't flush	see note[5]/flush	see note [3]/ don't flush	don't flush/see note [4]

[1] If Fbox is next to retire a macroinstruction, then the RMUX always selects the Fbox even if the Fbox doesn't request it.

[2] The Ebox is requesting the RMUX if the microword in S4 specifies anything other than NONE in the DST field.

[3] Run if Fbox not requesting RMUX or if Fbox is requesting and there is no stall on the operation. Stall if Fbox is requesting a store and/or retire and there is a stall on the operation.

[4] Don't flush if Fbox not requesting RMUX or if Fbox is requesting and there is no stall on the operation. Flush if Fbox is requesting a store and/or retire and there a RMUX S4 stall on the operation.

[5] Stall if RMUX S4 clock is stalled. Otherwise run.

As is shown in Table 8–19, when an effective no-op is inserted into S4 during an S3 stall, S5 does not need to be flushed. The effective no-op in S4 will propagate into an effective no-op in S5.

VERIFICATION NOTE

> The interaction between stalls and microbranches is different than Rigel. That all microbranch tests work properly when S3 is stalled and S4 is not stalled should be verified carefully.

8.5.20.1 The STALL Microword

In any cycle that the instruction queue is empty (and the Ibox is not providing a bypassed instruction queue entry directly to the Microsequencer), the Microsequencer fetches the STALL microword. This microword specifies no operation, except SEQ.MUX/LAST.CYCLE, and can't cause a stall anywhere in the pipeline. This allows the microwords already in the pipeline to continue even when the Ibox is temporarily unable to supply new instruction execution dispatches. See Chapter 9 for more detail.

8.5.20.2 Field Queue Stall

When microcode uses the field queue, it executes a 4-way conditional microcode branch on two conditions, a not-empty condition and the 1-bit status in the current field queue entry. Only three of the 4 branch outcomes are actually possible, because the output of the field queue is forced off if the queue is empty. The Ibox makes an entry into to the field queue when it processes a field operand. While the queue is empty, the microcode loops continuously repeating the same conditional branch. This is very much like a stall condition in that the pipeline stages all have the same operation in them in every cycle while the field queue remains empty. See Section 8.5.15.8 for more on field queue operation.

8.5.20.3 Ebox Stall Conditions

The Ebox stall logic detects the need for stalls in various parts of the pipeline. The stalls must be detected on time to gate Φ_1 latches at the start of the next cycle. This section assumes the Ebox is next to retire a macroinstruction. The next section deals with stalls with the Fbox next to retire a macroinstruction.

The Ebox pipeline stalls in S3 when it is accessing some data in the register file which is not valid or when it requires an entry in the source queue which is not available. Up to two source queue entries and up to two MD or Wn registers can be accessed at once. The S3 stall lasts until all the accessed elements are valid and available.

Wn and MD registers have valid bits associated with them. A register is valid only if this bit is set. A register's valid bit is not set when a memory read has been initiated for that register and hasn't yet completed. The valid bit is set by the Mbox when the read completes.

The source queue read and write pointers are examined to determine when there are sufficient source queue entries to satisfy the microword in S3. Either one or two entries might be needed. Only one is needed if the source queue is referenced in the A or B microword fields but not both. Two are needed if the source queue is referenced in both microword fields. The Ebox stalls in S3 if exactly the number of entries needed aren't present. In particular, if only one entry is needed, then the Ebox only stalls if the source queue is completely empty, and if two entries are needed, the Ebox stalls until two entries are made.

The Ebox stalls in S3 if the microword in S3 is sending operands to the Fbox and the Fbox is indicating that it can't accept the any more operands.

The Ebox stalls in S3 if the microword in S3 is accessing at least one GPR which is marked in the Fbox destination scoreboard as having an Fbox result store pending.

Given that the retire queue indicates the Ebox is next to retire a macroinstruction, the Ebox stalls in S4 if the following are true:

- The microword in S4 specifies DST/DST.
- The destination queue is empty, **or** the destination queue isn't empty, the destination queue entry indicates a memory store, and the current PA queue entry is not valid.

The destination queue read and write pointers are examined to determine when the destination queue is empty. The current PA queue entry is valid when the Mbox has completed memory management checks for the store reference. The Mbox asserts **M%PA_Q_STATUS_H<0>** when the PA queue entry is valid.

The Ebox stalls in S4 if the microword in S4 initiates a memory operation and the Mbox is already working on an Ebox-initiated memory operation. The EM_LATCH in the Mbox holds the current Ebox memory request. It is not available until the Mbox has finished that request. The Mbox provides a status which informs the Ebox that the EM_LATCH is empty.

Destination queue indirect stores that are memory stores go in the EM_LATCH like any other Ebox memory access. So EM_LATCH-full S4 stalls can occur even when the microword in S4 specifies MRQ/NOP.

The Ebox stalls in S4 if the microword in S4 synchronizes with the branch queue and the branch queue is empty. The branch queue read and write pointers are examined to determine when the branch queue is empty.

The Ebox stalls in S4 if the microword in S4 specifies MISC2/FDEST.CHECK and the entry in the destination queue needed to complete this operation is not yet valid. This stall ends when the Ibox writes the needed entry.

The destination queue has a second access pointer, the FDest pointer. This pointer is compared to the destination queue write pointer to determine when the entry needed for the MISC2/FDEST.CHECK is available.

When it is next to retire an instruction, the Fbox can cause an S4 stall by asserting **F%STORE_STALL_H**, indicating that the Fbox is stalling for this cycle because the data on **F%FBOX_RESULT_H** is incorrect or there is a data exception to be evaluated in the Fbox's last stage. **F%STORE_STALL_H** is only supposed to be asserted if the Fbox is storing a result (i.e., **F%STORE_H** is asserted).

8.5.20.4 Fbox and RMUX Related Stall Conditions

The Ebox has several Fbox related stalls. When the Fbox requests the RMUX the Ebox may have to stall the Fbox. Also, depending on which box (Fbox or Ebox) is next to retire a macroinstruction, several different Ebox stalls may occur.

NOTE

> When the microcode needs to stall in S3 waiting for an Fbox operation to complete, one or two microwords which specify DST/WBUS should precede the microword needing the Fbox operation to be complete. Any microword specifying DST/WBUS will stall in S4 until the Fbox retires its instruction. The appropriate amount of delay depends on which result is being awaited.

The Ebox stalls in S4 if the current retire queue entry specifies that the Fbox is next to retire a macroinstruction and the Ebox is requesting the RMUX. The Ebox is requesting the RMUX if the microword in S4 specifies anything other than NONE in the DST field. Otherwise it is not requesting the RMUX.

The Ebox stalls the Fbox (by asserting a stall signal before the end of the cycle) when the Fbox is requesting the RMUX and one of the four following is true (note that if the Fbox is next to retire, the RMUX portion of the Ebox pipeline is stalled whenever the Ebox stalls the Fbox):

- The Ebox is next to retire a macroinstruction.
- The Fbox is next to retire a macroinstruction, is requesting to use the destination queue, and the current destination queue entry is not valid.

- The Fbox is next to retire a macroinstruction, is requesting to use the destination queue, the current destination queue entry is valid and indicates a memory destination, and the PA queue is not valid.
- The Fbox is next to retire a macroinstruction, is requesting to use the destination queue, the current destination queue entry is valid and indicates a memory destination, the PA queue is valid, and the EM_LATCH is full.

The Ebox determines all these conditions as described in the previous section. No part of the Ebox pipeline is stalled by an Fbox request if the Ebox is next to retire a macroinstruction.

The Fbox can cause an RMUX S4 stall by asserting **F%STORE_STALL_H**, indicating that the Fbox is stalling for this cycle because the data on **F%FBOX_RESULT_H** is incorrect or there is a data exception to be evaluated in the Fbox's last stage. (This also causes an S4 stall.) **F%STORE_STALL_H** is only supposed to be asserted if the Fbox is storing a result (i.e., **F%STORE_H** is asserted).

The Ebox is always stalled in S4 if an RMUX S4 stall occurs.

8.5.21 Miscellaneous Operations

The microword allows for a number of miscellaneous control and data movement operations. Most of them have been described elsewhere in this chapter, and are only summarized here. The following table lists all the miscellaneous operations by microword field and gives a description. Any of these fields can also specify NOP (no operation).

Table 8–20: Ebox Miscellaneous Operations

Mnemonic	Description
	MISC Field - Both Standard and Special Microword Formats
DL.BYTE	DL <-- byte; change effects next microword
DL.WORD	DL <-- word; change effects next microword
DL.LONG	DL <-- long; change effects next microword
RESTART.IBOX	restart Ibox operand specifier parsing in S5
RESTART.MBOX	restart Mbox operand processing in S5
RESET.CPU	flush Mbox and Fbox, initialize register file valid bits, flush Ebox queues, all in S6; stop Ibox in S5
CLR.PERF.COUNT	Clear the performance counters in S5. See Chapter 18
INCR.PERF.COUNT	Increment a performance counter in S5 if ECR<PMF_EMUX> is a certain value. See Chapter 18
CLR.STATE.3-0	clear flags<3:0>; change effects next microword
SET.STATE.0	set flag<0>; change effects next microword
SET.STATE.1	set flag<1>; change effects next microword
SET.STATE.2	set flag<2>; change effects next microword
MULL	disables reserved instruction fault normally generated for Fbox instructions when the Fbox is not enabled. Used in MULL2 and MULL3 so microcode can execute the macroinstruction instead of the Fbox.

Table 8–20 (Cont.): Ebox Miscellaneous Operations

MISC Field - Both Standard and Special Microword Formats

Mnemonic	Description
CONST.10.BIT	Special constant generation mode. See Section 8.5.2
LOAD.SC.FROM.A	SC <– E_BUS%ABUS_L<4:0>
LOAD.MPU.FROM.B	MPU <– E_BUS%BBUS_L<29:16>
LOAD.PSL.CC.IIIP	update PSL CCs: PSL<N,Z,V> <– S5 Condition Codes <N,Z,V> PSL<C> <– PSL<C> (Unchanged)
LOAD.PSL.CC.JIZJ	update PSL CCs: PSL<N> <– S5 Condition Code <N> .XOR. S5 Condition Code <V> PSL<Z> <– S5 Condition Code <Z> PSL<V> <– 0 PSL<C> <– .NOT. S5 Condition Code <C>
LOAD.PSL.CC.IIII	update PSL CCs: PSL<N,Z,V,C> <– S5 Condition Codes <N,Z,V,C>
LOAD.PSL.CC.IIIJ	update PSL CCs: PSL<N,Z,V> <– S5 Condition Codes <N,Z,V> PSL<C> <– .NOT. S5 Condition Code <C>
LOAD.PSL.CC.IIIP.QUAD	update PSL CCs: PSL<Z> <– PSL<Z> .AND. S5 Condition Code <Z> PSL<N,V> <– S5 Condition Codes <N,V> PSL<C> <– PSL<C> (Unchanged)
LOAD.PSL.CC.PPJP	update PSL CCs: PSL<N,Z,V> <– PSL<N,Z,V> (Unchanged) PSL<V> <– .NOT. S5 Condition Code <Z>
CLR.VECT.RDY	S3 clear of VECTOR_RDY condition. See Section 8.5.18

MISC1 Field - Special Format Microword

Mnemonic	Description
RETIRE.INSTRUCTION	generate Ibox retire instruction signal in S5
FLUSH.VIC	flush Ibox virtual instruction cache in S5
FLUSH.BPC	flush Ibox branch prediction cache in S5
FOP.VALID	Fbox operand on E%ABUS_H<31:0> and E%BBUS_H<31:0> or both
FLUSH.PCQ	Flush PC queue in Ibox
CLR.STATE.5-4	clear flags<5:4>; change effects next microword
SET.STATE.3	set flag<3>; change effects next microword
SET.STATE.4	set flag<4>; change effects next microword
SET.STATE.5	set flag<5>; change effects next microword

Table 8–20 (Cont.): Ebox Miscellaneous Operations

MISC2 Field - Special Format Microword

Mnemonic	Description
F.DEST.CHECK	Access destination queue and make entry in Fbox destination scoreboard
FLUSH.PAQ	Flush PA queue in Mbox

MRQ Field - Both Standard and Special Format Microwords

Mnemonic	Description
SYNC.BDISP	stall if branch displacement invalid in S4; microtrap if fault
SYNC.BDISP.RETIRE	stall if branch displacement invalid in S4; microtrap if fault; S5 retire entry
SYNC.BDISP.TEST.PRED	stall if branch displacement invalid in S4; microtrap if fault; S5 microtrap if mispredict and retire entry
LOAD.PC	load new PC (always followed by MISC/RESTART.IBOX)

DISABLE.RETIRE Field - Special Format Microword

Mnemonic	Description
YES	Disable the retire macroinstruction and retire retire queue entry effects of SEQ.MUX/LAST.CYCLE and SEQ.MUX/LAST.CYCLE.OVERFLOW
NO	Enable the retire macroinstruction and retire retire queue entry effects of SEQ.MUX/LAST.CYCLE and SEQ.MUX/LAST.CYCLE.OVERFLOW

The MISC1/RETIRE.INSTRUCTION function signals the Ibox to retire an instruction in order to bring the backup PC queue and the RLOG into the correct state for restoring GPRs and providing the backup PC after a microtrap. It does not retire a retire queue entry. Therefore MISC1/RETIRE.INSTRUCTION must always be followed by a MISC/RESET.CPU before the next macroinstruction execution dispatch (via SEQ.MUX/LAST.CYCLE).

The MISC/RESET.CPU function causes **E%STOP_IBOX_H** to be asserted in S5 and **E%FLUSH_MBOX_H**, **E_MSC%FLUSH_EBOX_H**, and **E%FLUSH_FBOX_H** to be asserted in S6.

8.5.22 Ebox IPRs

The Ebox implements two IPRs. They are IPRs 124-125 (decimal), PCSCR and ECR.

ECR is a possible source of **E_BUS%ABUS_L<31:0>**, accessed by specifying ECR in the A field of the microword. ECR and PCSCR are also possible destinations of **E_BUS%WBUS_L<31:0>**, written by specifying PCSCR or ECR in the DST field of the microword. On writes, the entire register is written, regardless of the current DL value.

Figure 8–9: IPR 7C (hex), PCSCR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0| 0| 0|              |  | 0| 0| 0| 0| 0| 0| 0| 0| 0| 0|  |  |  |  |  | 0| 0| 0| 0| 0| 0| 0| 0| :PCSCR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                 |        |                                |  |  |  |  |
                 |        +-- NONSTANDARD_PATCH            |  |  |  |  |
                 +-- PATCH_REV                             |  |  |  |  |
                                                    DATA --+  |  |  |  |
                                                  RWL_SHIFT --+  |  |  |
                                                     PCS_WRITE --+  |  |
                                                          PCS_ENB --+  |
                                                        PAR_PORT_DIS --+
```

8.5.22.1 IPR 7C (hex), Patchable Control Store Control Register

The PCSCR is used to load control store patches. Chapter 9 describes the patchable control store function in detail. Figure 8–9 and Table 8–21 show the bit fields and give descriptions.

Table 8–21: PCSCR Field Descriptions

Name	Extent	Type	Description
PAR_PORT_DIS	8	RW,0	Writing a 1 disables control by the testability parallel port of the section of the internal scan used in loading the control store CAM (content addressable memory) and RAM. This is necessary when using this register to load the control store CAM and RAM.
PCS_ENB	9	RW,0	Enables the control store CAM and RAM so that patches are fetched and supersede the control store ROM.
PCS_WRITE	10	WO	The event of writing a 1 to this bit causes the PCS scan chain contents to be written into the control store CAM and RAM. The control signal which enables the write returns to the inactive state automatically; there is no need for software to write a 0 to this bit after writing a 1. This bit always reads as 0.
RWL_SHIFT	11	WO	The event of writing a 1 to this bit causes the PCS scan chain to shift by one. The control signal which enables the shift returns to the inactive state automatically; there is no need for software to write a 0 to this bit after writing a 1. This bit always reads as 0.
DATA	12	RW,0	This bit holds the data which is shifted into the PCS scan chain when a 1 is written to RWL_SHIFT. By repeatedly setting DATA and writing a 1 to RWL_SHIFT, software can shift any data pattern into the PCS scan chain.
NONSTANDARD_PATCH[1]	23	RW	This bit is set by software after loading a microcode patch. If it is 1, it indicates a non-standard microcode patch has been loaded. This bit is returned as bit <8> in a read from the SID processor register, except that 0 is substituted for this bit in microcode for a SID read if PCSCR<PCS_ENB> is 0.
PATCH_REV[1]	28:24	RW	This field is set by software after loading a microcode patch. It indicates the revision of the standard microcode patch which has been loaded. This field is returned as bits <13:9> in a read from the SID processor register, except that 0 is substituted for this field in microcode for a SID read if PCSCR<PCS_ENB> is 0.

[1] This bit or field not implemented in pass 1 chips.

8.5.22.2 IPR 7D (hex), Ebox Control Register

The ECR is used to configure certain Ebox functions. Figure 8–10 and Table 8–22 show the bit fields and give descriptions.

Figure 8–10: IPR 7D (hex), ECR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  | 0| 0| 0| 0| 0| 0| 0| 0|  |        |     |  | 0| 0|  | 0| 0| 0| 0| 0|  |  |  |  |  |  |  |  | :ECR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
  |                          |     |      |    |        |                 |  |  |  |  |  |  |  |
  +-- PMF_CLEAR              |     |      |    |        |                 |  |  |  |  |  |  |  |
                  PMF_LFSR --+     |      |    |        |                 |  |  |  |  |  |  |  |
                        PMF_EMUX --+      |    |        |                 |  |  |  |  |  |  |  |
                               PMF_PMUX --+    |        |                 |  |  |  |  |  |  |  |
                                  PMF_ENABLE --+        |                 |  |  |  |  |  |  |  |
                                     FBOX_TEST_ENABLE --+                 |  |  |  |  |  |  |  |
                                                               ICCS_EXT --+  |  |  |  |  |  |  |
                                                             TIMEOUT_CLOCK --+  |  |  |  |  |  |
                                                                 TIMEOUT_TEST --+  |  |  |  |  |
                                                                TIMEOUT_OCCURRED --+  |  |  |  |
                                                             FBOX_ST4_BYPASS_ENABLE --+  |  |  |
                                                                           TIMEOUT_EXT --+  |  |
                                                                             FBOX_ENABLE  --+  |
                                                                              VECTOR_PRESENT --+
```

Table 8–22: ECR Field Descriptions

Name	Extent	Type	Description
VECTOR_PRESENT	0	RW,0	This bit is for vector unit support in a future version of this chip.
FBOX_ENABLE	1	RW,0	This bit is set to a 1 by configuration code to enable the Fbox.
TIMEOUT_EXT	2	RW,0	This bit is set to a 1 by configuration code to select an external timebase for the S3 stall timeout timer.
FBOX_ST4_BYPASS_ ENABLE	3	RW,0	This bit is set to a 1 by configuration code to enable Fbox Stage 4 bypass.
TIMEOUT_OCCURRED	4	WC	This bit indicates that an S3 stall timeout occurred. Writing it with a 1 clears it.
TIMEOUT_TEST	5	RW,0	If this bit is a 1, the S3 stall timeout circuit counts cycles instead of cycles in which E%TIMEOUT_ENABLE_H is asserted. In this test mode the S3 stall timeout time is roughly 50 microseconds instead of roughly 3 seconds.
TIMEOUT_CLOCK	6	RO	This bit is most significant bit of the timeout base counter. It is used as an indication that E%TIMEOUT_ENABLE_H is functioning (though some logic is not covered by this test). It should be 1 half of the time and 0 the other half of the time. The period of the oscillation is 65536 time the cycle time of the chip or of the waveform on **P%OSC_TC1_H**, depending on ECR<TIMEOUT_EXT>. For ECR<TIMEOUT_EXT> set to 0 and a 14 nsec cycle time, this is a period of roughly 900 microseconds.
ICCS_EXT	7	RW,0	This bit is set by configuration code to select the interval timer mode. When it is 0, the CPU implements a subset interval timer with ICCS<6> maintained on the chip. When set to 1, the CPU implements a full interval timer with ICCS, NICR, and ICR processor registers implemented off chip. See Chapter 10.
FBOX_TEST_ENABLE	13	RW,0	When this bit is set to a 1, E%FBOX_TEST_ENB_H is asserted. This puts the Fbox is a test mode in which data is passed from stage to stage unaltered.
PMF_ENABLE	16	RW,0	This bit is the internal implementation of the PME processor register. See Section 18.2.4 for more detail.
PMF_PMUX	18:17	RW,0	This field selects the source of the events counted by the performance monitoring facility, when enabled, to be Ibox, Ebox, Mbox, or Cbox. See Section 18.2.3 for more detail.
PMF_EMUX	21:19	RW,0	This field selects the Ebox events counted by the performance monitoring facility, when the performance monitoring facility is configured to count Ebox events. See Table 18–3 for more detail.
PMF_LFSR	22	RW,0	This bit enables the E%WBUS_H<31:0> LFSR (linear feedback shift register) accumulator. This is a testability feature. See Section 8.5.26.2 for more detail.

Table 8–22 (Cont.): ECR Field Descriptions

Name	Extent	Type	Description
PMF_CLEAR	31	WO	Writing a 1 to this bit clears the performance monitoring facility counters (which are also the E%WBUS_H<31:0> LFSR accumulator). It is not implemented in hardware. Microcode handles this function.

8.5.23 Initialization

The main mechanism for Ebox initialization is the power-up microtrap, and the MISC/RESET.CPU which occurs in the first microword of this microtrap flow. When this trap occurs, the Microsequencer will assert **E_USQ%PE_ABORT_L**, aborting the Ebox pipeline as it does for any microtrap. None of the registers in the register file or elsewhere in the Ebox are cleared on initialization, except that IPR bits are cleared where indicated by the bit type (see Section 8.5.22). The state flags are also cleared by reset.

The Ebox asserts **E%STOP_IBOX_H**, **E_MSC%FLUSH_EBOX_H**, **E%FLUSH_MBOX_H**, and **E%FLUSH_FBOX_H** during reset. This is the same effect as MISC/RESET.CPU. See the sections on initialization for each of the boxes for more detail.

8.5.24 Timing

TBS. A timing diagram for major Ebox signals will someday appear here.

8.5.25 Error Detection

Ebox handling of memory management faults and hardware errors detected by the Mbox while processing an Ebox or Ibox request is covered in Section 8.5.19 and Section 8.5.17.

8.5.25.1 S3 Stall Timeout

The Ebox implements an S3 stall timeout timer. The timeout time is shown in Table 8–23.

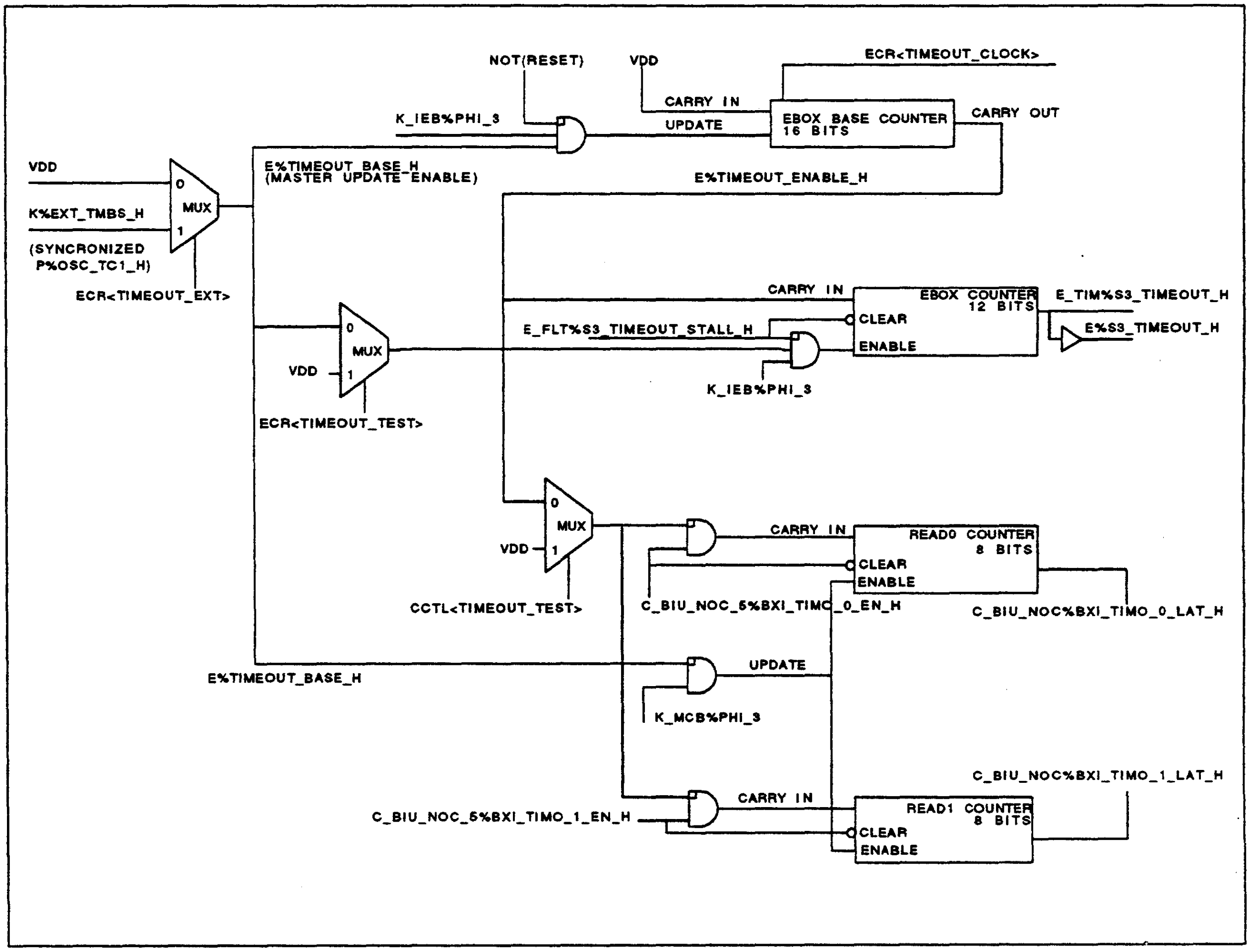
Figure 8–11 shows all the NVAX timeout timers, including those implemented in the Cbox. The Cbox timeout timers are shown because they use **E%TIMEOUT_BASE_H** as their timebase. See Section 13.4.3.4 for more detail on the Cbox timeout timers.

The timeout timer input is **E%TIMEOUT_BASE_H**, which is created internally by dividing the CPU clock by 65536. As an alterative in systems in which require longer timeout times than NVAX implements, this timer can use an externally supplied timebase. To select the external timebase, **K%EXT_TMBS_H**, ECR<TIMEOUT_EXT> is set to 1. In this case the base counter counts cycles of **K%EXT_TMBS_H** instead of the NVAX CPU internal clock. **K%EXT_TMBS_H** is a synchronized version of the signal received on pin **P%OSC_TC1_H**. Note that **P%OSC_TC1_H** is synchronized in the clock section to NDAL clocks and must therefore be driven with a clock signal which is high for longer than one NDAL cycle and low for longer than one NDAL cycle. For a square wave clock waveform this implies a speed of 11.9 MHz or less.

Table 8–23: S3 Stall Timeout Values in Normal Mode

Cycle time	Timeout Granularity	S3 Stall timeout
10-ns NVAX	655 microseconds	2.6837 (min) to 2.68345 (max) seconds
12-ns NVAX	786 microseconds	3.22044 (min) to 3.22123 (max) seconds
14-ns NVAX	917 microseconds	3.75718 (min) to 3.7581 (max) seconds

Figure 8–11: NVAX Timeout Counters



In every cycle the Ebox counter increments, if one of the following is true:

- S3 is stalled, or
- The microword in S3 is the STALL microword (as determined by the piped version of **E_USQ%IQ_STALL_H** sent from the Microsequencer).
- The field queue is being accessed via a microcode conditional branch and is empty (**E_FLQ%FQ_STALL_H** is asserted).

These conditions are accumulated into one condition, **E_FLT%S3_TIMEOUT_STALL_H**, in the fault logic section of the Ebox. If none of the above conditions is true, the Ebox counter is reset to 0. If the counter reaches its maximum value and overflows, an immediate asynchronous hardware error microtrap is forced. The microtrap breaks the Ebox stall by aborting the pipeline.

When the S3 stall timeout timer overflows, forcing a microtrap, the signal **E%S3_TIMEOUT_H** is asserted for one cycle. This causes the chip reset logic to reset the Mbox and Cbox. Microcode, in handling the asynchronous hardware error microtrap, must also do MISC/RESET.CPU in order to properly reset the Mbox.

The Ebox timeout counter treats cycles in which the pipeline advances the STALL microword into S3 as an S3 stall cycle. If the Microsequencer sends STALL microwords into the pipeline continuously, the timer will eventually timeout. This is the case when the instruction queue in the Microsequencer remains empty forever.

Similarly, if microcode is in an infinite loop, conditionally branching on the field queue contents, an S3 stall timeout will occur.

Any true S3 or S4 stall which lasts forever will cause an S3 stall timeout. It is expected that some hardware failures within the NVAX CPU could cause the Ebox to get out of sync with the Ibox, Ebox, or Fbox. This could result in the Ebox waiting forever for an event which will never happen. This timeout timer causes a machine check exception to occur instead of allowing the CPU to simply hang.

8.5.25.1.1 Testing the S3 Stall Timeout Timer

The Ebox timeout counter may be configured for testing by writing a 1 to ECR<S3_TIMEOUT_TEST>. When this bit is 1, the Ebox counter counts NVAX CPU internal clock cycles instead of cycles of **E%TIMEOUT_BASE_H**. Table 8–24 gives the timeout times in test mode. See the timeout counter test discussion in Section 13.4.3.4 for detail on how to cause a timeout for test purposes. The timeout will cause the asynchronous hardware error machine check (see Chapter 15).

Table 8–24: S3 Stall Timeout Values in Test Mode

Cycle time	Timeout Granularity	S3 Stall timeout
10-ns NVAX	10 nanoseconds	40.95 (min) to 40.96 (max) microseconds
12-ns NVAX	12 nanoseconds	49.14 (min) to 49.152 (max) microseconds
14-ns NVAX	14 nanoseconds	57.33 (min) to 57.344 (max) microseconds

DERIVATION OF TIMEOUT VALUES

The timeout values given above were derived as follows:

Table 8–25: Derivation of NVAX Timeout Values

NVAX mode	Timeout Granularity (in NVAX cycles)	S3 Stall timeout (in NVAX cycles)
Normal	2**16	2**28—2**16 (min) to 2**28 (max)
Test	1	2**12—1 (min) to 2**12 (max)

8.5.26 Testability

This section describes the testability features in the Ebox.

8.5.26.1 Parallel Port Test Features

The microaddress currently being used to access the control store is visible on the parallel port. Much information about Ebox execution can be inferred from the sequence of microaddresses seen on the parallel port. See Section 9.5.

No other Ebox signal is visible directly at the parallel port. Quite a few are visible through the internal observability scan chain controlled via the parallel port controlled inputs. Table 8–26 shows these signals. Timing information and a description is given for each signal.

The scan chain loads input data in Φ_4. If a signal is not ready to be latched in Φ_4, it has to be delayed before being loaded into the scan chain. This implies that the particular signal's value sampled by the scan chain is from one cycle earlier than the cycle in which the scan chain was loaded. This is shown Table 8–26 in the timing column.

Table 8–26 lists the scan chain data bits in the order in they would appear at the parallel port. The value of **E_RGF%ERROR_H** appears first and the value of **F%STORE_STALL_H** appears last.

Table 8–26: Ebox Observe Scan Signals

Schematic Signal	Timing	Description
E_RGF%ERROR_L	delayed	A 1 value means the Ebox is detecting a hardware error associated with the current MD read (including bypassed MD reads) or with a current S3 Ibox-to-Ebox queue access (instruction queue, source queue, or field queue).
E_RGF%FAULT_L	delayed	A 1 value means the Ebox is detecting a memory management fault associated with the current MD read (including bypassed MD reads) or with a current S3 Ibox-to-Ebox queue access (instruction queue, source queue, or field queue).
I%IMEM_MEXC_H		A 0 value means the Ibox is signaling a memory management exception associated with one of the Ibox-to-Ebox queues (instruction queue, source queue, field queue, branch queue, or destination queue).

Table 8–26 (Cont.): Ebox Observe Scan Signals

Schematic Signal	Timing	Description
F%INPUT_STALL_H		A 0 value means the Fbox is currently requesting that no more input data be sent.
I%IMEM_HERR_H		A 0 value means the Ibox is signaling a hardware error associated with one of the Ibox-to-Ebox queues (instruction queue, source queue, field queue, branch queue, or destination queue).
E_SDQ%DQ_STALL_L		A 1 value means the destination queue is being accessed and there isn't a valid entry.
E_MEM%F_STORE_H		A 0 value means the Fbox is requesting a store in this cycle.
E_RGF%IW_BYPASS_B_L	delayed	A 1 value means the Ibox register file write is being bypassed to E_BUS%BBUS_L.
E_RGF%BDATA_VALID_L	delayed	A 1 value means the data on E_BUS%BBUS_L is valid (otherwise a MD or WN stall would occur).
E_RGF%IW_BYPASS_A_L	delayed	A 1 value means the Ibox register file write is being bypassed to E_BUS%ABUS_L.
E_RGF%ADATA_VALID_L	delayed	A 1 value means the data on E_BUS%ABUS_L is valid (otherwise a MD or WN stall would occur).
E_SDQ%SCOREBOARD_HIT_STALL_H		A 0 value means the Fbox destination scoreboard in the destination queue has a hit (i.e., a current source queue based register file read is to a register the Fbox will update in the future).
E_SDQ%SQ_STALL_H		A 0 value means the current source queue read(s) is (are) accessing an empty location - one kind of S3 stall.
E_RTQ%RQ_SEL_EBOX_STL_L		A 1 value means the Ebox is next to retire an instruction, not the Fbox.
E_FLT%MME_ERR_H	delayed	A 0 value means the Ebox is signaling the microsequencer to initiate a memory management fault microtrap.
E_FLT%HW_ERR_H	delayed	A 0 value means the Ebox is signaling the microsequencer to initiate a synchronous hardware error microtrap.
E_SDQ%S4_FDEST_STALL_L		A 1 value means the Ebox is stalled in S4 doing the FDEST.CHECK operation and the destination queue doesn't contain the necessary entry or entries.
VSS		Always a 1 value.
E_RGF%WNGPR_ERROR_H		A 0 value means the Ebox is recognizing a hardware error because the Mbox wrote a working register or GPR while asserting M%HARD_ERR_H.
E_FLT%Q_FAULT_H		A 0 value means the Ebox is detecting a memory management fault with a current S3 Ibox-to-Ebox queue access (instruction queue, source queue, or field queue).

Table 8–26 (Cont.): Ebox Observe Scan Signals

Schematic Signal	Timing	Description
E_FLT%Q_ERROR_H		A 0 value means the Ebox is detecting a hardware error with a current S3 Ibox-to-Ebox queue access (instruction queue, source queue, or field queue).
F%CC_MAP_H<1>		A 0 value means the most significant bit of this field is a 1. See Table 8–6. This data is valid for the condition code alteration in the current cycle (S5), provided it is a Fbox instruction being retired.
F%CC_MAP_H<0>		A 0 value means the least significant bit of this field is a 1. See Table 8–6. This data is valid for the condition code alteration in the current cycle (S5), provided it is a Fbox instruction being retired.
F%RETIRE_H		A 0 value means the Fbox is requesting an instruction retire in this cycle.
E_STL%LAT_PAQ_ _STL_H		A 0 value means the Ebox is stalling because the PA queue is not valid and the current destination queue access is requiring the use of the PA queue.
E_STL%BQ_STALL_H		A 0 value means the Ebox is stalling because the branch queue is empty and the current microinstruction in S4 accesses it.
F%MERR_H		A 0 value means the Fbox is signaling a hardware error on one of the source operands for the currently retiring instruction.
F%RSVD_ADDR_MODE_H		A 0 value means the Fbox is signaling a reserved address mode fault on one of the source operands for the currently retiring instruction.
F%MMGT_FAULT_H		A 0 value means the Fbox is signaling a memory management fault on one of the source operands for the currently retiring instruction.
F%RSV_H		A 0 value means the Fbox is signaling a reserved operand fault for the currently retiring instruction.
F%FOV_H		A 0 value means the Fbox is signaling a floating overflow fault for the currently retiring instruction.
F%FDBZ_H		A 0 value means the Fbox is signaling a floating divide-by-zero fault for the currently retiring instruction.
F%FU_H		A 0 value means the Fbox is signaling a floating underflow fault for the currently retiring instruction.
M%EM_LAT_FULL_H	delayed	A 0 value means the Mbox is signaling that the EM_LATCH is full.
E%EREF_REQ_H		A 0 value means the Ebox is making an Mbox request in this current cycle.
VSS		Always a 1 value.
E%START_IBOX_IO_RD_H		A 0 value means the Ebox is signaling the Mbox that an Ibox IO space read may begin in the current cycle, subject to certain Mbox restrictions.

Table 8–26 (Cont.): Ebox Observe Scan Signals

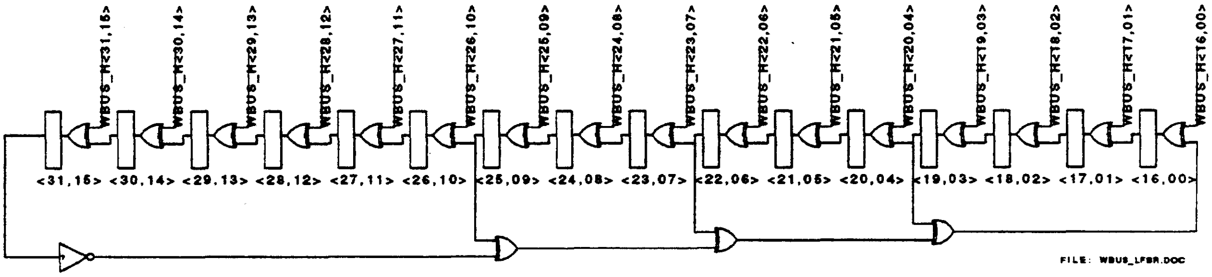
Schematic Signal	Timing	Description
F%STORE_STALL_H		A 0 value means the Fbox aborted a store request late in the current cycle.

8.5.26.2 E%WBUS_H<31:0> LFSR

E%WBUS_H<31:0> (the buffered copy of E_BUS%WBUS_L<31:0> which is driven to the Mbox) has an LFSR (linear feedback shift register) accumulator. The LFSR is implemented as part of the performance monitoring facility that is described in Chapter 18, and controlled by two bits in the ECR processor register: PMF_LFSR and PMF_CLEAR.

The E%WBUS_H<31:0> LFSR is implemented as two identical 16-bit LFSRs, one for E%WBUS_H<31:16> and one for E%WBUS_H<15:0>. A block diagram of one of these 16-bit LFSRs is shown in Figure 8–12. The reader should note that the output of the left-most bit in the LFSR chain is inverted before being XORed with earlier taps. This was done for implementation reasons.

Figure 8–12: E%WBUS_H LFSR Block Diagram



Both halves of the E%WBUS_H<31:0> LFSR may be cleared by software by writing a 1 to ECR<PMF_CLEAR> (which results in microcode executing the MISC/CLR.PERF.COUNT function). The operation of the pair of LFSRs is started by software by writing a 1 to ECR<PMF_LFSR> and stopped by writing a 0 to the same bit. The current state of the E%WBUS_H<31:0> LFSR may be read by software via the PMFCNT processor register (an E_BUS%ABUS_L<31:0> source available via MFPR) in the format shown in Figure 8–13.

Figure 8–13: PMFCNT Processor Register in E%WBUS_H<31:0> LFSR Format

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|           E_WBUS<31:16> LFSR Value            |           E_WBUS<15:00> LFSR Value            | :PMFCNT
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

CAUTION

The **E%WBUS_H<31:0>** LFSR hardware also provides the performance monitoring facility function under control of ECR<PMF_ENABLE>. The operation of the hardware is UNDEFINED if both ECR<PMF_ENABLE> and ECR<PMF_LFSR> are on, or if software uses a single MTPR write to turn off one bit and turn on the other simultaneously. That is, if either bit is on, software must turn off both bits with one MTPR and turn on the other with a second MTPR.

8.5.27 Microcode Restrictions

This section gives microcode restrictions due to Ebox microarchitecture and the VAX architecture.

8.5.27.1 Register Access Restriction

The first microword of any execution microflow must not read GPRs explicitly, and an explicit read must be preceded by at least one microword specifying something other than NONE in the DST field. (A/S1, A/S2, B/S1, and B/S2 are always allowed.) This restriction has to do with the fact that the Fbox destination scoreboard only examines the source queue outputs to detect GPR read-before-write hazards. Therefore it specifically does not apply in a microtrap flow since the Fbox can never write a result after a microtrap.

8.5.27.2 FLUSH.PAQ Restriction

MISC2/FLUSH.PAQ should only be specified when the MRQ field specifies an Mbox operation which is sent in the EM latch (i.e., other than MRQ/SYNC.BDISP, MRQ/SYNC.BDISP.RETIRE, MRQ/SYNC.BDISP.TEST.PRED, or MRQ/NOP). Otherwise the Mbox will not flush the PA queue.

8.5.27.3 Memory access restrictions

Microcode must ensure that all accesses from the current microflow are complete before allowing the microsequencer to dispatch to the next microflow.

Destination queue indirect writes (DST/DST) may be implicit memory operations. The MRQ field must specify NOP,SYNC.BDISP,SYNC.BDISP.RETIRE, or SYNC.BDISP.TEST.PRED when this operation is specified.

8.5.27.4 Shifter Restrictions

If the shifter uses the SC register as the source of the shift amount, the SC must have been loaded from **E_BUS%ABUS_L<4:0>** by the previous microword or from **E_BUS%WBUS_L<4:0>** by the microword before that. Otherwise the old SC value is used as the shift amount.

8.5.27.5 SHIFT.SIGN Restriction

The saved copy of the shifter sign bit (Saved-SHF<N>) is UNPREDICTABLE after executing a special format microword.

8.5.27.6 MMGT.MODE Restrictions

The MMGT.MODE register must be loaded before (in a microword preceding) the microword specifying a memory management probe in the MRQ field.

8.5.27.7 MPU Restrictions

If the MPU mask value is loaded by microword N specifying MISC/LOAD.MPU.FROM.B, microcode may not branch on the new value until microword N+2. Microcode may branch on the old value in N and N+1.

8.5.27.8 Microbranch Condition Restrictions

The first microword of a macroinstruction execution microflow should not branch based on the state flags. (It may set or clear them.)

8.5.27.9 Ibox IPR read restriction

Microcode should not use GPRs as the target for read type accesses to Ibox IPRs. There is no synchronization mechanism to determine when the result is ready. Also, the control logic in the Ibox IPR assumes a working register is the destination.

8.5.27.10 RETIRE.INSTRUCTION

The MISC1 field operation, RETIRE.INSTRUCTION must always be followed by a MISC/RESET.CPU. The MISC/RESET.CPU may come any number of cycles later, but must come before the next macroinstruction microflow is dispatched.

8.5.27.11 VAX Restart Bit Restriction

The VAX Restart Bit should not be read until two microwords after the last microword whose effect is expected to be reflected in the bit's state. For example, the machine check microflow should wait until the second microword before reading the bit to put it on the stack. Then the bit will reflect the state for the aborted execution microflow.

8.5.27.12 Q Register Interaction With SMUL.STEP and UDIV.STEP

In the microword after the last ALU/SMUL.STEP or ALU/UDIV.STEP, the Q register should not be sourced to **E_BUS%ABUS_L<31:0>** or **E_BUS%BBUS_L<31:0>**. Bypassing is not implemented for this kind of Q register update.

The microword before an ALU/SMUL.STEP must not update the Q register (Q/UPDATE.Q) unless that microword also specifies ALU/SMUL.STEP.

8.5.27.13 UDIV/SMUL Restrictions

The Q field must specify Q/UPDATE.Q if the ALU field functions SMUL.STEP or UDIV.STEP are specified. Also the ALU result must be specified as the source of **E_BUS%WBUS_L<31:0>**, and the shifter operation must be NOP.

8.5.27.14 F.DEST.CHECK Restrictions

The F.DEST.CHECK miscellaneous operation should only be used as intended. It should be specified in the last microword of a microflow which sends operands to the Fbox. It should never be specified in a microword which also specifies DST in the DST field.

8.5.27.15 Fbox Operand Delivery Restriction

IN delivering operands to the Fbox microcode may only not use A/S2 or B/S1. Short literal bypass to the Fbox source operand buses is not implemented for these decodes. Use of these decodes for Fbox operands could cause improper input data formatting in the Fbox if a short literal data item is present in the source queue.

8.5.27.16 RMUX control Restrictions

Every microword with an S4 or S5 side effect of modifying any state (examples include SYNC.BDISP.RETIRE, RESET.CPU, and LOAD.PSL.CC.XXXX) must specify a DST other than NONE. A DST of WBUS is acceptable. This restriction specifically does not apply to FDEST.CHECK.

Every microword specifying any operation other than NOP in the MRQ field must specify a DST other than NONE. A DST of WBUS is acceptable.

8.5.27.17 Control Bits

After changing either of ECR<1 or 3> (FBOX_ENABLE or FBOX_ST4_BYPASS_ENABLE) microcode should not do a SEQ.MUX/LAST.CYCLE or SEQ.MUX/LAST.CYCLE.OVERFLOW in the three microwords following the one altering the control bit.

8.5.27.18 Microtrap Dispatch and RESET.CPU Restrictions

8.5.27.18.1 Microtrap Flows

In a microtrap handler for any microtrap except branch mispredict, microcode must do a MISC/RESET.CPU before it can read any of the registers in the register file which has a valid bit. This restriction is necessary to avoid deadlock. Specifically, microcode must not source any Wn register (working register) until the microword after the one which specifies MISC/RESET.CPU.

In a microtrap handler for any microtrap except branch mispredict, there should be no memory request until the third microword after the one specifying MISC/RESET.CPU.

In a microtrap handler for any microtrap except branch mispredict, any microcode operation which causes an entry in the retire queue to be retired is illegal until a MISC/RESET.CPU is executed and a second microword specifying SEQ.MUX/LAST.CYCLE and DISABLE.RETIRE/YES is executed. This second microword must not occur until after the third microword after the one specifying MISC/RESET.CPU.

In a microtrap handler for any microtrap except branch mispredict, UNPREDICTABLE or UNDEFINED results could occur if microcode accesses the source queue, destination queue, instruction queue, branch queue, or field queue until the fourth microword after the one which specifies MISC/RESET.CPU. Similarly, UNPREDICTABLE results could occur if microcode reads from the Wn or GPR register before the fourth microword after the one which specifies MISC/RESET.CPU, or writes to these registers before the second microword after the one which specifies MISC/RESET.CPU.

8.5.27.18.2 MISC/RESET.CPU Restrictions

The fourth microword after one specifying MISC/RESET.CPU may specify SEQ.MUX/LAST.CYCLE (with DISABLE.RETIRE/YES), but the first three must not. The first three microwords after a MISC/RESET.CPU must not access the source queue or field queue. The first two microwords after a MISC/RESET.CPU must not access the destination queue, branch queue.

The first two microwords after a MISC/RESET.CPU must not issue memory requests.

After a microword specifying MISC/RESET.CPU, any microcode operation which causes an entry in the retire queue to be retired is illegal until a microword specifying SEQ.MUX/LAST.CYCLE and DISABLE.RETIRE/YES is executed. This microword must not occur until after the third microword after the one specifying MISC/RESET.CPU.

8.5.27.18.3 Asynchronous Hardware Error Microtrap Restriction

There are two possible causes of this microtrap, TB parity error and S3 stall timeout. If the cause is S3 stall timeout then the Mbox and Cbox are reset by Ebox hardware for 17.5 cycles. Microcode must not issue any memory requests during that reset time period. Also, the Mbox requires that the MISC/RESET.CPU function be done during the reset period. The first microword of the microtrap handler does not reach S6 until 5 cycles after the S3 stall timeout is detected. Hence the earliest the effect of MISC/RESET.CPU on the Mbox can occur is 5 cycles into the 17.5 cycle reset period. Microcode currently issues the MISC/RESET.CPU upon entry to the asynchronous hardware error microtrap (regardless of the cause) and then waits 23 cycles before beginning normal exception handling procedures. This is the recommended procedure.

8.5.27.18.4 First Part Done Dispatch Restriction

The microcode flow at the dispatch for PSL<FPD> set must determine if the opcode is that of an Fbox instruction. If it is, then a MISC/RESET.CPU must occur before the next SEQ.MUX/LAST.CYCLE or SEQ.MUX/LAST.CYCLE.OVERFLOW. This case results in the Fbox and Ebox being out of synch in the protocol for sending opcodes and operands. The Fbox must be flushed. If the instruction is not an Fbox instruction, microcode may continue without the MISC/RESET.CPU (as it does in the case of unpacking and continuing the execution of an interrupted string instruction such as MOVC3).

8.5.27.19 PSL Use Restrictions

The PSL must not be loaded in the first microword of a macroinstruction execution microflow.

The first two microwords of any macroinstruction execution microflow (any opcode dispatch or the FPD dispatch) should not use the PSL as a source. The PSL<TP> bit read onto E_BUS%ABUS_L<31:0> will not necessarily be correct. Microcode may disregard this restriction if it is acceptable for this bit to be incorrect. (Reading the PSL does not prevent the automatic copy of <T> to <TP>.)

The PSL should not be read in the microword after it is updated. If this rule were not followed, it is UNPREDICTABLE whether the second microword will source the old or the new PSL value. (Actually it depends on whether an S3 stall occurs on the second microword.)

On loading a new PSL, the third microword after the one altering the PSL may specify LAST.CYCLE for a decode dispatch, but the first two may not. If it is known that the PSL<FPD, T, or TP> bits will not change, then this restriction does not apply.

On loading a new value to PSL<FU>, the microword after the one altering the PSL may specify SEQ.MUX/LAST.CYCLE for a decode dispatch, but the one which altered the PSL must not.

If microcode loads a new value to PSL<IPL> in microword N, then microwords N through N+3 must not specify SEQ.MUX/LAST.CYCLE or SEQ.MUX/LAST.CYCLE.OVERFLOW, but N+4 may.

After changing the PSL microcode generally should not micro-branch on PSL bits in the next two microwords. Assuming microword N updates the PSL, if microwords N or N+1 branch on the PSL the old PSL value will determine the result of the microbranch. However, if microword N+2 branches on the PSL, it is UNPREDICTABLE whether the old or new PSL bits will be used to determine the branch outcome. (Actually, it is predictable if S3 stalls on microword N+1 are known.) If N+3 branches on the PSL, the new PSL value will definitely determine the result of the microbranch. This restriction specifically does not apply if PSL<29,26:22> are not changed by the load.

Many microcode flows alter the condition code bits, PSL<3:0>, in the last cycle of the flow. This implies that microcode should not source the PSL in the first microwords of any flow except microtrap flows (i.e., don't in these flows: opcode dispatch, FPD dispatch, trace fault dispatch, or interrupt dispatch) unless it is acceptable that the incorrect value might be read for the condition code bits. (This assumes that the first microword of the flow synchronizes to any outstanding Fbox retire by specifying a DST other than NONE.)

Certain restrictions accompany changes to PSL<CUR_MOD>. The Mbox must not be processing any Ebox references or operand prefetches while PSL<CUR_MOD> is being changed. The microword after the one changing PSL<CUR_MOD> can issue a memory reference which will be access checked using the new PSL<CUR_MOD> value.

There are no restrictions on reading or writing the PSL in beginning of a microtrap flow. The Ebox pipeline has been flushed before the microtrap flow begins, so there can't be updates to the PSL after this microflow starts.

The following table summarizes PSL restrictions at beginnings and ends of flows.

Table 8–27: PSL Restrictions Summary

PSL Bits	At beginning of new flow[1]	Before end of any flow[2]
PSL<3:0>; PSL<N,Z,V,C>[3]	1[4]	0
PSL<4,27>; PSL<T,FPD>	0	3
PSL<6>; PSL<FU>	0	1
PSL<20:16>; PSL<IPL>	0	4
PSL<30>; PSL<TP>	2	3
PSL<any other>	1	0

[1] Number of microwords required at beginning of microflow before microword in which these bits are read. Applies to macroinstruction execution flows (including FPD dispatches), and to trace fault and interrupt dispatches, but not to microtrap dispatches.

[2] Number of microwords after one which alters these bits (before and including the one which specifies SEQ.MUX/LAST.CYCLE or SEQ.MUX/LAST.CYCLE.OVERFLOW)

[3] This assumes the microcode convention of altering the PSL condition code bits in the last microword of some execution flows.

[4] This assumes that the first microword of the flow synchronizes to any outstanding Fbox retire.

8.5.27.20 S+PSW Restrictions

The PSL is written in S5 while the S+PSW source is read in S3. If microword N updates the PSL, microword N+1 should not source S+PSW. It is UNPREDICTABLE whether the old or new value would be sourced if this restriction were not obeyed.

8.5.27.21 RN.MODE.OPCODE Restrictions

For the RN field to be valid, the A field of the microword must specify S1 (the current source queue entry), and the microcode must know from context that the source queue entry points to a GPR. If these restrictions are not met, the value returned in the RN field is UNPREDICTABLE.

The PSL is written in S5 while the RN.MODE.OPCODE source is read in S3. If microword N updates the PSL, microword N+1 must not source the new value of RN.MODE.OPCODE. It could receive the old value.

8.5.28 Signal Name Cross-Reference

The following table gives a cross reference for selected signal names in this chapter. Only signal names which have different names in this chapter than they do on the schematics are listed. Different names are used in this chapter only where the resulting description is significantly clearer.

Table 8–28: Signal Name Cross-Reference

Name in this chapter	Name on schematics	Name in behavioral model
E_ALU%RESULT_H<31:15>	E_ALU_AD2%R_L<31:15>	E_ASH_ALU%RESULT_H<31:15>
E_ALU%RESULT_H<14:0>	E_ALU_AD1%R_L<14:0>	E_ASH_ALU%RESULT_H<14:0>
E_ALU%CI_H<32>	E_ALU_AD2%CI_L<32>	E_ASH_ALU%CARRIES_OUT_H<31>
E_ALU%CI_H<31,29,27,25, 23,21,19,17,15>	E_ALU_AD2%CI_L<31,29,27,25, 23,21,19,17,15>	E_ASH_ALU%CARRIES_OUT_H<30,28,26,24, 22,20,18,16,14>
E_ALU%CI_H<30,28,26,24, 22,20,18,16>	E_ALU_AD2%CI_H<30,28,26,24, 22,20,18,16>	E_ASH_ALU%CARRIES_OUT_H<29,27,25, 23,21,19,17,15>
E_ALU%CI_H<14,12,10,8,6,4,2>	E_ALU_AD1%CI_H<14,12,10,8,6,4,2>	E_ASH_ALU%CARRIES_OUT_H<13,11,9,7,5,3,1>
E_ALU%CI_H<13,11,9,7,5,3,1>	E_ALU_AD1%CI_L<13,11,9,7,5,3,1>	E_ASH_ALU%CARRIES_OUT_H<12,10,8,6,4,2>
E_ALU%CI_H<0>	E_ALU_AC2%CIN_H	E_ASH_ALU%C_IN_H
E%RMUX_S4_FLUSH	no exact match, roughly equals the following: E_STL%VERY_LATE_NOP_RMUX_S4_L	no exact match, roughly equals the following: E_STL%LATE_F_NOP_RMUX_S4_H, E_STL%VERY_LATE_NOP_RMUX_S4_H
E%RMUX_S4_STALL	no exact match, roughly equals the following: E_STL%STALL_RMUX_S4_L, E_STL%LATE_STALL_RMUX_S4_L, E_STL%VERY_LATE_STALL_RMUX_S4_L	no exact match, roughly equals the following: E_STL%STALL_RMUX_S4_H, E_STL%LATE_STALL_RMUX_S4_H, E_STL%VERY_LATE_STALL_RMUX_S4_H
E%RMUX_S5_FLUSH	no exact match, roughly equals the following: E_STL%NOP_RMUX_S5_L, E_STL%F_NOP_RMUX_S5_H	no exact match, roughly equals the following: E_STL%F_NOP_RMUX_S5_H, E_STL%LATE_F_NOP_RMUX_S5_H, E_STL%NOP_RMUX_S5_H
E%S3_FLUSH	no exact match, roughly equals the following: E_USQ%PE_ABORT_L, E_STF%PE_ABORT_H, E_STL%%PE_ABORT_H	no exact match, roughly equals the following: E_USQ%PE_ABORT_H
E%S3_STALL	no exact match, roughly equals the following: E_STL%STALL_S3_L, E_STL%LATE_STALL_S3_L, E_STL%VERY_LATE_STALL_S3_L	no exact match, roughly equals the following: E_STL%STALL_S3_H, E_STL%LATE_STALL_S3_H, E_STL%VERY_LATE_STALL_S3_H
E%S4_STALL	no exact match, roughly equals the following: E_STL%STALL_S4_L, E_STL%LATE_STALL_S4_L	no exact match, roughly equals the following: E_STL%STALL_S4_H, E_STL%LATE_STALL_S4_H

Table 8–28 (Cont.): Signal Name Cross-Reference

Name in this chapter	Name on schematics	Name in behavioral model
E%S4_FLUSH	no exact match, roughly equals the following: E_STL%F_NOP_S4_H, E_STL%LATE_F_NOP_S4_H	no exact match, roughly equals the following: E_STL%F_NOP_S4_H, E_STL%LATE_F_NOP_S4_H
E%S5_FLUSH	no exact match, roughly equals the following: E_STL%NOP_S5_L, E_STL%F_NOP_S5_H	no exact match, roughly equals the following: E_STL%NOP_S5_H, E_STL%F_NOP_S5_H

8.5.29 Revision History

Table 8–29: Revision History

Who	When	Description of change
John Edmondson	30-NOV-1988	Initial Release.
John Edmondson	19-DEC-1988	Corrections and Updates.
John Edmondson	06-MAR-1989	Release for external review.
John Edmondson	29-NOV-1989	Updates after external review and modeling complete.
John Edmondson	18-DEC-1989	Further updates, particularly adding real signal names.
John Edmondson	31-JAN-1990	Updates reflecting minor implementation motivated changes - rev 0.5.
John Edmondson	4-MAY-1990	Updates reflecting minor implementation motivated changes - post rev 0.5.
John Edmondson	20-FEB-1991	Further updates post implementation.
John Edmondson	31-MAY-1991	Minor updates for pass 2 changes.

Chapter 9

The Microsequencer

9.1 Overview

The microsequencer is a microprogrammed finite state machine that controls the three Ebox sections of the NVAX pipeline: S3, S4, and S5. The microsequencer itself resides in the S2 section of the pipeline. It accesses microcode contained in an on-chip control ROM, and microcode patches contained in an on-chip SRAM. Each microword is made up of fields that control all three pipeline stages. A complete microword is issued to S3 each cycle, and the appropriate microword decodes are pipelined forward to S4 and S5 under Ebox control.

Each microword contains a microsequencer control field that specifies the next microinstruction in the microflow. This field may specify an explicit address contained in the microword or direct the microsequencer to accept an address from another source. It also allows the microcode to conditionally branch on various NVAX states.

Frequently used microcode can be made into microsubroutines. When a microsubroutine is called, the return address is pushed onto the microstack. Up to six levels of subroutine nesting are possible.

Stalls, which are transparent to the microcoder, occur when an NVAX resource is unavailable, such as when the ALU requires an operand that has not yet been provided by the Mbox. The microsequencer stalls when S3 of the Ebox is stalled.

Microtraps allow the microcoder to deal with abnormal events that require immediate service. For example, a microtrap is requested on a branch mispredict, when the Ebox branch calculation is different from that predicted by the Ibox for a conditional branch instruction. When a microtrap occurs, the microcode control is transferred to a service microroutine.

9.2 Functional Description

9.2.1 Introduction

The NVAX microsequencer consists of several functional units of logic that are explained in the following sections and illustrated in the block diagram, Figure 9–1.

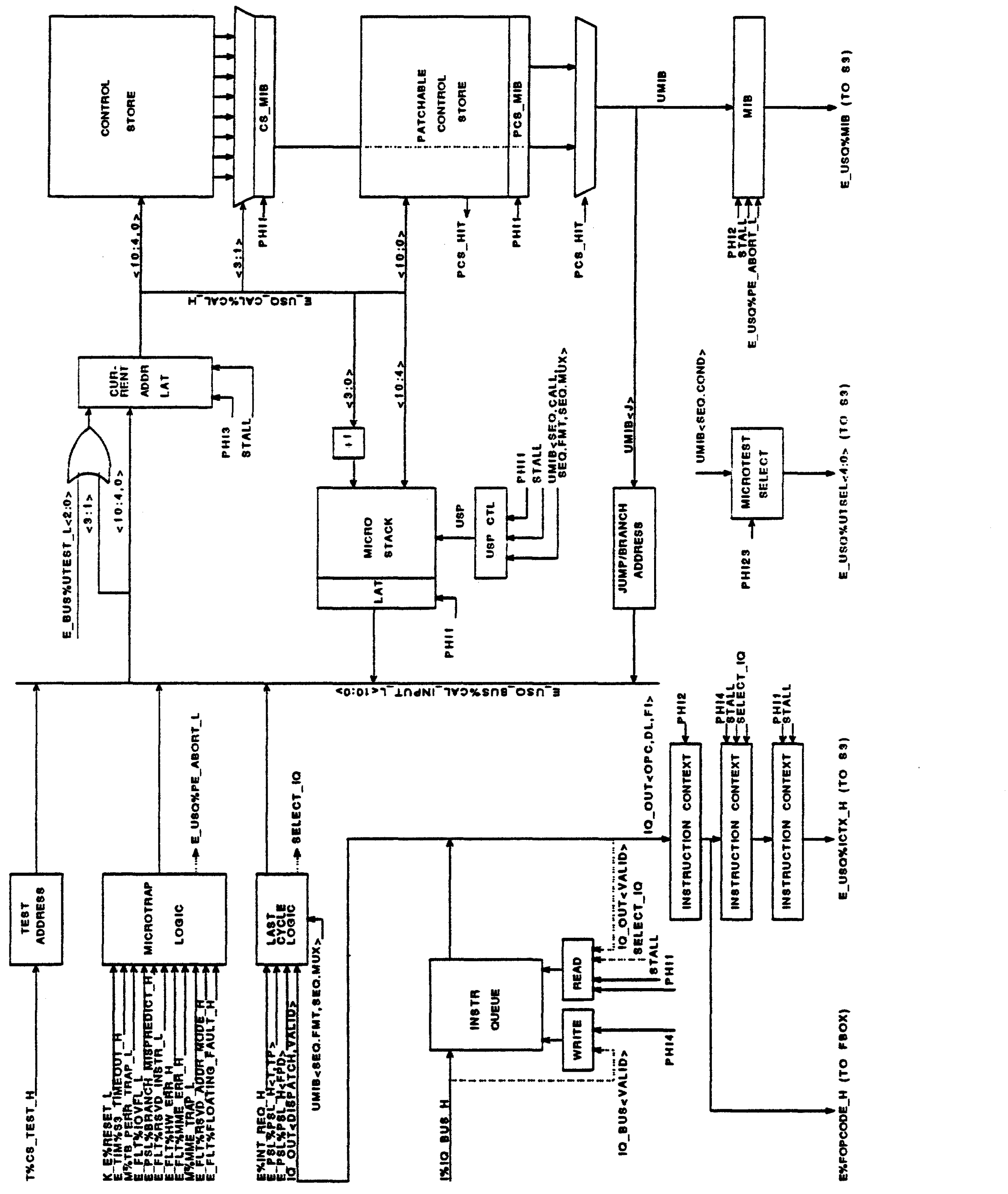


Figure 9–1: Microsequencer Block Diagram

9.2.2 Control Store

The control store is an on-chip ROM which contains the microcode used to execute macroinstructions and microtraps. It is made up of up to 1600 microwords. These are arranged as 200 entries, each entry consisting of 8 microwords. Each microword is 61 bits long, with bits <14:0> being used to control the microsequencer. The remainder of the microword, bits <60:15>, is used by the Ebox to control S3 through S5. The Ebox also receives bits <14,12:11>, enabling it to recognize the last cycle of a microflow and the validity of the microtest bus select lines.

The control store access is performed during Φ_{34} of S2 and Φ_1 of S3 of the NVAX pipeline. The output of the Current Address Latch (CAL), E_USQ_CAL%CAL_H<10:0>, is used to address the control store. Bits <10:4,0> are used to select one of the 200 entries. The eight microwords in the selected entry then enter an eight-way multiplexer, where E_USQ_CAL%CAL_H<3:1> select the final control store output. This structure is used because E_USQ_CAL%CAL_H<3:1> are valid later than bits <10:4,0>, since E_USQ_CAL%CAL_H<3:1> must be OR'd with the microtest bus for a BRANCH format microinstruction (see Section 9.2.2.2.2 for details).

9.2.2.1 Patchable Control Store

The patchable control store is an on-chip SRAM which contains microcode patches. It consists of up to 20 microwords. It operates in parallel with the control store. The microaddress from the CAL is the input to its CAM (Content Addressable Memory). If the address hits in the CAM, the output of the patchable control store is selected as the new microword, rather than the output of the ROM control store.

The patchable control store and CAM are precharged in Φ_3 and evaluate in Φ_{41}. The CAL output, E_USQ_CAL%CAL_H<10:0>, is used in its entirety as the lookup address in the CAM, as opposed to the 1-of-200 selection followed by the 1-of-8 selection used in the ROM control store.

9.2.2.1.1 Loading the Patchable Control Store

Entries in the Patchable Control Store and its CAM are written under software control from the Patchable Control Store Control Register (PCSCR) in the Ebox. The CAM must be disabled during this operation, so that no hits can occur. This is done by writing a zero to PCSCR<PCS_ENB>. In addition, Parallel Test Port control of the MIB scan chain must be disabled, by writing a one to PCSCR<PAR_PORT_DIS>. Following assertion of K_E%RESET_L, PCSCR<PCS_ENB> and PCSCR<PAR_PORT_DIS> both contain zeroes.

Data is serially scanned into the MIB scan chain, in the order shown in Table 9–2 (data is shifted from bit 0 to bit 91). The data is taken from PCSCR<DATA>; shifting into the scan chain is enabled by PCSCR<RWL_SHIFT>.

The final 20 bits scanned in (positions<19:0> in the scan chain) are used to select which entry in the patchable control store is to be written. Only one of these 20 bits may be asserted at a time. When all 92 bits of the scan chain have been serially loaded, the selected patchable control store and CAM entry are written under control of PCSCR<PCS_WRITE>.

All patchable control store entries must be written with either valid or NULL patches before the PCS is enabled. A NULL patch is an entry whose CAM location is written with an unused/unreferenced microaddress; there can never be a hit on this microaddress. The values of the MIB bits in a NULL patch are don't-care.

When the patchable control store is loaded, the patch revision must be loaded into PCSCR<PATCH_REV>. If the patch is non-standard (i.e., one which is not a formally distributed patch, such as a performance analysis patch), PCSCR<NONSTANDARD_PATCH> must be set to 1; otherwise it must be set to 0. These fields can be read by software to determine which patches are present in the machine. These fields are included in reads of the SID processor register.

Enabling of the patchable control store is done by writing a zero to PCSCR<PAR_PORT_DIS> and then writing a one to PCSCR<PCS_ENB>.

See Section 8.5.22.1 for more details on PCSCR operation.

The following table shows an example of writing an entry in the patchable control store.

Table 9–1: Example: Writing an Entry in the Patchable Control Store

Phase	Action
Microcycle 1	
1	
2	
3	Write 0 to PCSCR<PCS_ENB>[1] (disable the CAM) CAM NOW DISABLED Write a 1 to PCSCR<PAR_PORT_DIS>[1] (disable parallel port control)
4	
Microcycle 2	
1	
2	PARALLEL PORT CONTROL NOW DISABLED[2]
3	Write data for MIB scan chain bit<91> to PCSCR<DATA>[1] Write 1 to PCSCR<RWL_SHIFT>[1]
4	
Microcycle 3	
1	
2	
3	Write data for MIB scan chain bit<90> to PCSCR<DATA> Write 1 to PCSCR<RWL_SHIFT>
4	Data for MIB scan chain bit<91> shifted into MIB scan chain bit<0>[2]
Microcycle 4	
1	
2	

[1] An S5 operation.

[2] Note 1-cycle delay between some PCSCR fields and MIB scan chain.

Table 9–1 (Cont.): Example: Writing an Entry in the Patchable Control Store

Phase	Action
	Microcycle 4
3	Write data for MIB scan chain bit<89> to PCSCR<DATA> Write 1 to PCSCR<RWL_SHIFT>
4	Data for MIB scan chain bit<90> shifted into MIB scan chain bit<0> Data for MIB scan chain bit<91> shifted into MIB scan chain bit<1>
	Microcycle 94
1	
2	
3	Write 1 to PCSCR<PCS_WRITE>[1] (write data into patchable control store)
4	Data for MIB scan chain bit<91> shifted into MIB scan chain bit<91>
	Microcycle 95
1	
2	
3	
4	DATA WRITTEN INTO PCS ENTRY FROM MIB SCAN CHAIN[2]

[1] An S5 operation.

[2] Note 1-cycle delay between some PCSCR fields and MIB scan chain.

Note that this example assumed no stalls within the Ebox. Also note that PCSCR<PCS_ENB> and PCSCR<PAR_PORT_DIS> must be re-written with the correct values every cycle that PCSCR<DATA> is written.

Table 9–2: Contents of MIB Scan Chain, When Loading Patchable Control Store

Position	Description	Comment
<91>	MIB_H<0>	Microword Field BRANCH.OFFSET[1]
<90>	MIB_H<1>	"
<89>	MIB_H<2>	"
<88>	MIB_H<3>	"
<87>	MIB_H<4>	"
<86>	MIB_H<5>	"
<85>	MIB_H<6>	"

[1] See Chapter 6 for details on microword fields.

Table 9–2 (Cont.): Contents of MIB Scan Chain, When Loading Patchable Control Store

Position	Description	Comment
<84>	MIB_H<7>	"
<83>	MIB_H<34>	Microword Field L
<82>	MIB_H<49>	Microword Field MISC1
<81>	MIB_H<48>	"
<80>	MIB_H<47>	"
<79>	MIB_H<46>	"
<78>	MIB_H<60>	Microword Field FMT
<77>	MIB_H<19>	Microword Field MISC
<76>	MIB_H<18>	"
<75>	MIB_H<17>	"
<74>	MIB_H<16>	"
<73>	MIB_H<15>	"
<72>	MIB_H<31>	Microword Field DST
<71>	MIB_H<30>	"
<70>	MIB_H<29>	"
<69>	MIB_H<28>	"
<68>	MIB_H<27>	"
<67>	MIB_H<26>	"
<66>	MIB_H<25>	Microword Field A
<65>	MIB_H<24>	"
<64>	MIB_H<23>	"
<63>	MIB_H<22>	"
<62>	MIB_H<21>	"
<61>	MIB_H<20>	"
<60>	CAM MICROADDRESS<10>	Microaddress to be patched
<59>	CAM MICROADDRESS<9>	"
<58>	CAM MICROADDRESS<8>	"
<57>	CAM MICROADDRESS<7>	"
<56>	CAM MICROADDRESS<6>	"
<55>	CAM MICROADDRESS<5>	"
<54>	CAM MICROADDRESS<4>	"
<53>	CAM MICROADDRESS<3>	"
<52>	CAM MICROADDRESS<2>	"
<51>	CAM MICROADDRESS<1>	"
<50>	CAM MICROADDRESS<0>	"
<49>	MIB_H<10>	Microword Field SEQ.COND

Table 9–2 (Cont.): Contents of MIB Scan Chain, When Loading Patchable Control Store

Position	Description	Comment
<48>	MIB_H<9>	"
<47>	MIB_H<8>	"
<46>	MIB_H<14>	Microword Field SEQ.FMT
<45>	MIB_H<13>	Microword Field SEQ.CALL
<44>	MIB_H<12>	Microword Field SEQ.COND
<43>	MIB_H<11>	"
<42>	MIB_H<39>	Microword Field B
<41>	MIB_H<38>	"
<40>	MIB_H<37>	"
<39>	MIB_H<36>	"
<38>	MIB_H<35>	"
<37>	MIB_H<44>	Microword Field MISC2
<36>	MIB_H<43>	"
<35>	MIB_H<42>	"
<34>	MIB_H<41>	"
<33>	MIB_H<45>	Microword Field LIT
<32>	MIB_H<40>	Microword Field D
<31>	MIB_H<54>	Microword Field MRQ
<30>	MIB_H<53>	"
<29>	MIB_H<52>	"
<28>	MIB_H<51>	"
<27>	MIB_H<50>	"
<26>	MIB_H<33>	Microword Field W
<25>	MIB_H<32>	Microword Field V
<24>	MIB_H<59>	Microword Field ALU
<23>	MIB_H<58>	"
<22>	MIB_H<57>	"
<21>	MIB_H<56>	"
<20>	MIB_H<55>	"
<19>	PCS ENTRY SELECT<19>	Entry in PCS to be written
<18>	PCS ENTRY SELECT<18>	"
<17>	PCS ENTRY SELECT<17>	"
<16>	PCS ENTRY SELECT<16>	"
<15>	PCS ENTRY SELECT<15>	"
<14>	PCS ENTRY SELECT<14>	"
<13>	PCS ENTRY SELECT<13>	"

Table 9–2 (Cont.): Contents of MIB Scan Chain, When Loading Patchable Control Store

Position	Description	Comment
<12>	PCS ENTRY SELECT<12>	"
<11>	PCS ENTRY SELECT<11>	"
<10>	PCS ENTRY SELECT<10>	"
<9>	PCS ENTRY SELECT<9>	"
<8>	PCS ENTRY SELECT<8>	"
<7>	PCS ENTRY SELECT<7>	"
<6>	PCS ENTRY SELECT<6>	"
<5>	PCS ENTRY SELECT<5>	"
<4>	PCS ENTRY SELECT<4>	"
<3>	PCS ENTRY SELECT<3>	"
<2>	PCS ENTRY SELECT<2>	"
<1>	PCS ENTRY SELECT<1>	"
<0>	PCS ENTRY SELECT<0>	"

9.2.2.2 Microsequencer Control Field of Microcode

The microsequencer control field of the NVAX microword is used to help select the next microword address. The next address source is explicitly coded in the current microword; there is no concept of sequential next address.

The SEQ.FMT field, bit <14> of the microsequencer control field, selects between the following two formats:

Figure 9–2: Microcode Microsequencer Control Field Formats

```
           14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
          +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
JUMP      | 0|  |     |               J                |
          +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
            |  |  |
            |  |  +--- SEQ.MUX
            |  +--- SEQ.CALL
            +--- SEQ.FMT

           14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
          +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
BRANCH    | 1|  |   SEQ.COND   |     BRANCH.OFFSET     |
          +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
            |  |
            |  +--- SEQ.CALL
            +--- SEQ.FMT
```

Table 9–3: Jump Format Control Field Definitions

Name	Extent	Description
SEQ.FMT	14	0 for JUMP
SEQ.CALL	13	Controls whether return address is pushed on microstack
SEQ.MUX	12:11	Selects source of next microaddress
J	10:0	JUMP target address

Table 9–4: Branch Format Control Field Definitions

Name	Extent	Description
SEQ.FMT	14	1 for BRANCH
SEQ.CALL	13	Controls whether return address is pushed on microstack
SEQ.COND	12:8	Selects source of Microtest Bus
BRANCH.OFFSET	7:0	Page offset of next microinstruction

9.2.2.2.1 Jump Format

Jump format microinstructions choose the next address from one of three possible sources: the J field (bits<10:0> of the current microword), the microstack, or the last cycle logic. The microword fields decode as follows:

Table 9–5: Jump Format Control Field Decodes

SEQ.CALL	SEQ.MUX	NEXT ADDRESS SOURCE	REMARKS
0	0	J	JUMP microinstruction.
1	0	J	CALL microinstruction. Current microword address with bits <3:0> incremented by one is pushed onto microstack.
X	1	STACK	RETURN microinstruction. Top entry of microstack is selected.
X	2	Last Cycle Logic	Last cycle. Select next microflow.
X	3	Last Cycle Logic	Last cycle and enable integer overflow trap. Select next microflow.

On a CALL microinstruction, the address of the current microinstruction, with bits <3:0> incremented by one, is pushed onto the Microstack. The CALL address is modified to avoid a RETURN to the CALL address, which would cause an infinite loop.

9.2.2.2.2 Branch Format

Branch format microinstructions allow the microcoder to perform CASE operations on NVAX state. The SEQ.COND field drives the microtest bus select lines which select the source that drives the microtest bus. (Refer to Section 9.2.3.1.1 for details.) The microtest bus is OR'd with bits <3:1> of the BRANCH.OFFSET field, allowing up to an eight-way case. Casing may be reduced to two-way or four-way by setting to ones the appropriate bits in BRANCH.OFFSET<3:1>.

Table 9–6: Branch Format Control Field Decodes

SEQ.CALL	NEXT ADDRESS SOURCE	REMARKS
0	BRANCH.OFFSET	BRANCH microinstruction.
1	BRANCH.OFFSET	CONDITIONAL CALL microinstruction. Current microword address with bits<3:0> incremented by one is pushed onto microstack.

As in the JUMP format, the SEQ.CALL field is used to indicate that a RETURN address must be pushed on the microstack.

For the purposes of BRANCH microinstructions, the control store is divided into 256-microword pages. The target of a branch microinstruction must be in the same page as the BRANCH as only the least significant 8 bits of the address are modified. The BRANCH.OFFSET field is the destination address offset within the current page.

A branch address is made up as follows:

Table 9–7: Branch Address Formation

Bit(s)	Source
<10:8>	Current Address<10:8>
<7:4>	BRANCH.OFFSET<7:4>
<3:1>	BRANCH.OFFSET<3:1> OR UTEST<2:0>
<0>	BRANCH.OFFSET<0>

9.2.2.3 MIB Latches

The microword output from the Control Store 8-to-1 multiplexer is latched in Φ_1 into the Control Store Microinstruction Buffer (CS_MIB) latch. The microword output from the Patchable Control Store is also latched in Φ_1, into the PCS_MIB latch. The outputs of the CS_MIB and PCS_MIB latches drive a multiplexer, which selects the PCS_MIB output if the CAL output hit in the Patchable Control Store CAM; otherwise, the multiplexer selects the CS_MIB output.

Bits <14:0> of the multiplexer output (the Microsequencer Microinstruction, E_USQ_CSM%UMIB_H<14:0>) are driven back to the microsequencer; all bits are driven to the Microinstruction Buffer (MIB) latch which operates in Φ_2. Bits <60:14,12:11> of the MIB latch output (E_USQ%MIB_H) are driven to S3 of the Ebox; all bits are driven to the MIB scan chain (see Section 9.5.2). When a microtrap is detected, the contents of the MIB latch are forced to NOP. The MIB latch is stalled on a microsequencer stall.

9.2.3 Next Address Logic

The remainder of the microsequencer is devoted to determining the next control store lookup address. There are five next address sources:

1. JUMP/BRANCH.OFFSET field of Microword
2. Microtrap Logic
3. Last Cycle Logic
4. Microstack
5. Test Address Generator

9.2.3.1 CAL and CAL INPUT BUS

The CAL, or Current Address Latch, is a static latch which holds the 11 bit address used to access the control store. It operates in Φ_3, and is stalled on a microsequencer stall. Bits <10:8> are also "stalled" when forming a branch address (see Table 9–7).

The input to the CAL is the CAL Input Bus (E_USQ_BUS%CAL_INPUT_L). The CAL Input Bus is a dynamic bus, precharged in Φ_2. The selected next address source drives this bus in Φ_3. Bits <14,12:11> of the microsequencer control field are used in selecting three of the next address sources: E_USQ_CSM%UMIB_H<10:0> (for a BRANCH or JUMP address), the output of the last cycle logic, and the microstack output. The fourth CAL Input Bus source is the microtrap address; if a microtrap is detected, this input is selected regardless of the value of E_USQ_CSM%UMIB_H<14,12:11>. The fifth source is a test address, driven from the Test Address Generator. This input has the highest priority. In summary:

Table 9–8: Current Address Selection

TEST ADDR	TRAP DETECTED	SEQ.FMT <14>	SEQ.MUX <12:11>	NEXT ADDRESS SOURCE	REMARKS
0	0	1	XX	Branch Address[1]	BRANCH/CONDITIONAL CALL microinstructions
0	0	0	00	J	JUMP/CALL microinstructions
0	0	0	01	Microstack	RETURN microinstruction
0	0	0	1X	Last Cycle Logic	Start new microflow
0	1	X	XX	Microtrap Logic	Microtrap
1	X	X	XX	Test Address Generator	Test address

[1] See Table 9–7

9.2.3.1.1 Microtest Bus

The microtest bus allows conditional branches and conditional calls based on information generated outside the microsequencer, such as Ebox condition codes. The SEQ.COND field of the BRANCH format is driven on the microtest select lines, E_USQ%UTSEL_H<4:0>, in Φ_{23}. These lines are decoded by all conditional information sources in the Ebox. The selected source drives its information on the microtest bus, E_BUS%UTEST_L<2:0>. E_BUS%UTEST_L must be valid in time to be OR'd with value on the CAL Input Bus and latched in the CAL in Φ_3.

The sources for the microtest bus are as follows:

Table 9–9: Microtest Bus Sources

UTSEL<4:0>	Select	UTEST<2:0>
00	No source	000
01	ALU.NZV[2]	ALU_CC.N,ALU_CC.Z,ALU_CC.V
02	ALU.NZC[2]	ALU_CC.N,ALU_CC.Z,ALU_CC.C
03	B.2-0[1]	EB_BUS<2:0>
04	B.5-3[1]	EB_BUS<5:3>
05	A.7-5[1]	EA_BUS<7:5>
06	A.15-12[1]	EA_BUS<15:14>, EA_BUS<13> OR EA_BUS<12>
07	A31.BQA.BNZ1[1]	EA_BUS<31>, EB_BUS<2:0> = 0, EB_BUS<15:8> NEQ 0
08	MPU.0-6[2]	MPU0_6<2:0>
09	MPU.7-13[2]	MPU7_13<2:0>
0A	STATE.2-0[2]	STATE<2:0>
0B	STATE.5-3[2]	STATE<5:3>
0C	OPCODE.2-0[1]	OPCODE<2:0>
0D	PSL.26-24[3]	PSL<26:24>
0E	PSL.29.23-22[3]	PSL<29>, PSL<23:22>
0F	SHF.NZ[2],INT	SHF_CC.N, SHF_CC.Z, INTERRUPT_REQUEST
10	VECTOR,TEST	ECR<VECTOR_UNIT_PRESENT>[3], TEST DATA, TEST STROBE
11	FBOX	Encoded fault<1:0>[4], ECR<FBOX.ENABLED> = 0[3]
12	FQ.VR[1]	0, FIELD_QUEUE_NOT_VALID, FIELD_QUEUE_RMODE
13-1F	Not Used	

[1] Data is taken from S3.

[2] Data is taken from S4.

[3] Data is taken from S6.

[4] See Section 8.5.19.7.

The microtest select lines are always driven with bits <12:8> of the MIB output regardless of the microinstruction format. The microtest bus is only OR'd with the CAL Input Bus if the BRANCH source is selected to drive that bus.

Two of the microtest sources, the Field Queue (FQ) and the Mask Processing Unit (MPU), perform some function based on the value of the microtest select lines. These functions must check SEQ.FMT, **E_USQ%MIB_H<14>**, for validity of the microtest select lines.

The microtest select lines are precharged to a value of zero during Φ_1; no microtest source is selected for this value.

9.2.3.2 Microtrap Logic

Microtraps allow the microcoder to deal with abnormal events that require immediate service. When a microtrap occurs, the microcode control is transferred to a service microroutine. Operations further behind in the pipe than the one which caused the microtrap are aborted.

Microtraps are generated by the Ebox, Mbox, or Ibox. Those Ebox microtrap requests considered faults are asserted in S4 of the microinstruction in which they occurred. Those that are considered traps are asserted in S5 of the microinstruction in which they occurred.

Microtraps have higher priority than all other next address sources except the Test Address Generator. Microtraps are detected in Φ_4. The microtrap signals are OR'd together in Φ_1 to form **E_USQ%PE_ABORT_L**. The trap signals are prioritized and address lookup is done to select the appropriate microtrap handler address, which is driven on the CAL Input Bus in Φ_3.

Since microtraps are not detected until Φ_4, too late for control store access in that cycle, the signal **E_USQ%PE_ABORT_L** is used to force NOPs in all the Ebox and microsequencer inter-stage latches in Φ_1 and Φ_2. This effectively flushes the pipe. In the cycle following microtrap detection, control store access is done using the microtrap handler address, and the first microword of the trap handler is driven to S3 on **E_USQ%MIB_H**.

Microtrap microcode flows flush the Ebox, Fbox, the specifier queue in the Mbox, the Instruction Queue in the microsequencer, and the Ibox. The only exception to this is the branch mispredict microtrap, which does not flush the Ibox. The microtrap handler also loads a new PC which allows the Ibox to start prefetching. At the end of the microtrap, microcode control is returned to the last cycle logic.

Microtrap signals must be asserted for only one cycle, to prevent multiple detections of the same trap.

9.2.3.2.1 Microtraps

1. **Powerup**

   The powerup microtrap is requested when the chip is powered up. This forces the internal state of the chip to a known condition. See Chapter 16 for details.

2. **Asynchronous Hardware Error**

   The asynchronous hardware error microtrap request can happen at any time regardless of what is in the pipeline. The following conditions cause execution of this microtrap:

   - S3 Stall Timer Expiration

     The S3 stall timer counts the number of consecutive cycles that S3 is stalled. When the counter reaches its limit, it initiates the Asynchronous Hardware Error microtrap by asserting **E_TIM%S3_TIMEOUT_H**. See Section 8.5.25.1 for more detail concerning the timer.

   - Translation Buffer Parity Error

If the Mbox detects a TB parity error, it initiates the Asynchronous Hardware Error microtrap by asserting **M%TB_PERR_TRAP_L**.

3. **Integer Overflow**

   The integer overflow microtrap request, **E_FLT%IOVFL_L**, is asserted in S5 when the Ebox detects an integer overflow condition (see Section 8.5.19.3) during the last cycle of a macroinstruction with overflow checking enabled. The microinstruction that checked the overflow condition completes, but any microinstruction initiated after it is aborted.

4. **Branch Mispredict**

   A branch mispredict microtrap request, **E_PSL%BRANCH_MISPREDICT_H**, is asserted in S5 by the Ebox when the output of the Branch Queue (the Ibox branch prediction) does not match the branch direction calculated by the Ebox. See Section 8.5.19.3.

5. **Reserved Instruction Fault**

   The Ebox initiates the reserved instruction microtrap in S4 when the Fbox is disabled and any Fbox instruction other than MULL is issued. It asserts **E_FLT%RSVD_INSTR_L** to initiate the microtrap.

6. **Hardware Errors**

   The Ebox hardware error microtrap request, **E_FLT%HW_ERR_H**, is asserted in S4 on operand-related hardware errors, such as the attempted access of an MD register which has its error bit set.

7. **Memory Management Exceptions**

   - Reported by Mbox

     An explicit read or write request by the Ebox can result in a memory management exception. This causes the Mbox to assert the microtrap request signal, **M%MME_TRAP_L**. See Section 12.5.1.5.3.7 for further detail.

   - Reported by Ebox

     A memory management fault can also occur during a memory access initiated by the Ibox, such as for an opcode or operand specifier. When this happens the Ibox asserts **I%IMEM_MEXC_H**. The Ebox combines this signal with several other conditions to generate **E_FLT%MME_ERR_H**. It initiates the memory management microtrap by asserting **E_FLT%MME_ERR_H** in S4. See Section 8.5.15.14 and Section 8.5.19 for more detail.

8. **Reserved Addressing Mode**

   A reserved addressing mode fault occurs when the Ibox detects a reserved addressing mode on an operand specifier. The reserved addressing mode microtrap request, **E_FLT%RSVD_ADDR_MODE_H**, is asserted in S4 by the Ebox. Refer to Section 8.5.15.14 and Section 8.5.19 for details.

9. **Floating Point Fault**

   A floating point fault is a fault detected by the Fbox. If the current entry of the retire queue points to the Fbox, the request **E_FLT%FLOATING_FAULT_H** can be asserted. If the retire queue points to the Ebox, the request is stalled until the retire queue does point to the Fbox. There are four possible causes for assertion of **E_FLT%FLOATING_FAULT_H**: floating overflow, floating underflow, reserved operand, and floating divide by zero. The trap handler cases on the floating point fault code on the microtest bus. See Section 8.5.16.5 for further detail.

9.2.3.2.2 Microtrap Request Timing

The exceptions which result in microtrap requests to the microsequencer are detected in different pipeline seqments. In addition, some microtrap requests are delayed in order to align the request with a particular pipeline segment.

The following table gives the pipeline segment in which the exception is detected and the pipeline segment in which the microtrap request is made for each type of microtrap.

Table 9–10: Microtrap Request Timing

Microtrap	Exception Detected	Microtrap Requested
Powerup	N/A	N/A
Asynchronous Hardware Error, S3 Stall Timer	S3	S3
Asynchronous Hardware Error, TB Parity Error	N/A	N/A
Integer Overflow	S5	S5
Branch Mispredict	S5	S5
Reserved Instruction Fault	S3	S4
Hardware Error	S3,S4	S4
Memory Management Exception, Mbox	N/A	N/A
Memory Management Exception, Ebox	S3.S4	S4
Reserved Addressing Mode	S3,S4	S4
Floating Point Faults	S4	S4

9.2.3.2.3 Prioritization of Microtraps

Microtraps must be prioritized since more than one request may be asserted at a time. Microtrap priorities and microtrap handler addresses are given in the following table.

Table 9–11: Microtraps

Priority	Microtrap	Dispatch Address (Hex)
1	Powerup	00
2	Asynchronous hardware errors	04
3	Integer overflow	08
4	Branch mispredict	0C
5	Reserved instruction fault	10
6	Hardware error	14
7	Memory management exceptions	18
8	Reserved addressing mode faults	1C
9	Floating point faults	20

The priorities of the microtraps are assigned utilizing the following dependencies:

1. The chip must be placed in a known state upon powerup.
2. Once in a known state, asynchronous hardware errors take precedence over all, since they indicate a serious problem.
3. Microtrap requests issued in S5 have priority over those in S4 since they are further down the pipe.
4. Opcode faults take priority over operand faults.
5. Of the requests issued in S4, whichever physically took place first (was forwarded the furthest) has priority.
6. Architecturally defined faults or traps (i.e. integer overflow) have priority over implementation defined faults or traps (i.e. branch mispredict).
7. Reserved addressing mode faults are mutually exclusive of operand memory management faults for the same operand, because the source queue is empty before a reserved addressing mode fault request is made.
8. The floating point fault may only be requested when the retire queue points to the Fbox.

9.2.3.2.4 Erroneous Microtrap Interruption

A window of at least 4 cycles exists between initiation of a microtrap (assertion of E_USQ%PE_ABORT_L) and decoding of RESET.CPU for all microtraps except Branch Mispredict. (A subset of the RESET.CPU operations is performed immediately on detection of branch mispredict.) During this window, a lower priority microtrap based on state which will be cleared by RESET.CPU must not be allowed to interrupt the higher priority microtrap which has begun execution. This restriction is met by the following rules:

- **Powerup**

  Powerup can interrupt any microtrap as it has the highest priority. The powerup microtrap is initiated by deassertion of K_E%RESET_L. Assertion of K_E%RESET_L causes all NVAX state to be initialized, so no microtraps will occur to interrupt powerup based on previous state.
- **Asynchronous Hardware Error**

  Asynchronous hardware errors can interrupt any microtrap but Powerup. Due to the effects of K_E%RESET_L described above, no special logic is needed to meet this constraint.
- **Ebox-Generated Microtraps**

  All Ebox-generated microtrap requests (integer overflow, branch mispredict, reserved instruction fault, Ebox hardware error, Ebox memory management exception, reserved addressing mode, and floating point faults) are cleared within the Ebox immediately on assertion of E_USQ%PE_ABORT_L. Thus, none of these microtraps can interrupt another.
- **Mbox-Generated Microtrap**

  The Mbox memory management exception can occur at any time between assertion of E_USQ%PE_ABORT_L and decoding of RESET.CPU, if an Ebox-initiated memory reference is outstanding. The following list describes the possibility of assertion of M%MME_TRAP_L (initiation of the Mbox memory management exception microtrap) during each type of microtrap.
  - Powerup:

    As described above, M%MME_TRAP_L cannot be asserted.
  - Asynchronous Hardware Error:

By the nature of these errors, the Ebox may be performing any operation during initiation of this microtrap, so **M%MME_TRAP_L** could be asserted.

- Integer Overflow, Branch Mispredict:

  Detection of these traps occurs only on the last cycle of a microflow, in S5. All outstanding Ebox-initiated memory references which could produce an error have been completed by this time, so **M%MME_TRAP_L** cannot be asserted.

- Reserved Instruction Fault:

  Initiation of this microtrap occurs in S4, on the first cycle of the microflow for the offending instruction. If that same microword begins an Ebox-initiated memory reference, the reference will be aborted on initiation of the microtrap.

  On initiation of a Reserved Instruction Fault microtrap, S5 can only contain the last microword of the previous microflow. As described above, **M%MME_TRAP_L** could not be asserted at that point.

- Ebox Hardware Error, Ebox Memory Management Exception, Reserved Address Mode:

  These faults are generated during operand access. By microcode convention, no operands are referenced while there is an outstanding Ebox-initiated memory reference. Thus, **M%MME_TRAP_L** cannot be asserted.

- Mbox Memory Management Exception:

  Multiple Ebox-initiated memory references can be outstanding at any time, so a second Mbox Memory Management Exception could occur.

- Floating Point Fault:

  Similar to the Reserved Instruction Fault, this fault is detected in S4, with the first result transfer from the Fbox. Any memory reference initiated during this cycle will be aborted on initiation of the microtrap. S5 could only contain the last cycle of a microflow. Thus **M%MME_TRAP_L** cannot be asserted.

In summary, the Mbox Memory Management Exception microtrap is the only trap which could incorrectly interrupt a higher priority microtrap in this window. In order to prevent this error, detection of the Mbox Memory Management Exception is blocked at the microtrap logic for the cycles from microtrap initiation (assertion of **E_USQ%PE_ABORT_L**) through execution of RESET.CPU (assertion of **E_MSC%EARLY_FLUSH_EBOX_H**). Mbox Memory Management Exception detection is enabled again in the cycle following execution of RESET.CPU.

Branch Mispredict is the only microtrap for which RESET.CPU is not executed. In this case, **E_MSC%EARLY_FLUSH_EBOX_H** is asserted in the same cycle as **E_USQ%PE_ABORT_L**; therefore, detection of the Mbox Memory Management Exception is only blocked at the microsequencer for one cycle. However, as described above, **M%MME_TRAP_L** cannot be asserted during the Branch Mispredict microtrap, so the blocking is not necessary for proper execution of this microtrap.

9.2.3.2.5 Microtrap Detection Abort Effects

The microsequencer aborts operation on detection of a microtrap (assertion of **E_USQ%PE_ABORT_L**). The following table shows the timing for all microsequencer logic that is cleared or reset on an abort.

Table 9–12: Abort Effects in the Microsequencer

Phase	What is Cleared/Reset
Φ_1	E_USQ_STL%LATE_USQ_STALL_L
Φ_2	E_USQ%MIB_H to S3
	E_USQ%MACRO_1ST_CYCLE_H to S3
	E%FBOX_1ST_CYCLE_L to Fbox
Φ_3	E_USQ_STL%VERY_LATE_USQ_STALL_L
	E%FBOX_1ST_CYCLE_L to Fbox
	E_USQ%MACRO_1ST_CYCLE_H master latch
Φ_4	-

9.2.3.3 Last Cycle Logic

The last cycle logic examines several conditions used to determine which new microflow is to be taken when LAST.CYCLE or LAST.CYCLE.OVERFLOW is detected on E_USQ_CSM%UMIB_H, no microtraps are detected, and no test address is driven. There are five possible new microflows, listed in order of priority:

1. Interrupt Request Handler
2. Trace Fault Handler
3. First Part Done Handler
4. Instruction Queue Stall
5. The macroinstruction microcode indicated by the top entry in the instruction queue.

The last cycle logic prioritizes these sources and performs address lookup. In addition, the signal E_USQ_LST%SELECT_IQ_H is derived. This signal is asserted when a valid entry is taken from the instruction queue.

Table 9–13: Microaddresses for Last Cycle Interrupts or Exceptions

Priority	Interrupt or Exception	Dispatch Address (Hex)
1	Interrupt request	24
2	Trace fault	28
3	First part done	2C
4	Instruction Queue Stall	30

The priorities in the last cycle logic are assigned using the following dependencies:

1. Interrupts and trace faults must be handled between instructions. (Interrupts may also be serviced at defined points during long instructions such as string instructions; this servicing is handled by microcode.)
2. By definition, an interrupt that is permitted to request service has a higher priority level (IPL) than any exception that occurs in the process to be interrupted, or any instruction to be executed by that process.

3. When tracing is enabled (E_PSL%PSL_H<TP> is set), a trace fault must be taken before the execution of each instruction.
4. If an instruction begins execution with PSL<FPD> set, the first part done handler must be entered rather than the normal entry point for the instruction.
5. PSL<TP> and PSL<FPD> cannot both be set when an instruction begins execution. In order for PSL<FPD> to be set, the instruction must have been interrupted previously; the interrupt handler always clears PSL<TP> before saving the PSL when interrupting an instruction. (Note that the interrupt handler does not clear PSL<TP> when the interrupt is taken between instructions.)
6. The Instruction Queue Stall microword is executed if an opcode is requested from the Instruction Queue but the queue is empty.

9.2.3.3.1 Interrupts

Interrupt servicing is requested by the Ebox by assertion of **E%INT_REQ_H**. For more information on interrupts, see Chapter 10.

9.2.3.3.2 Trace Fault

A trace fault should be requested when the PSL<TP> bit is set. Due to the pipelined implementation of the Ebox, a local version of the PSL<TP> bit must be maintained; thus, the trace fault is actually requested when LOCAL_TP is asserted.

There are two cases that must be considered in setting LOCAL_TP. In the first case, a macroinstruction starts execution with PSL<T> set. This is the normal program tracing mode. LOCAL_TP must be set immediately after the macroinstruction begins execution. In the second case, an interrupt was taken at the end of a macrointruction, and the trace must be taken when interrupt processing completes. In this case, PSL<TP> is set, and LOCAL_TP is asserted. LOCAL_TP is also updated whenever the PSL is written. LOCAL_TP is cleared by loading the PSL as a longword, with a value of 0 in the <TP> bit.

9.2.3.3.3 First Part Done

The first part done handler is selected when PSL<FPD> is asserted and the instruction queue output is valid. The top entry in the instruction queue is removed (**E_USQ_LST%SELECT_IQ_H** is asserted), but the last cycle address is the first part done handler address, rather than the dispatch taken from the instruction queue.

If PSL<FPD> is asserted and the instruction queue is empty, the Instruction Queue Stall microword is selected.

9.2.3.3.3.1 Interaction with Reserved Instructions

The Ibox detects unimplemented instructions (such as POLYx), and causes the microcode to enter the reserved instruction fault handler by placing the microaddress for that handler in the dispatch field of the instruction queue entry for the unimplemented instruction. However, if PSL<FPD> is asserted, the last cycle logic selects the first part done handler rather than the reserved instruction fault handler. The first part done handler detects this case and branches to the reserved instruction fault handler.

9.2.3.3.4 Instruction Queue

The instruction queue is a FIFO filled by the Ibox. This queue permits the Ibox to fetch and decode instructions ahead of Ebox execution.

The instruction queue is 6 entries deep. Each entry is 22 bits long. The format of each entry is as follows:

Figure 9–3: Instruction Queue Entry Format

```
 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|        OPCODE            | DL  |FI|        DISPATCH          | V|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Table 9–14: Instruction Queue Entry Format Field Definitions

Name	Extent	Description
OPCODE	21:13	9-bit opcode of the instruction.
DL	12:11	Initial data length of instruction operands.
FI	10	Set if entry is an Fbox instruction.
DISPATCH	9:1	Microcode address of the instruction's microflow.
V	0	Set if entry is valid.

The instruction queue entry indicated by the write pointer is written in Φ_4. The write pointer is advanced in Φ_2 if the valid bit is set in the new queue entry.

The instruction queue entry indicated by the read pointer is read in Φ_1. The address used to access the control store is derived from the instruction queue entry as follows:

Table 9–15: Control Store Address Formation

Bit(s)	Value
<10>	0
<9:1>	IQ entry DISPATCH field
<0>	0

If the valid bit of the entry being read is set, and the instruction queue is selected as the CAL Input Bus source, **E_USQ%MACRO_1ST_CYCLE_H** is asserted and driven to the Ebox in Φ_1 of S3. This signal is cleared on a microtrap, and stalls on a microsequencer stall. If the first cycle of an Fbox instruction is detected(<FI> is asserted), the signal **E%FBOX_1ST_CYCLE_L** is also asserted, and driven to the Fbox in Φ_{23} of S2. This signal is only asserted once per instruction; it is not stalled on a microsequencer stall.

If the valid bit of the entry to be read is not set, and the instruction queue is selected as the CAL Input Bus source, the last cycle logic selects the Instruction Queue Stall microaddress (030#16), which is used to look up the stall microword in the control store. The stall microword is a NOP for the Ebox; it selects the last cycle logic again in the microsequencer. In addition to driving the stall microword to the Ebox, E_USQ%IQ_STALL_H is asserted in Φ_1 of S2. This signal, in conjunction with memory management and hardware error signals driven by the Ibox, is used by the Ebox to detect instruction stream referencing errors.

The read pointer is advanced in Φ_3 if E_USQ_CSM%UMIB_H selects the last cycle logic, the last cycle logic selects the instruction queue, and the valid bit in the queue entry that was read out is set. When the read pointer is advanced, the valid bit in the entry read out is cleared. The read pointer is stalled on a microsequencer stall.

The instruction queue is flushed when the Ebox decodes RESET.CPU from the MIB (E_MSC%EARLY_FLUSH_EBOX_H is asserted). The pointers are reset, and the entry valid bits are cleared.

Table 9–16 shows the phase-by-phase events that occur on an instruction queue stall. Initially, the read and write pointers both have a value of 4; the queue is empty.

Table 9–16: Instruction Queue Operation

Phase	Action
	Microcycle 1
1	E_USQ_CSM%UMIB_H = LAST.CYCLE E_USQ%IQ_STALL_H asserted
2	Last microword of instruction flow driven to S3
3	CAL = IQ stall address
4	Write I%IQ_BUS_H to Entry[4] (value = valid data)
	Microcycle 2
1	E_USQ_CSM%UMIB_H = LAST.CYCLE E_USQ_INQ%IQ_OUT_H = Entry[4] E_USQ_LST%SELECT_IQ_H asssserted E_USQ%IQ_STALL_H deasserted
2	NOP microword driven to S3 Increment write pointer (pointer=5)
3	CAL = Entry[4] Increment read pointer (pointer=5) Clear valid bit in Entry[4]
4	Write I%IQ_BUS_H to Entry[5] (value = valid data)
	Microcycle 3
1	E_USQ_CSM%UMIB_H = microsequencer field of first microword E_USQ%MACRO_1ST_CYCLE_H asserted
2	First microword of new instruction flow driven to S3 Increment write pointer (pointer=6)
3	
4	

9.2.3.3.4.1 Instruction Context Latches

The instruction queue drives the dispatch address to the last cycle logic. The remainder of the queue entry (DL,OPCODE,FI) is latched in the instruction context (ICTX) latches. The format is as follows:

Figure 9–4: Instruction Context Format

```
 11 10 09 08|07 06 05 04|03 02 01 00
+---+---+---+---+---+---+---+---+---+---+---+---+
|          OPCODE                   |  DL   | FI|
+---+---+---+---+---+---+---+---+---+---+---+---+
```

Table 9–17: Instruction Context Format Field Definitions

Name	Extent	Description
OPCODE	11:3	9-bit opcode of the instruction.
DL	2:1	Initial data length of instruction operands.
FI	0	Set if entry is an Fbox instruction.

The output of the queue is latched every Φ_2 for hold-time reasons. The ICTX master latch operates in Φ_4 of S2, and is loaded from the queue output latch only when a valid entry is removed from the instruction queue (**E_USQ_LST%SELECT_IQ_H** is asserted). The ICTX slave latch operates in Φ_1 of S3; its output (**E_USQ%ICTX_H**) is driven to the Ebox. The instruction context latches are only valid when their respective pipeline stages are executing macroinstructions.

Both the master and slave latches are stalled on a microsequencer stall. The slave latch is stalled holding the correct value for the current S3 cycle, and the master latch is stalled holding the correct value for the next cycle.

The opcode portion of the instruction context (**E%FOPCODE_H**) is driven to the Fbox from the instruction queue output latch, in Φ_2 of S2.

9.2.3.4 Microstack

Frequently used microcode can be made into microsubroutines. When a microsubroutine is called, the return address is pushed onto the microstack. The output of the microstack is driven on the CAL Input Bus when a RETURN is decoded from the **E_USQ_CSM%UMIB_H**, no microtraps are detected, and no test address is driven.

The microstack is 6 entries deep. It is a circular stack, with the write pointer always one entry ahead of the read pointer. Each entry is an 11-bit control store address. The addresses stored in the microstack incorporate any modification done by the microtest bus.

Every Φ_1, the entry indicated by the microstack read pointer is read out into a Φ_1 latch, where it is held to be driven on the CAL Input Bus in Φ_3. Also in Φ_1, the RETURN address is written into the entry ahead of the microstack read pointer. The RETURN address is formed by adding 1 to bits <3:0> of the CALL address in the CAL. Bits <10:4> are unchanged.

The microstack pointer is incremented in Φ_4 on a CALL or CONDITIONAL CALL microinstruction; it is decremented on a RETURN microinstruction. The microstack pointer is stalled on a microsequencer stall. It is only reset when the chip reset signal, **K_E%RESET_L**, is asserted.

Figure 9–5: Microstack Organization

```
POINTER                    ARRAY
+------+         +-------------------------------------+
|  0   |         |                                     |
+------+         +-------------------------------------+
|  1   |         |       First Call writes here        |
+------+         +-------------------------------------+
|  2   |---+--->|       Pointer = 2  read entry        |
+------+   |     +-------------------------------------+
|  3   |   +--->|       Pointer = 2  write entry       |
+------+         +-------------------------------------+
|  4   |         |                                     |
+------+         +-------------------------------------+
|  5   |         |                                     |
+------+         +-------------------------------------+
```

Consider a CALL followed immediately by a RETURN with an initial microstack pointer value of 2. Table 9–18 shows the phase-by-phase operation of the microstack during the next three cycles.

```
X:          CALL Y
X+1:        {next microword}
.
.
.
Y:          RETURN
```

Table 9–18: Microstack Pointer Example

Phase	Action
	Microcycle 1
1	
2	
3	CAL = X
4	
	Microcycle 2
1	Write X+1[1] to Array[3] USTACK_OUT<10:0>=Array[2] E_USQ_CSM%UMIB_H = CALL
2	
3	CAL = Y
4	Increment microstack pointer (pointer=3)

[1] Assumption: the result of the increment to bits<3:0> of X is X+1.

Table 9–18 (Cont.): Microstack Pointer Example

Phase	Action
	Microcycle 3
1	Write Y+1 to Array[4] USTACK_OUT<10:0>=Array[3] (value = X+1) E_USQ_CSM%UMIB_H = RETURN
2	
3	CAL = X+1
4	Decrement microstack pointer (pointer=2)

9.2.4 Stall Logic

The microsequencer is stalled whenever S3 is stalled. The Ebox derives the signal **E_STL%USEQ_STALL_H** which is used to stall the microsequencer. The microsequencer creates delayed versions of this signal as needed to stall various latches. The signals **E_USQ%PE_ABORT_L** (asserted on initiation of a microtrap) and **E_USQ_TST%FORCE_TEST_ADDR_L** (asserted on detection of the Test Address Generator driving a control store microaddress, see Section 9.5) break a microsequencer stall by clearing the delayed versions of **E_STL%USEQ_STALL_H**.

The following table shows the timing for all stallable logic in the microsequencer.

Table 9–19: Stall Timing in the Microsequencer

Phase	What Stalls
Φ_1	ICTX slave latch to S3
	E_USQ%MACRO_1ST_CYCLE_H latch to S3
Φ_2	E_USQ%MIB_H to S3
Φ_3	Current Address Latch
	E_USQ%MACRO_1ST_CYCLE_H master latch
	Instruction queue read pointer
Φ_4	ICTX master latch to S3
	Microstack pointer

9.3 Initialization

A reset (assertion of **K_E%RESET_L**) causes the microsequencer to initialize in the following state:

- A powerup microtrap is initiated (see Table 9–12 for microtrap ABORT effects).
- The microstack pointer is reset to zero.
- The instruction queue valid bits are flushed and its pointers are reset by **E_MSC%EARLY_FLUSH_EBOX_H**.
- The Patchable Control Store CAM is disabled, since PCSCR<PCS_ENB> is cleared in the Ebox.

- The MIB scan chain is controlled by the Parallel Test Port command pins, since PCSCR<PAR_PORT_DIS> is cleared in the Ebox.
- The Test Address Generator is reset to an address value of zero.

9.4 Microcode Restrictions

1. Every microtrap except Branch Mispredict must contain a RESET.CPU in order to reset the Instruction Queue. (The Ebox is flushed automatically, clearing the queues, on detection of branch mispredict.) RESET.CPU must not be issued within the 3 microwords preceding LAST.CYCLE in order to allow time for the Instruction Queue to be cleared (if RESET.CPU is present in microword N, LAST.CYCLE cannot be present until microword N+4).
2. For correct operation of Trace Fault and First Part Done in the Last Cycle Logic, PSL<T,TP,FPD> must not be changed within the 2 microwords preceeding LAST.CYCLE (if any of these PSL bits are changed in microword N, LAST.CYCLE cannot be present until microword N+3).
3. No Ebox-initiated memory requests can be made in the last cycle of a microflow, other than writes with the translation already known to be valid.
4. No Ebox-initiated memory requests can be outstanding when the microcode references an operand (queue entry or register file location).
5. The instruction queue stall microword must indicate LAST.CYCLE.
6. PSL<TP> must be cleared by the interrupt handler before it allows execution of an interrupted instruction to resume.
7. The Patchable Control Store (PCS) WRITE command, issued by writing a "1" into PCSCR<PCS_WRITE> in microinstruction N, must not be followed by a PCS ENABLE command (issued by writing a "1" into PCSCR<PCS_ENB>) before microinstruction N+2.
8. Following the writing of the Patchable Control Store ENABLE bit (PCSCR<PCS_ENB>) in S5 by microinstruction N, the first microinstruction for which Patchable Control Store can be considered enabled is microinstruction N+4.
9. The First Part Done microflow must check for the case in which an unimplemented instruction begins execution with PSL<FPD> set. In this case, microcode must branch to the Reserved Instruction Fault microflow, rather than executing the normal First Part Done microflow.

9.5 Testability

9.5.1 Test Address

The control store microaddress is both controllable and observable. A microcode address can be driven to the microsequencer from the Test Address Generator. The Test Address Generator is an 11-bit counter which is initialized to a value of zero on assertion of **K_E%RESET_L**. It increments its address counter once on each deassertion of **T%CS_TEST_H**, thus cycling through all possible control store addresses.

This microaddress source takes priority over all others. To ensure immediate control store lookup using this microaddress, assertion of **T%CS_TEST_H** sets an S/R latch whose output is **E_USQ_TST%FORCE_TEST_ADDR_L**. Assertion of this signal breaks any stall on Φ_2, Φ_3, and Φ_4 latches in the microsequencer. This allows the control store to operate, driving the selected microword into the MIB scan chain (see Section 9.5.2). The Ebox stall(s), if any, are unaffected, along with stalls on Φ_1 latches in the microsequencer.

E_USQ_TST%FORCE_TEST_ADDR_L is deasserted when the Test Address Generator has completed generation of all possible addresses (when its counter overflows).

The microaddress driven from the CAL can be be observed on the Parallel Test Port data pins under control of the Parallel Test Port command pins. The microsequencer drives to the Parallel Test Port in Φ_1.

Figure 9–6: Parallel Port Output Format

```
 11 10 09 08|07 06 05 04|03 02 01
+--+--+--+--+--+--+--+--+--+--+--+
|         CAL OUTPUT             |
+--+--+--+--+--+--+--+--+--+--+--+
```

Table 9–20: Parallel Port Output Format Field Definitions

Name	Extent	Description
CAL OUTPUT	11:1	Microaddress driven from CAL

9.5.2 MIB Scan Chain

A 92-bit scan chain is present at the output of the MIB, allowing the complete microword to be latched and scanned out of the chip. The scan chain master latches operate in Φ_4; the slave latches operate in Φ_2. In observe mode, the scan chain is loaded and shifted under control of the Parallel Test Port command pins. When scanning out, MIB scan chain bit<91> is the first bit to reach the Parallel Test Port.

Note that control of the MIB scan chain must be given to the parallel port during this operation, by writing a 0 to PCSCR<PAR_PORT_DIS>. See Section 8.5.22.1 for details.

Table 9–21: Contents of MIB Scan Chain, in Observe Mode

Position	Description	Comment
<91>	**E_USQ%MIB_H<0>**	Microword Field BRANCH.OFFSET[1]
<90>	**E_USQ%MIB_H<1>**	"
<89>	**E_USQ%MIB_H<2>**	"
<88>	**E_USQ%MIB_H<3>**	"
<87>	**E_USQ%MIB_H<4>**	"
<86>	**E_USQ%MIB_H<5>**	"
<85>	**E_USQ%MIB_H<6>**	"
<84>	**E_USQ%MIB_H<7>**	"

[1] See Chapter 6 for details on microword fields.

Table 9–21 (Cont.): Contents of MIB Scan Chain, in Observe Mode

Position	Description	Comment
<83>	E_USQ%MIB_H<34>	Microword Field L
<82>	E_USQ%MIB_H<49>	Microword Field MISC1
<81>	E_USQ%MIB_H<48>	"
<80>	E_USQ%MIB_H<47>	"
<79>	E_USQ%MIB_H<46>	"
<78>	E_USQ%MIB_H<60>	Microword Field FMT
<77>	E_USQ%MIB_H<19>	Microword Field MISC
<76>	E_USQ%MIB_H<18>	"
<75>	E_USQ%MIB_H<17>	"
<74>	E_USQ%MIB_H<16>	"
<73>	E_USQ%MIB_H<15>	"
<72>	E_USQ%MIB_H<31>	Microword Field DST
<71>	E_USQ%MIB_H<30>	"
<70>	E_USQ%MIB_H<29>	"
<69>	E_USQ%MIB_H<28>	"
<68>	E_USQ%MIB_H<27>	"
<67>	E_USQ%MIB_H<26>	"
<66>	E_USQ%MIB_H<25>	Microword Field A
<65>	E_USQ%MIB_H<24>	"
<64>	E_USQ%MIB_H<23>	"
<63>	E_USQ%MIB_H<22>	"
<62>	E_USQ%MIB_H<21>	"
<61>	E_USQ%MIB_H<20>	"
<60>	Value Undefined	No Observe Input
<59>	Value Undefined	No Observe Input
<58>	Value Undefined	No Observe Input
<57>	Value Undefined	No Observe Input
<56>	Value Undefined	No Observe Input
<55>	Value Undefined	No Observe Input
<54>	Value Undefined	No Observe Input
<53>	Value Undefined	No Observe Input
<52>	Value Undefined	No Observe Input
<51>	Value Undefined	No Observe Input
<50>	Value Undefined	No Observe Input
<49>	E_USQ%MIB_H<10>	Microword Field SEQ.COND
<48>	E_USQ%MIB_H<9>	"

Table 9–21 (Cont.): Contents of MIB Scan Chain, in Observe Mode

Position	Description	Comment
<47>	E_USQ%MIB_H<8>	"
<46>	E_USQ%MIB_H<14>	Microword Field SEQ.FMT
<45>	E_USQ%MIB_H<13>	Microword Field SEQ.CALL
<44>	E_USQ%MIB_H<12>	Microword Field SEQ.COND
<43>	E_USQ%MIB_H<11>	"
<42>	E_USQ%MIB_H<39>	Microword Field B
<41>	E_USQ%MIB_H<38>	"
<40>	E_USQ%MIB_H<37>	"
<39>	E_USQ%MIB_H<36>	"
<38>	E_USQ%MIB_H<35>	"
<37>	E_USQ%MIB_H<44>	Microword Field MISC2
<36>	E_USQ%MIB_H<43>	"
<35>	E_USQ%MIB_H<42>	"
<34>	E_USQ%MIB_H<41>	"
<33>	E_USQ%MIB_H<45>	Microword Field LIT
<32>	E_USQ%MIB_H<40>	Microword Field D
<31>	E_USQ%MIB_H<54>	Microword Field MRQ
<30>	E_USQ%MIB_H<53>	"
<29>	E_USQ%MIB_H<52>	"
<28>	E_USQ%MIB_H<51>	"
<27>	E_USQ%MIB_H<50>	"
<26>	E_USQ%MIB_H<33>	Microword Field W
<25>	E_USQ%MIB_H<32>	Microword Field V
<24>	E_USQ%MIB_H<59>	Microword Field ALU
<23>	E_USQ%MIB_H<58>	"
<22>	E_USQ%MIB_H<57>	"
<21>	E_USQ%MIB_H<56>	"
<20>	E_USQ%MIB_H<55>	"
<19>	Value Undefined	No Observe Input
<18>	Value Undefined	No Observe Input
<17>	Value Undefined	No Observe Input
<16>	Value Undefined	No Observe Input
<15>	Value Undefined	No Observe Input
<14>	Value Undefined	No Observe Input
<13>	Value Undefined	No Observe Input
<12>	Value Undefined	No Observe Input

Table 9–21 (Cont.): Contents of MIB Scan Chain, in Observe Mode

Position	Description	Comment
<11>	Value Undefined	No Observe Input
<10>	Value Undefined	No Observe Input
<9>	Value Undefined	No Observe Input
<8>	Value Undefined	No Observe Input
<7>	Value Undefined	No Observe Input
<6>	Value Undefined	No Observe Input
<5>	Value Undefined	No Observe Input
<4>	Value Undefined	No Observe Input
<3>	Value Undefined	No Observe Input
<2>	Value Undefined	No Observe Input
<1>	Value Undefined	No Observe Input
<0>	Value Undefined	No Observe Input

9.6 Signal Cross Reference

Note that the signal names used in this specification are the schematic signal names.

Table 9–22: Schematic Signal Names, in Alphabetical Order

Schematic Signal Name	Behavioral Model Signal Name
E%FBOX_1ST_CYCLE_L	E%FBOX_1ST_CYCLE_L
E%FOPCODE_H	E%FOPCODE_H
E%INT_REQ_H	E%INT_REQ_H
E_BUS%UTEST_L	E_BUS%UTEST_H
E_FLT%FLOATING_FAULT_H	E%FLOATING_FAULT_H
E_FLT%HW_ERR_H	E%HW_ERR_H
E_FLT%IOVFL_L	E%IOVFL_H
E_FLT%MME_ERR_H	E%MME_ERR_H
E_FLT%RSVD_ADDR_MODE_H	E%RSVD_ADDR_MODE_H
E_FLT%RSVD_INSTR_L	E%RSVD_INSTR_FAULT_H
E_MSC%EARLY_FLUSH_EBOX_H	E_MSC%EARLY_FLUSH_EBOX_H
E_PSL%BRANCH_MISPREDICT_H	E%BRANCH_MISPREDICT_H
E_PSL%PSL_H	E_PSL%PSL_H
E_STL%USEQ_STALL_H	E_STL%USEQ_STALL_H
E_TIM%S3_TIMEOUT_H	E_TIM%S3_TIMEOUT_H
E_USQ%ICTX_H	E_USQ%ICTX_H
E_USQ%IQ_STALL_H	E_USQ%IQ_STALL_H
E_USQ%MACRO_1ST_CYCLE_H	E_USQ%MACRO_1ST_CYCLE_H
E_USQ%MIB_H	E_USQ%MIB_H
E_USQ%PE_ABORT_L	E_USQ%PE_ABORT_H
E_USQ%UTSEL_H	E_USQ%UTSEL_H
E_USQ_BUS%CAL_INPUT_L	E_USQ_BUS%CAL_INPUT_L
E_USQ_CAL%CAL_H	E_USQ_CAL%CAL_H
E_USQ_CSM%UMIB_H	E_USQ_CSM%UMIB_H
E_USQ_INQ%IQ_OUT_H	E_USQ_INQ%IQ_OUT_H
E_USQ_LST%SELECT_IQ_H	E_USQ_LST%SELECT_IQ_H
E_USQ_STL%LATE_USQ_STALL_L	E_USQ_STL%LATE_USQ_STALL_L
E_USQ_STL%VERY_LATE_USQ_STALL_L	E_USQ_STL%VERY_LATE_USQ_STALL_L
E_USQ_TST%FORCE_TEST_ADDR_L	E_USQ_TST%FORCE_TEST_ADDR_L
I%IMEM_MEXC_H	I%IMEM_MEXC_H
I%IQ_BUS_H	I%IQ_BUS_H

Table 9–22 (Cont.): Schematic Signal Names, in Alphabetical Order

Schematic Signal Name	Behavioral Model Signal Name
K_E%RESET_L	K%RESET_L
M%MME_TRAP_L	M%MME_TRAP_H
M%TB_PERR_TRAP_L	M%TB_PERR_TRAP_H
T%CS_TEST_H	T%CS_TEST_H

Table 9–23: Behavioral Model Signal Names, in Alphabetical Order

Behavioral Model Signal Name	Schematic Signal Name
E%BRANCH_MISPREDICT_H	E_PSL%BRANCH_MISPREDICT_H
E%FBOX_1ST_CYCLE_L	E%FBOX_1ST_CYCLE_L
E%FLOATING_FAULT_H	E_FLT%FLOATING_FAULT_H
E%FOPCODE_H	E%FOPCODE_H
E%HW_ERR_H	E_FLT%HW_ERR_H
E%INT_REQ_H	E%INT_REQ_H
E%IOVFL_H	E_FLT%IOVFL_L
E%MME_ERR_H	E_FLT%MME_ERR_H
E%RSVD_ADDR_MODE_H	E_FLT%RSVD_ADDR_MODE_H
E%RSVD_INSTR_FAULT_H	E_FLT%RSVD_INSTR_L
E_BUS%UTEST_H	E_BUS%UTEST_L
E_MSC%EARLY_FLUSH_EBOX_H	E_MSC%EARLY_FLUSH_EBOX_H
E_PSL%PSL_H	E_PSL%PSL_H
E_STL%USEQ_STALL_H	E_STL%USEQ_STALL_H
E_TIM%S3_TIMEOUT_H	E_TIM%S3_TIMEOUT_H
E_USQ%IQ_STALL_H	E_USQ%IQ_STALL_H
E_USQ%MACRO_1ST_CYCLE_H	E_USQ%MACRO_1ST_CYCLE_H
E_USQ%MIB_H	E_USQ%MIB_H
E_USQ%PE_ABORT_H	E_USQ%PE_ABORT_L
E_USQ%UTSEL_H	E_USQ%UTSEL_H
E_USQ_BUS%CAL_INPUT_L	E_USQ_BUS%CAL_INPUT_L
E_USQ_CAL%CAL_H	E_USQ_CAL%CAL_H
E_USQ_CSM%UMIB_H	E_USQ_CSM%UMIB_H
E_USQ%ICTX_H	E_USQ%ICTX_H
E_USQ_INQ%IQ_OUT_H	E_USQ_INQ%IQ_OUT_H
E_USQ_LST%SELECT_IQ_H	E_USQ_LST%SELECT_IQ_H
E_USQ_STL%LATE_USQ_STALL_L	E_USQ_STL%LATE_USQ_STALL_L
E_USQ_STL%VERY_LATE_USQ_STALL_L	E_USQ_STL%VERY_LATE_USQ_STALL_L

Table 9–23 (Cont.): Behavioral Model Signal Names, in Alphabetical Order

Behavioral Model Signal Name	Schematic Signal Name
E_USQ_TST%FORCE_TEST_ADDR_L	E_USQ_TST%FORCE_TEST_ADDR_L
I%IMEM_MEXC_H	I%IMEM_MEXC_H
I%IQ_BUS_H	I%IQ_BUS_H
K%RESET_L	K_E%RESET_L
M%MME_TRAP_H	M%MME_TRAP_L
M%TB_PERR_TRAP_H	M%TB_PERR_TRAP_L
T%CS_TEST_H	T%CS_TEST_H

9.7 Revision History

Table 9–24: Revision History

Rev	Who	When	Description of change
0.0	Elizabeth M. Cooper	06-Mar-1989	Release for external review.
0.1	Elizabeth M. Cooper	14-Sep-1989	Post-modelling update.
0.5	Elizabeth M. Cooper	10-Dec-1989	Updates for Rev 0.5 spec release.
0.5A	Elizabeth M. Cooper	5-Jan-1990	Remove vector microtrap and V bit from IQ.
0.5B	Elizabeth M. Cooper	20-Jun-1990	Accumulated updates.
0.6A	Elizabeth M. Cooper	26-Nov-1990	Final updates.
0.6B	Elizabeth M. Cooper, Tim C. Fischer	12-Dec-1990	Final final updates.
0.6C	Elizabeth M. Cooper	1-Jan-1991	Add signal cross reference tables.
0.6D	Elizabeth M. Cooper	13-Feb-1991	Add description of patch revision.

Chapter 10

The Interrupt Section

10.1 Overview

The interrupt section receives interrupt requests from both internal and external sources, and compares the IPL associated with the interrupt request to the current interrupt level in the PSL. If the interrupt request is for an IPL that is higher than the current PSL IPL, the interrupt section signals an interrupt request to the microsequencer which will initiate a microcode interrupt handler at the next macroinstruction boundary.

When an interrupt is serviced by the Ebox microcode, the interrupt section provides an encoded interrupt ID on **E_BUS%ABUS_L<20:16>**, which allows the microcode to determine the highest priority interrupt request that is pending. Interrupt requests are cleared in one of three ways, depending on the type of request.

Software interrupt requests are supported via a 15-bit SISR register, which is read and written by the microcode, and which makes requests to the interrupt generation logic.

Both full and subset interval timer support is provided, based on the state of the ICCS_EXT bit in the ECR processor register, as described in Section 8.5.22. If ECR<ICCS_EXT>=0, a subset interval timer is supported by implementing the interrupt enable bit of the ICCS processor register in internal logic. If ECR<ICCS_EXT>=1, a full interval timer is supported, and external logic must implement the full ICCS, ICR, and NICR processor registers. In this instance, reads from and writes to these registers are converted to I/O space addresses and transmitted off-chip, as described in Section 2.12, Processor Registers.

10.2 Interrupt Summary

Interrupt requests received from external logic are divided into two categories: those received by edge-sensitive logic, and those received by level-sensitive logic. Both are synchronized to internal clocks. In addition, there are several internal sources of interrupt requests.

10.2.1 External Interrupt Requests Received by Edge-Sensitive Logic

Five of the external interrupt requests are received by edge-sensitive logic and synchronized to internal clocks. These signals request the following special-purpose interrupts.

- **P%HALT_L**: The assertion of **P%HALT_L** causes the CPU to enter the console at IPL 1F (hex) at the next macroinstruction boundary. This interrupt is not gated by the current IPL, and always results in console entry, even if the IPL is already 1F (hex). Note that the implementation of this event is different from a normal interrupt in which a PC/PSL pair are pushed onto the interrupt stack. For this event, the current PC, PSL, and halt code are stored in the SAVPC and SAVPSL processor registers. The mechanism by which the console is entered, and a description of the SAVPC and SAVPSL processor registers is given in Section 15.4, Console Halt and Halt Interrupt.
- **P%PWRFL_L**: The assertion of **P%PWRFL_L** indicates that a power failure is pending. This results in the dispatch of the interrupt to the operating system at IPL 1E (hex) through SCB vector 0C (hex).
- **P%H_ERR_L**: The assertion of **P%H_ERR_L** indicates that a hard error has been detected in the system environment. This results in the dispatch of the interrupt to the operating system at IPL 1D (hex) through SCB vector 60 (hex).
- **P%S_ERR_L**: The assertion of **P%S_ERR_L** indicates that a soft error has been detected in the system environment. This results in the dispatch of the interrupt to the operating system at IPL 1A (hex) through SCB vector 54 (hex).
- **P%INT_TIM_L**: The assertion of **P%INT_TIM_L** indicates that the interval timer period has expired. If the interrupt enable bit in the ICCS processor register is set (whether this bit is implemented internally or externally), an interrupt is dispatched to the operating system at IPL 16 (hex) through SCB vector C0 (hex). If ICCS<6> is not set, no interrupt is dispatched.

Each signal must make a high-to-low transition to assert the interrupt request. A pseudo-edge detect circuit is used to capture this transition asynchronously. Details of the edge detect logic given in Section 10.3.1. Because these are special-purpose interrupt requests with an implied SCB vector, no acknowledgement of the interrupt is required. Ebox microcode explicitly clears the interrupt request when the interrupt is serviced.

10.2.2 External Interrupt Requests Received by Level-Sensitive Logic

Four of the external interrupt requests are received by level-sensitive logic and synchronized to internal clocks. These signals request general-purpose interrupts at the following IPLs.

Interrupt	Request IPL	
Request	(Hex)	(Dec)
P%IRQ_L<3>	17	23
P%IRQ_L<2>	16	22
P%IRQ_L<1>	15	21
P%IRQ_L<0>	14	20

Each signal must be driven low and remain low to assert the interrupt request. When one of these interrupts is to be serviced, the Ebox microcode acknowledges the interrupt by issuing an

NDAL read of word length to one of four longword-aligned interrupt vector offset registers to obtain the SCB offset through which the interrupt should be dispatched. The address of the register depends on the interrupt being serviced, as shown in Table 10–1.

Table 10–1: Interrupt Vector Offset Registers

Interrupt Request	Vector Offset Register Address	Processor Register[1]
P%IRQ_L<3>	E100010C	IAK17
P%IRQ_L<2>	E1000108	IAK16
P%IRQ_L<1>	E1000104	IAK15
P%IRQ_L<0>	E1000100	IAK14

[1] Direct access to the interrupt vector offset registers is provided via processor register reads for system test. Software references to these processor registers during normal system operation can result in UNDEFINED behavior

In response, the microcode expects to receive an interrupt SCB vector offset, which is shown in Figure 10–1. The fields are described in Table 10–2.

Figure 10–1: Interrupt SCB Vector Offset

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| x  x  x  x  x  x  x  x  x  x  x  x  x  x  x  x|        System Control Block Offset        |PR|IL| :IAK1x
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Table 10–2: Interrupt SCB Vector Offset

Name	Extent	Description
IL	0	Interrupt Level Override. In normal operation, the IPL at which the interrupt is serviced is implied by the request signal that was asserted. If the IL bit is set in the interrupt vector offset, the IPL at which the interrupt is taken is forced to 17 (hex). This capability supports external buses, such as the Q-bus, that can not guarantee that the device that responds with the interrupt SCB vector offset is the device that originally requested the interrupt. For example, the Q-bus has four separate interrupt request signals that correspond to **P%IRQ_L<3:0>** but only one signal to daisy chain the interrupt grant. Furthermore, devices on the Q-bus are ordered so that higher priority devices are electrically closer to the bus master. If an **P%IRQ_L<1>** request is being serviced, there is no guarantee that a higher priority device will not intercept the grant. Software must determine the level of the device that was serviced and set the IPL to the correct value.

Table 10–2 (Cont.): Interrupt SCB Vector Offset

Name	Extent	Description
PR	1	Passive Release Flag. In certain circumstances, notably in multi-processor configurations, an interrupt may be requested but removed by the time the microcode acknowledges it by reading the interrupt vector offset register. If the PR bit is set in the interrupt SCB vector offset, the microcode treats this interrupt as an internal passive release and resumes the interrupted instruction stream without dispatching the interrupt. If the interrupt request is deasserted before the microcode reads the interrupt ID, the ID will be zero, indicating that no interrupt is pending. In that instance, no read of the interrupt vector offset register is done, and the microcode generates an immediate passive release.
	15:2	Longword offset from the start of the SCB of the vector to use to dispatch this interrupt. After zero-extending to longword length, microcode adds this value to the contents of the SCBB register, reads that location, and uses it as the SCB vector with which to dispatch the interrupt to the operating system.

NOTE

If both the PR and IL bits are set in the interrupt SCB vector offset, the PR bit takes priority and a passive release is done.

10.2.3 Internal Interrupt Requests

The Cbox, Ibox, and Mbox report error conditions by asserting internal interrupt request signals that are logically ORed with the synchronized versions of P%H_ERR_L and P%S_ERR_L. These requests are then handled in exactly the same manner as requests generated by external sources, as specified above. The following table details the internal interrupt sources

Table 10–3: Internal Interrupt Requests

Signal	Source	Type
C%CBOX_H_ERR_H	CBOX	H_ERR_L
C%CBOX_S_ERR_H	CBOX	S_ERR_L
I%IBOX_S_ERR_L	IBOX	S_ERR_L
M%MBOX_S_ERROR_H	MBOX	S_ERR_L

The performance monitoring facility requests an interrupt at IPL 1B (hex) when the performance counters become half full. The performance monitoring hardware asserts the signal E_PMN%PMON_L to perform this request. This request is serviced entirely by microcode, and cleared by writing to the appropriate bit in the ISR. Chapter 18 should be consulted more details about the Peformance Monitoring facilities.

Architecturally defined software interrupt requests are implemented through an internal register in the interrupt section. Under control of the SISR and SIRR processor registers which are described in Chapter 2, the Ebox microcode sets the appropriate bit in this register, which then results in the dispatch of the interrupt to the operating system at an IPL and through the SCB vector implied by the interrupt request. The association between the interrupt request, requested IPL, and SCB vector for these requests is shown in the following table.

Table 10–4: Software Interrupts

	Request IPL		SCB Vector
SISR bit	(Hex)	(Dec)	(Hex)
SISR<15>	0F	15	BC
SISR<14>	0E	14	B8
SISR<13>	0D	13	B4
SISR<12>	0C	12	B0
SISR<11>	0B	11	AC
SISR<10>	0A	10	A8
SISR<09>	09	09	A4
SISR<08>	08	08	A0
SISR<07>	07	07	9C
SISR<06>	06	06	98
SISR<05>	05	05	94
SISR<04>	04	04	90
SISR<03>	03	03	8C
SISR<02>	02	02	88
SISR<01>	01	01	84

Ebox microcode explicitly clears the interrupt request when the interrupt is serviced.

10.2.4 Special Considerations for Interval Timer Interrupts

The NVAX CPU may be configured to support either a subset interval timer, or a full interval timer, depending on the state of ECR<ICCS_EXT>, as described in Section 8.5.22, Ebox IPRs. Console firmware initializes this bit to the correct state based on the system environment in which the CPU chip is used.

The internal implementation of the interval timer interrupt request gates the assertion of **P%INT_TIM_L** with the internal copy of the interrupt enable bit of the ICCS processor register (ICCS<6>). The CPU chip does not know the source of the signal driving **P%INT_TIM_L**, and this fact is used to allow the implementation of both a subset and full interval timer.

If ECR<ICCS_EXT>=0, an SRM-approved subset interval timer may be implemented by driving **P%INT_TIM_L** with an oscillator whose period is 10ms. In this mode, the NICR and ICR processor registers are not required nor implemented, and microcode maintains the subset ICCS processor register with an internal copy of only the interrupt enable bit from ICCS<6>. References to the ICCS processor register affect only ICCS<6>, and are handled internally without being transmitted on the NDAL.

If ECR<ICCS_EXT>=1, a full interval timer consisting of the ICCS, NICR, and ICR processor registers may be implemented in external logic. **P%INT_TIM_L** is asserted when the programmed interval has expired. Processor register references to the ICCS, NICR, and ICR processor registers are converted to I/O space references and transmitted onto the NDAL, as described in Section 2.12, Processor Registers. However, even in this mode, microcode maintains the internal copy of ICCS<6> consistent with a write to ICCS that is transmitted onto the NDAL. As a result, if interrupts are enabled in the off-chip ICCS register, they are also allowed by the internal ICCS interrupt enable bit. Conversely, if interrupts are disabled in the off-chip ICCS register, they are also disabled by the internal bit. External logic is expected to return all 32 bits when the ICCS processor register is read, including the correct state of the interrupt enable bit. Microcode does not attempt to merge the external data with the internal copy of ICCS<6> to satisfy a processor register read of ICCS.

It should be noted that ECR<ICCS_EXT> has no effect on the operation of the interrupt section hardware. It is used strictly as a control bit which directs the microcode operation of references to the ICCS processor register. Independent of the state of ECR<ICCS_EXT>, processor register writes to ICCS cause microcode to update the internal copy of the interrupt enable bit. If ECR<ICCS_EXT>=1, references to the ICCS processor register are also transmitted onto the NDAL. References to the NICR and ICR processor registers are always transmitted onto the NDAL; they are simply not used if the system implements a subset interval timer.

Table 10–5 gives a summary of the results of references to the ICCS, NICR, and ICR processor registers, with both states of ECR<ICCS_EXT>.

Table 10–5: References to Interval Timer Processor Registers

Operation	ECR<ICCS_EXT>=0	ECR<ICCS_EXT>=1
MTPR x,#PR$_ICCS	Update internal ICCS<6>	Update internal ICCS<6>, write data to E1000060[1]
MFPR #PR$_ICCS,x	Return internal ICCS<6>	Read and return data from E1000060[1]
MTPR x,#PR$_NICR	Write data to E1000064[1]	Write data to E1000064[1]
MFPR #PR$_NICR,x	Read and return data from E1000064[1]	Read and return data from E1000064[1]
MTPR x,#PR$_ICR	Write data to E1000068[1]	Write data to E1000068[1]
MFPR #PR$_ICR,x	Read and return data from E1000068[1]	Read and return data from E1000068[1]

[1] See Section 2.12

10.2.5 Priority of Interrupt Requests

When multiple interrupt requests are pending, the interrupt section prioritizes the requests. Table 10–6 shows the relative priority (from highest to lowest) of all interrupt requests. For reference, this table also includes the IPL at which the interrupt is taken, and the SCB vector through which the interrupt is dispatched.

Table 10–6: Relative Interrupt Priority

Interrupt	Request IPL		SCB Vector	
Request	(Hex)	(Dec)	(Hex)	
P%HALT_L	1F	31	None[1]	Highest priority
P%PWRFL_L	1E	30	0C	
P%H_ERR_L[2]	1D	29	60	
E_PMN%PMON_L	1B	27	None[5]	
P%S_ERR_L[2]	1A	26	54	
P%IRQ_L<3>	17	23	Specified by device[3]	
P%IRQ_L<2>	16	22	Specified by device[3]	
P%INT_TIM_L[4]	16	22	C0	
P%IRQ_L<1>	15	21	Specified by device[3]	
P%IRQ_L<0>	14	20	Specified by device[3]	
SISR<15>	0F	15	BC	
SISR<14>	0E	14	B8	
SISR<13>	0D	13	B4	
SISR<12>	0C	12	B0	
SISR<11>	0B	11	AC	
SISR<10>	0A	10	A8	
SISR<09>	09	09	A4	
SISR<08>	08	08	A0	
SISR<07>	07	07	9C	
SISR<06>	06	06	98	
SISR<05>	05	05	94	
SISR<04>	04	04	90	
SISR<03>	03	03	8C	
SISR<02>	02	02	88	
SISR<01>	01	01	84	Lowest priority

[1] Direct dispatch to console; PC, PSL placed in SAVPC, SAVPSL processor registers

[2] Includes Cbox, Ibox, and Mbox internally generated requests

[3] SCB vector offset supplied by the device

[4] When enabled by the internal ICCS<6>

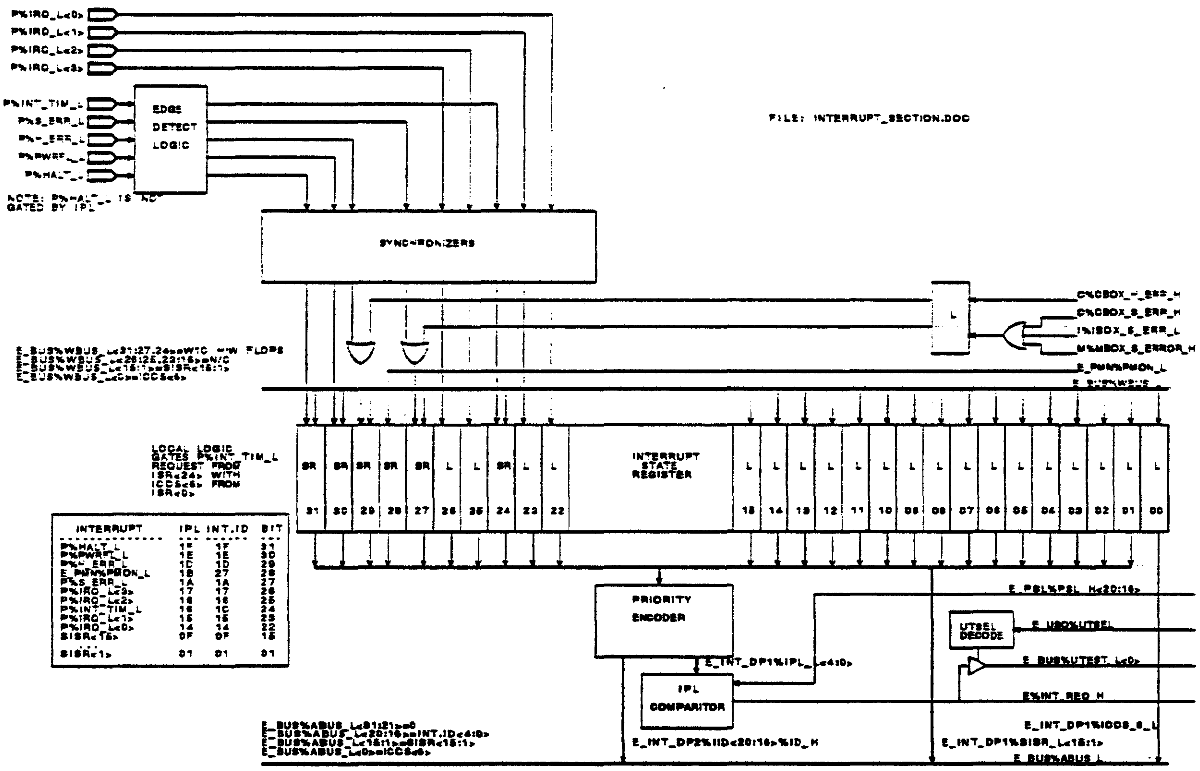
[5] Interrupt processed entirely by microcode

The **P%IRQ_L<2>** request takes priority over the **P%INT_TIM_L** request, both of which are at IPL 16 (hex). Inter-processor interrupts in multi-processor systems are requested via **P%IRQ_L<2>**, and they must take priority over interval timer requests.

10.3 Interrupt Section Structure

The interrupt section consists of three basic components: the edge detect and synchronization logic, the interrupt state register (ISR), and the interrupt generation logic. A block diagram of the interrupt section is shown in Figure 10–2.

Figure 10–2: Interrupt Section Block Diagram



INTERRUPT	IPL	INT.ID	BIT
P%HALT_L	1F	1F	31
P%PWRFL_L	1E	1E	30
P%H_ERR_L	1D	1D	29
E_PMN%PMON_L	1B	27	28
P%S_ERR_L	1A	1A	27
P%IRQ_L<3>	17	17	26
P%IRQ_L<2>	16	16	25
P%INT_TIM_L	16	1C	24
P%IRQ_L<1>	15	15	23
P%IRQ_L<0>	14	14	22
SISR<15>	0F	0F	15
...			
SISR<1>	01	01	01

10.3.1 Edge Detect and Synchronization Logic

10.3.1.1 Edge Detect Circuitry

The pads for the five special-purpose external interrupt request signals contain logic which detects high-to-low transitions on these signals. A falling edge sets an SR flip-flop which begins the interrupt request process. This interrupt request process involves setting another SR flip-flop to register the interrupt. This second flip-flop may only be cleared by microcode. Microcode clears this flip-flop while servicing the interrupt request. The edge detect circuitry resets itself automatically (clearing the first SR) within two NDAL cycles following the low-to-high transition of the pin.

10.3.1.2 Interrupt Synchronization

The pads for all external interrupt request signals (both the edge and level sensitive types) contain synchronizers to allow the use of asynchronous signals for interrupt requests. The pin signals are synchronized to the internal NVAX clocks and are then passed to the ISR. More deterministic timing behavior may be desired in some applications such as during test. This may be achieved by driving the signals synchronously with respect to the input clocks. The chapter on Electrical Characteristics should be consulted for details about setup and hold times.

10.3.2 Interrupt State Register

The interrupt state register is a composite register that implements the 15-bit architecturally defined SISR register, the internal copy of the interrupt enable bit from the ICCS processor register. the interrupt latch for the performance monitoring facility interrupt, and the interrupt request latches for the 5 special-purpose and 4 general-purpose interrupts. The ISR contains two kinds of elements: SR flops for the special-purpose interrupt requests, and latches for the other requests. The following table lists the types and positions of all elements in the ISR.

ISR bit	State Element	Description
31	SR	Interrupt request for **P%HALT_L** interrupt
30	SR	Interrupt request for **P%PWRFL_L** interrupt
29	SR	Interrupt request for **P%H_ERR_L** and internal hard error interrupts
28	SR	Interrupt request for **E_PMN%PMON_L**, the performance monitoring facility interrupt
27	SR	Interrupt request for **P%S_ERR_L** and internal soft error interrupts
26	L	Interrupt request for **P%IRQ_L<3>** interrupt
25	L	Interrupt request for **P%IRQ_L<2>** interrupt
24	SR	Interrupt request for **P%INT_TIM_L** interrupt
23	L	Interrupt request for **P%IRQ_L<1>** interrupt
22	L	Interrupt request for **P%IRQ_L<0>** interrupt
15:1	L	SISR<15:1> latches and requests for software interrupts
0	L	Internal ICCS<6> latch

State Element

SR—SR flop
L—Latch

Synchronized inputs from the external special-purpose interrupt requests are logically ORed with the internal requests from the Cbox, Ibox, and Mbox. The assertion of one of these signals causes the appropriate request flop to be set in ISR<31:29,27,24>. These request flops are cleared under Ebox microcode control when written with a 1 from the corresponding bits of **E_BUS%WBUS_L**.

Synchronized inputs from the general-purpose interrupt requests are loaded into the appropriate latch in ISR<26:25,23:22>. These request latches are cleared when the interrupting device deasserts the interrupt request in response to a CPU request for an interrupt vector offset.

The performance monitoring facility interrupt request is loaded into the request flop in ISR<28>. The request is cleared under Ebox microcode control when written with a 1 from E_BUS%WBUS_L<28>.

SISR<15:1> is implemented via ISR<15:1>, and is loaded from bits <15:1> of E_BUS%WBUS_L under Ebox microcode control. These request latches are cleared under Ebox microcode control when a new value is loaded from E_BUS%WBUS_L.

The internal copy of the interrupt enable bit in the ICCS processor register (ICCS<6>) is implemented via ISR<0>, and is loaded from E_BUS%WBUS_L<0> under Ebox microcode control. Local logic gates the interval timer request from ISR<24> with the state of ISR<0>.

The interrupt request elements of the interrupt state register (ISR<31:22,15:1>) go to the interrupt generation logic. ISR<0> and ISR<15:1> may also be read onto E_BUS%ABUS_L for return to the Ebox.

10.3.3 Interrupt Generation Logic

The interrupt generation logic priority encodes all interrupt requests from the interrupt state register to determine the highest priority request. The output of the encoder is the request IPL and the interrupt ID of the highest priority request. If any request is pending, the request IPL is compared against E_PSL%PSL_H<20:16> from the Ebox. If the request IPL is higher than the PSL IPL, or if the request is for P%HALT_L (P%HALT_L is not gated by the IPL), E%INT_REQ_H is asserted to the microsequencer.

The assertion of E%INT_REQ_H causes the microsequencer to initiate a microcode interrupt handler at the next macroinstruction boundary. The same signal is available on the microtest bus (E_BUS%UTEST_L<0> as a microbranch condition, which is checked by the Ebox microcode during long instructions.

Along with the request IPL, the interrupt generation logic provides an encoded interrupt ID that identifies the highest priority interrupt. The interrupt ID is read onto bits <20:16> of E_BUS%ABUS_L along with ISR<0> and ISR<15:1> when microcode references the A/INT.SYS source. For each interrupt, the interrupt ID encoding, request IPL, ISR bit number, method for clearing the interrupt, and SCB vector is shown in Table 10–7.

Table 10–7: Summary of Interrupts

Interrupt	Int ID		Request IPL		ISR Bit	Reset	SCB Vector
Request	(Hex)	(Dec)	(Hex)	(Dec)	(Dec)	Method	(Hex)
P%HALT_L	1F	31	1F	31	31	Write 1 to ISR bit	Console Halt
P%PWRFL_L	1E	30	1E	30	30	Write 1 to ISR bit	0C
P%H_ERR_L[1]	1D	29	1D	29	29	Write 1 to ISR bit	60

[1] Includes Cbox, Ibox, and Mbox internally generated requests

Table 10–7 (Cont.): Summary of Interrupts

Interrupt	Int ID		Request IPL		ISR Bit	Reset	SCB Vector
Request	(Hex)	(Dec)	(Hex)	(Dec)	(Dec)	Method	(Hex)
E_PMN%PMON_L	1B	27	1B	27	28[2]	Write 1 to ISR bit	Handled by microcode
P%S_ERR_L[1]	1A	26	1A	26	27[2]	Write 1 to ISR bit	54
P%IRQ_L<3>	17	23	17	23	26	Read IAK17 IPR	Supplied by device
P%IRQ_L<2>	16	22	16	22	25	Read IAK16 IPR	Supplied by device
P%INT_TIM_L	1C[3]	28	16	22	24[2]	Write 1 to ISR bit	C0
P%IRQ_L<1>	15	21	15	21	23	Read IAK15 IPR	Supplied by device
P%IRQ_L<0>	14	20	14	20	22	Read IAK14 IPR	Supplied by device
SISR<15>	0F	15	0F	15	15	Write 0 to ISR bit	BC
SISR<14>	0E	14	0E	14	14	Write 0 to ISR bit	B8
SISR<13>	0D	13	0D	13	13	Write 0 to ISR bit	B4
SISR<12>	0C	12	0C	12	12	Write 0 to ISR bit	B0
SISR<11>	0B	11	0B	11	11	Write 0 to ISR bit	AC
SISR<10>	0A	10	0A	10	10	Write 0 to ISR bit	A8
SISR<09>	09	09	09	09	09	Write 0 to ISR bit	A4
SISR<08>	08	08	08	08	08	Write 0 to ISR bit	A0
SISR<07>	07	07	07	07	07	Write 0 to ISR bit	9C
SISR<06>	06	06	06	06	06	Write 0 to ISR bit	98
SISR<05>	05	05	05	05	05	Write 0 to ISR bit	94
SISR<04>	04	04	04	04	04	Write 0 to ISR bit	90
SISR<03>	03	03	03	03	03	Write 0 to ISR bit	8C
SISR<02>	02	02	02	02	02	Write 0 to ISR bit	88
SISR<01>	01	01	01	01	01	Write 0 to ISR bit	84
No Interrupt	00	00	—	—	—	Dismiss interrupt	—

[1] Includes Cbox, Ibox, and Mbox internally generated requests

[2] Write-1-to-clear ISR bit is different than IPL and interrupt ID

[3] Interrupt ID is different than IPL

The interrupt ID is the same as the request IPL for all interrupt requests except for the interval timer request.

DESIGN CONSTRAINT

A value of zero for the interrupt ID must be returned if an interrupt is no longer present, or if the highest priority interrupt request is no longer higher than the PSL IPL. Normally, once an interrupt request is made, it remains until it is cleared by the microcode. However, the level-sensitive interrupt requests may be deasserted after the interrupt is dispatched, but before the microcode reads the interrupt ID. Therefore, it is possible that the highest remaining interrupt has a request IPL lower than the current PSL IPL. If zero is not returned for the interrupt ID in this instance, the processor will not function correctly.

10.4 Ebox Microcode Interface

The Ebox microcode interfaces with the interrupt section primarily through reads (via **E_BUS%ABUS_L**) and writes (via **E_BUS%WBUS_L**) of the ISR accomplished through the A/INT.SYS and DST/INT.SYS decodes. These decodes provide access to the so-called INT.SYS register, which is shown in Figure 10–3. The fields of the register are listed in Table 10–8.

Figure 10–3: IPR 7A (hex), INTSYS

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  | 0| 0|  | 0| 0| 0|    INT.ID    |                 SISR<15:1>                 |  | :INTSYS
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
  |  |  |  |  |        |                                                                       |
  |  |  |  |  |        |                                                                       +-- ICCS<6>
  |  |  |  |  |        +-- INT_TIM_RESET
  |  |  |  |  +-- S_ERR_RESET
  |  |  |  +-- PMON_RESET
  |  |  +-- H_ERR_RESET
  |  +-- PWRFL_RESET
  +-- HALT_RESET
```

Table 10–8: INTSYS Field Descriptions

Name	Extent	Type	Description
ICCS<6>	0	RW,0	This field contains the internal copy of the interrupt enable bit from the ICCS processor register. It is set to 0 by microcode at powerup.
SISR	15:1	RW,0	This field contains the 15 architecturally-defined software interrupt request bits. It is set to 0 by microcode at powerup.
INT.ID	20:16	RO	This field contains the encoding of the highest priority interrupt request as listed in Table 10–7. Writes to this field are ignored.
INT_TIM_RESET	24	WC,0	Writing a 1 to this field clears the **P%INT_TIM_L** interrupt request. Writing a 0 has no effect on the request. The field is read as a 0 and the interrupt request is cleared by microcode at powerup.

Table 10–8 (Cont.): INT.SYS Register Fields

Name	Extent	Type	Description
S_ERR_RESET	27	WC,0	Writing a 1 to this field clears the **P%S_ERR_L** interrupt request. Writing a 0 has no effect on the request. The field is read as a 0 and the interrupt request is cleared by microcode at powerup.
PMON_RESET	28	WC,0	Writing a 1 to this field clears the **E_PMN%PMON_L** interrupt request. Writing a 0 has no effect on the request. The field is read as a 0 and the interrupt request is cleared by microcode at powerup.
H_ERR_RESET	29	WC,0	Writing a 1 to this field clears the **P%H_ERR_L** interrupt request. Writing a 0 has no effect on the request. The field is read as a 0 and the interrupt request is cleared by microcode at powerup.
PWRFL_RESET	30	WC,0	Writing a 1 to this field clears the **P%PWRFL_L** interrupt request. Writing a 0 has no effect on the request. The field is read as a 0 and the interrupt request is cleared by microcode at powerup.
HALT_RESET	31	WC,0	Writing a 1 to this field clears the **P%HALT_L** interrupt request. Writing a 0 has no effect on the request. The field is read as a 0 and the interrupt request is cleared by microcode at powerup.

DESIGN CONSTRAINT

When read onto **E_BUS%ABUS_L**, INT.SYS<31:27,24> must be zero. Microcode updates the internal copy of ICCS<6> and SISR<15:1> by reading the INT.SYS register,modifying the appropriate bits, and writing the updated value back. The write-one-to-clear bits must be read as zero because the microcode does not mask them out before writing them back.

MICROCODE RESTRICTION

The INT.SYS register is not bypassed. A write to INT.SYS in microinstruction *n* must not be followed by a read of INT.SYS sooner than microinstruction *n+4*.

MICROCODE RESTRICTION

Changes to machine state that affect the generation of interrupts (PSL<IPL>, ICCS<6>, or SISR<15:1>) done by microinstruction *n* must not be followed by a LAST CYCLE microinstruction sooner than microinstruction *n+4* if the change is to be observed by the next macroinstruction.

10.5 Processor Register Interface

Software can interact with the interrupt section hardware and microcode via references to processor registers, as follows:

- ICCS: References to the ICCS processor register allow access to the copy of ICCS<6> that is implemented in INT.SYS<0>, as described in Section 10.2.4.
- NICR, ICR: References to the NICR and ICR processor registers are transmitted off-chip to an optional full interval timer implementation as described in Section 10.2.4.
- SISR, SIRR: References to the architecturally-defined SISR and SIRR processor registers allow access to SISR<15:1>, which are implemented in INT.SYS<15:1>.

- ECR: References to ECR<ICCS_EXT> select the interval timer configuration, as described in Section 10.2.4.
- IAK14, IAK15, IAK16, IAK17: Reads of the IAK processor registers allow diagnostic and test software direct access to device interrupt vectors, as described in Section 10.2.2. References to these processor registers during normal system operation can result in UNDEFINED behavior.
- INTSYS: References to the INTSYS processor register allow diagnostic and test software direct access to the INT.SYS register. Reads of the INTSYS processor register return the format shown in Figure 10–3. Writes of the INTSYS processor register are internally masked by microcode such that only the left halt write-to-clear bits are written. Other bits remain unchanged. Writes to the INTSYS processor during normal system operation can result in UNDEFINED behavior.

10.6 Interrupt Section Interfaces

10.6.1 Ebox Interface

10.6.1.1 Signals From Ebox

- E_BUS%WBUS_L: Write data bus, from which ICCS<6> and SISR<15:1> are loaded, and from which the write-one-to-clear interrupt latches are cleared.
- E_PMN%PMON_L: Performance monitoring facility interrupt request.
- E_PSL%PSL_H<20:16>: IPL field from the current PSL.
- E_STL%F_NOP_S5_H: Force a NOP into S5 of the MIB decode pipe when an S3 or S4 stall exists
- E_STL%LATE_F_NOP_S4_H: Force a NOP into S4 of the MIB decode pipe when an S3 stall exists
- E_STL%LATE_STALL_S4_H: Stall the MIB decode pipe when an S4 stall exists

10.6.1.2 Signals To Ebox

- E_BUS%ABUS_L: A-port operand bus, on which ICCS<6>, SISR<15:1>, and the interrupt ID are returned.

10.6.2 Microsequencer Interface

10.6.2.1 Signals from Microsequencer

- E_USQ%MIB_H<31:20>: MIB lines used to decode the writes/reads to INT.SYS
- E_USQ%MIB_L<31:20>: MIB lines used to decode the writes/reads to INT.SYS
- E_USQ%UTSEL_H<4:0>: Microtest bus select code.
- E_USQ%UTSEL_L<4:0>: Microtest bus select code.

10.6.2.2 Signals To Microsequencer

- E%INT_REQ_H: Interrupt pending.
- E_BUS%UTEST_L<0>: Microtest bus.

10.6.3 Cbox Interface

10.6.3.1 Signals From Cbox

- C%CBOX_H_ERR_H: Hard error interrupt request.
- C%CBOX_S_ERR_H: Soft error interrupt request.

10.6.4 Ibox Interface

10.6.4.1 Signals From Ibox

- I%IBOX_S_ERR_L: Soft error interrupt request.

10.6.5 Mbox Interface

10.6.5.1 Signals From Mbox

- M%MBOX_S_ERROR_H: Soft error interrupt request.

10.6.6 Pin Interface

10.6.6.1 Input Pins

- P%HALT_L: Special-purpose halt "interrupt" signal, sampled by edge-sensitive logic.
- P%H_ERR_L: Special-purpose hard error interrupt signal, sampled by edge-sensitive logic.
- P%INT_TIM_L: Special-purpose interval timer interrupt signal, sampled by edge-sensitive logic.
- P%IRQ_L<3:0>: General-purpose interrupt signals, sampled by level-sensitive logic.
- P%PWRFL_L: Special-purpose power failure interrupt signal, sampled by edge-sensitive logic.
- P%S_ERR_L: Special-purpose soft error interrupt signal, sampled by edge-sensitive logic.

10.6.7 Signal Dictionary

Table 10–9: Cross-reference of all names appearing in the interrupt chapter

Schematic Name	Behavioral Model Name
C%CBOX_H_ERR_H	C%CBOX_H_ERR_H

Table 10–9 (Cont.): Cross-reference of all names appearing in the interrupt chapter

Schematic Name	Behavioral Model Name
C%CBOX_S_ERR_H	C%CBOX_S_ERR_H
E%INT_REQ_H	E%INT_REQ_H
E_BUS%ABUS_L<31:0>	E_BUS%ABUS_H
E_BUS%UTEST_L<0>	E_BUS%UTEST_H
E_BUS%WBUS_L<31:0>	E%WBUS_H
E_PMN%PMON_L	E_PMN%PMON_H
E_PSL%PSL_H<20:16>	E_PSL%PSL_H
E_STL%F_NOP_S5_H	E_STL%F_NOP_S5_H
E_STL%LATE_F_NOP_S4_H	E_STL%LATE_F_NOP_S4_H
E_STL%LATE_STALL_S4_H	E_STL%LATE_STALL_S4_H
E_USQ%MIB_H<31:20>	E_USQ%MIB_H
E_USQ%MIB_L<31:20>	E_USQ%MIB_H
E_USQ%UTSEL_H<4:0>	E_USQ%UTSEL_H
E_USQ%UTSEL_L<4:0>	E_USQ%UTSEL_H
I%IBOX_S_ERR_L	I%IBOX_S_ERR_L
M%MBOX_S_ERROR_H	M%MBOX_S_ERROR_H
P%HALT_L	P%HALT_L
P%H_ERR_L	P%H_ERR_L
P%INT_TIM_L	P%S_ERR_L
P%IRQ_L<3:0>	P%INT_TIM_L
P%PWRFL_L	P%PWRFL_L
P%S_ERR_L	P%S_ERR_L

10.7 Revision History

Table 10–10: Revision History

Who	When	Description of change
Mike Uhler	06-Mar-1989	Release for external review.
Mike Uhler	14-Dec-1989	Update for second-pass release.
Ron Preston	09-Jan-1990	Changes to simplify implementation.
Mike Uhler	20-Jul-1990	Update for change to performance monitoring interrupt request and reflect implementation.
Ron Preston	07-Feb-1991	Update to reflect Pass 1 implementation.

Chapter 11

The Fbox

11.1 Overview

This chapter describes the floating point unit of the NVAX CPU chip. Only the major functional blocks, their interfaces to each other, and the interface to the rest of the NVAX system are described here. Circuit level implementation details are not of primary concern in this document.

11.2 Introduction

The Fbox is the floating point unit in the NVAX CPU chip. The Fbox is a 4 stage pipelined floating point processor, with an additional stage devoted to assisting division. It interacts with three different segments of the main CPU pipeline, these are the micro-sequencer in S2 and the Ebox in S3 and S4. The Fbox runs semi-autonomously to the rest of the CPU chip and supports the following operations:

- **VAX Floating Point Instructions and Data Types**
  The Fbox provides instruction and data support for VAX floating point instructions. VAX F-, D-, and G-floating point data types are supported.
- **VAX Integer Instructions**
  The Fbox implements longword integer multiply instructions.
- **Pipelined Operation**
  Except for all the divide instructions, DIV{F,D,G}, the Fbox can start a new single precision floating point instruction every cycle and a double precision floating point or an integer multiply instruction every two cycles. The Ebox can supply two 32-bit operands or one 64-bit operand to the Fbox every cycle on two 32 bit input operand buses. The Fbox drives the result operand to the Ebox on a 32-bit result bus.
- **Conditional "Mini-Round" Operation**
  Result latency is conditionally reduced by one cycle for the most frequently used instructions. Stage 3 can perform a "mini-round" operation on the LSB's of the fraction for all ADD, SUB, and MUL floating instructions. If the "mini-round" operation does not fail, then stage 3 drives the result directly to the output, bypassing stage 4 and saving a cycle of latency.
- **Fault and Exception Handling**
  The Ebox coordinates the fault and exception handling with the Fbox. Any fault or exception condition received from the Ebox is retired in the proper order. If the Fbox receives or generates any fault or exception condition, it does not change the flow of instructions in progress within the Fbox pipe.

Figure 11–1 is a top level block diagram of the Fbox showing the six major functional blocks within the Fbox and their interconnections.

Figure 11–1: Fbox block diagram

```
F       Fraction Data
E       Exponent Data
S       Sign Data
C       Control
                        Control   Data Bus

                           | |       | |
                           | |       | |
                           \ /       \ /
                    ----------------------------------+
                    |  Interface - Input Section      |
                    +---------------------------------+

                      |F|     |E|     |S|     |C|
                      \ /     \ /     \ /     \ /
                    ----------------------------------
                    |            Divider              |
                    ----------------------------------
                      |F|     |E|     |S|     |C|
                      \ /     \ /     \ /     \ /
                    ----------------------------------
                    |            Stage 1              |
                    ----------------------------------
                      |F|     |E|     |S|     |C|
                      \ /     \ /     \ /     \ /
                    ----------------------------------
                    |            Stage 2              |
                    +---------------------------------+
                      |F|     |E|     |S|     |C|
                      \ /     \ /     \ /     \ /
                    ----------------------------------+
                    |            Stage 3              |
                    +---------------------------------+
                      |F| | | |E| | | |S| | | |C| | |
                      \ / | | \ / | | \ / | | \ / | |
                    +---------------------------------+
                    |            Stage 4              |
                    +---------------------------------+
                         |F|     |E|     |S|     |C|
                         \ /     \ /     \ /     \ /
                    +---------------------------------+
                    |  Interface - Output Section     |
                    +---------------------------------+
                           | |       | |
                           | |       | |
                           \ /       \ /
                          Control   Data Bus
```

11.3 Fbox Functional Overview

The Fbox is the floating point accelerator for the NVAX CPU. Its instruction repertoire includes all VAX base group floating point instructions. The data types that are supported are F, D, and G. Additional integer instructions that are supported are MULL2, and MULL3.

The number of internal execution cycles and the total number of cycles to complete an instruction within the Fbox is measured as follows in Figure 11–2

Figure 11–2: Fbox Execute Cycle Diagram

```
For L and F Data Types
       1        2        3       4        5        6        7

|----------|--------|--------|--------|--------|--------|--------|
|                        |   FS1  |  FS2   |  FS3   |   FS4  |
|<-opcode->|        |<-->|<----Fbox internal execute cycles--><->|
   cycle            |operand                                 result
                    |cycle                                   to Ebox

For D and G Data Types
        1        2        3       4        5        6        7        8          9

|----------|--------|--------|--------|--------|--------|--------|--------|--------|
|          |        |        |    | FS1    |  FS2   |   FS3  |  FS4   |
|<-opcode->|        |<-->|   |<-->|<----Fbox internal execute cycles->|<->|    <->|
   cycle             operand1 operand2                                 result   result
                     cycle    cycle                                    to Ebox  to Ebox
                                                                       LLW      ULW
```

The internal execution time for all instructions except MUL{D,G,L} and DIV{F,D,G} is four cycles. The internal execution time of the various Fbox operations is given in the following Table 11–1.

Table 11–1: Fbox Internal Execute Cycles

INSTRUCTION	F	D	G	L
MUL	4	5	5	5
DIV	14	25	24	-
ALL OTHER	4	4	4	4

The total number of cycles taken by the Fbox to complete an instruction is given in Table 11–2. Note that this includes the cycles taken for opcode and operand transfer, in particular, the dead cycle between the opcode and the first operand is counted.

Table 11–2: List of the Fbox Total Execute Cycles

INSTRUCTION	F	D	G	L
MUL	7	10	10	8
DIV	17	30	29	-
ALL OTHER	7	9	9	-

11.3.1 Fbox Interface

This section is responsible for overseeing the protocol with the Ebox. This includes the sequence of receiving the opcode, operands, exceptions, and other control information, and also outputing the result with its accompanying status. The opcode and operands are transferred from the input interface to stage 1 in all operations except division. The result is conditionally received from either stage 3 or stage 4.

11.3.2 Divider

The divider receives its inputs from the interface and drives its outputs to stage 1. It is used only to assist the divide operation, for which it computes the quotient and the remainder in a redundant format.

11.3.3 Stage 1

Stage 1 receives its inputs from either the interface or the divider section and drives its outputs to stage 2. It is primarily used for determining the difference between the exponents of the two operands, subtracting the fraction fields, performing the recoding of the multiplier and forming three times the multiplicand, and selecting the inputs to the first two rows of the multiplier array.

11.3.4 Stage 2

Stage 2 receives its inputs from stage 1 and drives its outputs to stage 3. Its primary uses are: right shifting (alignment), multiplying the fraction fields of the operands, and zero and leading one detection of the intermediate fraction results.

11.3.5 Stage 3

Stage 3 receives most of its inputs from stage 2 and drives its outputs to stage 4 or, conditionally, to the output. Its primary uses are: left shifting (normalization), and adding the fraction fields for the aligned operands or the redundant multiply array outputs. This stage can also perform a "mini-round" operation on the LSB's of the fraction for ADD, SUB, and MUL floating instructions. If the "mini-round" does not overflow, and if there are no possible exceptions, then stage 3 drives the result directly to the output, bypassing stage 4 and saving a cycle of latency.

11.3.6 Stage 4

Stage 4 receives its inputs from stage 3 and drives its outputs to the interface section. It is used for performing the terminal operations of the instruction such as rounding, exception detection (overflow, underflow, etc.), and determining the condition codes.

11.4 Fbox - Ebox Interface

The Fbox depends on the Ebox for the delivery of instruction opcodes and source operands and for the storing of results. However, the Fbox does not require any assistance from the the Ebox in executing the Fbox instructions. The Fbox macroinstructions are decoded by the Ibox just like any other macroinstruction and the Ebox is dispatched to an execution flow which transfers the source operands, fetched during S3 of the CPU pipeline, to the Fbox early in S4. Once all the operands are delivered, the Fbox executes the macroinstruction. Upon completion, the Fbox requests to transfer the results back to the Ebox. When the current retire queue entry in the Ebox indicates an Fbox result and the Fbox has requested a result transfer, then the result is transferred to the Ebox, late in S4 of the CPU pipeline, and the macroinstruction is retired in S5.

The Fbox input interface has two input operand registers which can hold all of the data for one instruction, and a three segment opcode pipeline. If the Fbox input machine is unable to handle new opcodes or operands then F%INPUT_STALL_H is asserted to the Ebox, causing the next Fbox data input operation to stall the CPU pipeline at the end of its S3.

The Fbox output interface has a format mux and two result queues, the data queue and the control queue. The format mux is used to transform the result data into VAX storage format. The queues are used to hold data results and control information whenever result transfers to the Ebox become stalled.

11.4.1 Opcode Transfers to the Fbox

Whenever the Fbox indicates that it is ready to receive new information by negating F%INPUT_STALL_H, the Ebox may initiate the next opcode or operand transfer. The Fbox receives instructions from the Microsequencer (S2 of the CPU pipeline) on a 9 bit opcode bus. The opcode bus is made up of the 8 msb's of the macroinstruction along with a single bit which, when set, indicates a G data type operation (i.e., the low order macroinstruction opcode byte was FD (hex)). The Micro-sequencer indicates the presence of a new opcode by asserting the opcode valid flag, E%FBOX_1ST_CYCLE_H. This opcode valid flag is only asserted once for each new instruction. In particular, if the Microsequencer was stalled during an opcode transfer cycle then the same opcode could be driven for multiple cycles, however, E%FBOX_1ST_CYCLE_H is only asserted for one of those stalled cycles. A complete list of the instructions executed by the Fbox and the opcode received from the Micro-sequencer is contained in Table 11–3.

NOTE

> The Fbox does not check for an illegal opcode. However, if an illegal opcode is received then the Fbox will interpret it as if it were an ADDF. No indication is given that this error has occured, the Fbox simply assumes that an ADDF has been started. When the instruction is retired (assuming that it actually was not an ADDF) it will be possible for diagnostic software to determine that an error has occured. This processing of illegal opcodes is done entirely to keep the Fbox internal control signals in a predictable state and thus avoid any "catastrophic" failure.

Once a valid opcode has been received from the Microsequencer, it is processed in a three element pipeline/queue by the Fbox input logic. The first level, I1, is a static register which feeds the re-code PLA. The second level, I2, is the recoded opcode. The third level, I3, is the current instruction, this register is output to both the Divider and Fbox stage 1. Any operand being sent to the Fbox is always for the instruction that is in I3. Each level has a corresponding valid bit which indicates the presence of an instruction to be executed. When the Fbox input is not stalled then opcodes and operands flow in the following order:

a. Opcode from the Microsequencer is loaded into I1 during Φ_4. (CPU S2)

b. Re-code PLA runs during the following Φ_{12}.

c. Re-coded opcode is loaded into I2 at the end of Φ_2.

d. I2 is loaded into I3 during the following Φ_3.

e. Input operand latches are loaded during the next Φ_{12}, at the earliest

f. Fbox internal Data Valid is set on Φ_3 following the last operand reception.

If the final data is not received during phase 12, then the I3 register stalls. This back pressures the Fbox input instruction pipeline, if there is a valid instruction in I2 then it will also stall. Once I2 is stalled, I1 will stall on the next instruction from the micro-sequencer. When the final operand for the instruction in I3 is received the stall is removed and new instructions are allowed to advance within the input pipeline.

Besides stalling when waiting for operands from the Ebox, the input instruction pipeline stalls for a fixed number of cycles during MUL{D,G,L} and DIV{F,D,G} instructions. These internally generated stalls, termed opcode stalls, are needed to allow multiple passes in the multiply and the divide arrays. The opcode stalls not only keep the Fbox input pipeline from advancing, but also cause **F%INPUT_STALL_H** to be asserted back to the Ebox.

Because an opcode stall can not be started until all the operands for the stalling opcode have been received, a three level instruction pipeline/queue is needed in the Fbox input stage (refer to Section 11.4.3, Figure 11–3). It is possible for the Fbox to receive two additional new opcodes before the opcode stall can be asserted and take effect at the Ebox. These two additional opcodes, along with the original stalling opcode, must be held in the Fbox input stage until the stall is finished.

11.4.2 Operand Transfers to the Fbox

Source operands, which were accesed in the Ebox during S3, are transferred from the Ebox to the Fbox early in S4. There will always be at least one cycle between the opcode transfer and the corresponding operands, during which the Fbox decodes the opcode. The data type of the source operand, contained in the I3 register of the input instruction pipeline, is used to select the proper data input format. There are two 32-bit input data busses, **E%ABUS_H** and **E%BBUS_H**, which are used to transfer operands to the Fbox. If the instruction is either a single operand type or, an integer or floating F type, then all of the operands are transferred in one cycle. If the instruction is a floating D or G type then one complete 64 bit operand is transferred on the concatenated input busses at a rate of one per cycle. For a floating D or G data type, the lower longword (i.e., sign, exponent, and fraction MSB's) is transferred on the **E%ABUS_H** and the upper longword is transferred on the **E%BBUS_H<>**.

Each 32-bit input operand bus has a related short literal flag which indicates the presence of a short literal on bits<5:0> of the corresponding bus. If a double precision operand is being transferred then a short literal will be detected using the flag associated with the **E%ABUS_H** and the floating short literal data will be taken from **E%ABUS_H<5:0>**. The remaining **E%ABUS_H** and **E%BBUS_H** bits are zero, however the Fbox ignores them. When receiving an integer short literal, the integer is on bits<5:0> and the Fbox depends on the remaining bits of that bus being zero. The Fbox must transform all short literals to the proper format based on the instruction data type.

When all of the input operand information for both input data busses is valid, the Ebox asserts an input valid flag, **E%FDATA_VALID_H**. If the flag is not asserted then the Fbox input machine enters an input stalled state.

Along with the operands, the Ebox sends 3 different operand fault flags. These are the memory management, hardware error, and reserved address mode faults. Once an operand fault has been sent to the Fbox, it is unpredictable whether the Ebox will or will not assert the **E%FDATA_VALID_H** signal. It is also unpredictable whether or not any other outstanding operands will be sent. When the Fbox receives an input fault two actions take place:

1. The Fbox asserts data valid into the Fbox pipeline. This breaks any internal stall conditions, thus allowing the instruction to complete.
2. The Fbox asserts **F%INPUT_STALL_H**. This halts the transfer of any other operands and prevents the Fbox and Ebox from getting out of synchronization. This stall normally continues until after the faulting instruction has been retired by the Fbox. It is cleared by the assertion of **E%FLUSH_FBOX_H** or **K%RESET_H**.

Since the faulting operand data values used by the Fbox are undetermined, it is possible that the Fbox may generate additional faults. However, the Ebox prioritizes the faults on retirement, the three input operand faults are at the highest priority. Therefore, any Fbox generated fault is ignored if the Fbox received an input operand fault. On completion, the faulting instruction will be handled by the Ebox in the proper order, ensuring compliance with the VAX architecture standard. In addition, the Ebox will flush the Fbox, this will cause **F%INPUT_STALL_H** to be negated, releasing the stalled state.

Besides the operand fault flags, the Ebox also sends the current value of the PSL floating underflow enable bit, **E%PSL_FU_H**. If the FU bit is set then the Fbox will cause a fault on floating underflow. Whether the FU bit is set or clear, the Fbox will return a floating zero data value on the result bus if underflow is detected.

11.4.3 Summary of Fbox Input Stage Stall Rules

The following list is a set of input stall rules for the Fbox input stage. They center around opcode transfers and the actions related to the assertion and negation of **F%INPUT_STALL_H**.

1. Floating opcodes are transferred from the Microsequencer to the Fbox during the CPU's S2 cycle. There will always be at least one cycle between an opcode transfer, OPC1, and the first data transfer for that opcode. In addition, there can only be one new opcode transfer, OPC2, between OPC1 and OPC1's last data transfer. It is possible that a new opcode transfer, OPC3, could take place in the same cycle as OPC1's last data transfer. Refer to the following Figure 11–3.

Figure 11–3: Opcode Transfers to the Fbox

```
Cycle |    n     | n+1      |   n+2    | |    m     |   m+1    | m+2      |
      +----------+----------+----------+...+----------+----------+----------+
      |   OPC1   |          | 1ST DATA|   |          |          |LAST DATA|
      +----------+----------+----------+...+----------+----------+----------+---...
                                           |   OPC2   |          |          |
                                           +----------+----------+----------+---...
                                                                 |   OPC3   |
                                                                 +----------+---...
```

2. Assertion of **F%INPUT_STALL_H** implies that the **next** data transfer cycle will stall; i.e., if **F%INPUT_STALL_H** is asserted during a data transfer cycle then that cycle will not stall but the next data transfer cycle will. That next data transfer cycle can **not** have either **E%FBOX_1ST_CYCLE_H** or **E%FDATA_VALID_H** asserted. The Ebox will repeat the

stalled transfer cycle keeping the **E%ABUS_H**, **E%BBUS_H**, **E%FDATA_VALID_H**, and any faults unchanged.

3. If **F%INPUT_STALL_H** is released in the current data transfer cycle then the current data transfer cycle will be repeated once more in the next cycle, this time with **E%FDATA_VALID_H** asserted. In that next cycle it is also possible to have **E%FBOX_1ST_CYCLE_H** asserted, indicating a new opcode transfer.

11.4.4 Fbox Result Transfers to the Ebox

Data is returned to the Ebox on one 32-bit output bus. A single integer or floating F type result can be returned in one cycle. Floating D/G data requires two cycles, the lower 32-bits (i.e., sign, exponent, and mantissa msb's) are returned in the first cycle followed by the upper 32-bits in the next cycle. A two bit data length field and a two bit condition code map field are also returned with each result transfer, as are all of the result status bits. The data length field is used to indicate a result data length of Byte, Word, Longword, or Quadword. The condition code map field informs the Ebox which PSL condition code bits must be updated for the retiring instruction. If the Fbox is not trying to retire an instruction then the condition code map is forced to a value of "no update". For double precision results which require two transfers, the data length is set to Quadword during both transfers. The condition code map will be forced to a value of "no update" during the first transfer of a double precision result and then to the proper instruction dependent code during the second transfer. The other result status is broadcast during both transfers. The Ebox uses the result status to detect microtrap conditions before any store of result data occurs.

The Fbox supplies 12 bits of status information with the retirement of each instruction. These are made up of:

a. Operand faults received with the input operands.
   1. **F%MMGT_FLT_H** - memory management faults
   2. **F%MERR_H** - hardware read faults, etc
   3. **F%RSVD_ADDR_MODE_H** - Reserved Address Mode Fault

b. Fault conditions detected by the Fbox
   1. **F%RSV_H** - reserved operand
   2. **F%FOV_H** - floating overflow
   3. **F%FU_H** - floating underflow
   4. **F%FDBZ_H** - floating divide by zero

c. Fbox condition code values
   1. **F%CC_N_H** - result is negative
   2. **F%CC_Z_H** - result is zero
   3. **F%CC_V_H** - result caused an integer overflow
   4. **F%CC_MAP_H<1:0>** - cc update map select

If multiple exceptions are detected by the Fbox for an instruction that it is executing then all of the exceptions for that instruction are reported to the Ebox. The Ebox and Microsequencer will prioritize these faults. The source operand faults are at the highest priority. Refer to Section 8.5.19.7 for the priority of the Fbox detected faults.

There are two signals from the Ebox to the Fbox that control the transfer of results by the Fbox. E%RETIRE_OK_H informs the Fbox that it may be possible to retire an instruction. E%STORE_OK_H indicates that it is possible for the Fbox to store data. When the Fbox wants to store a result the request signal, F%STORE_H is asserted. Similarly, if the Fbox wants to retire an instruction then F%RETIRE_H is asserted. All instructions must be retired on completion, most instructions (with the exception of TST and CMP) also need to store data. Single precision and integer instructions which store a result request to both store and retire in a single transfer cycle. Double precision instructions which store a result need two transfer cycles, the first transfer requests only to store, the second transfer requests to both store and retire. All TST and CMP instructions, regardless of data type, will request to retire without a store in one transfer cycle.

The completion of a result transfer from the Fbox to the Ebox is recognized when the appropriate request and its corresponding OK signal are both asserted. Conversely, if the corresponding OK signal is not asserted then the Fbox stalls (repeats) the current transfer.

When an instruction is completed by the Fbox core, the Fbox output stage transforms the data result back into VAX memory format. The VAX formatted data, along with ten bits of result status, is then always written into the output data queue. This queue has seven entries, each of which are 74 bits wide. The data from this queue is transferred to the Ebox on the F%RESULT_H bus in a first-in/first-out fashion, one longword at a time. If the data queue is empty at the time that the core is retiring, then the low word of the formatted data, along with the result status, is also selected to bypass directly to the result bus. This action is performed by the result multiplexer, which can select one of three sources: the queue bypass bus, the output queue low word, or the output queue high word.

The data queue is written every cycle, its input (write) pointer is only advanced after writing valid data. Whenever an instruction is retired, the data queue output (read) pointer is advanced. When the input and and output pointers are selecting the same entry then the queue is empty. If the input pointer is only one entry ahead of the output pointer a condition called empty next is detected. The empty and empty next conditions are used to generate result transfer requests from the data queue, and also in selecting between the queue bypass bus or the queue read data. Because double precision results retire from the Fbox core in one cycle but require two cycles to be transferred back to the Ebox, the high word of a double precision result will always be sourced from the data queue. This allows the core to retire quadword results in consecutive cycles (which could happen when CVTx{D,G} instructions are executing).

Besides the data queue, the Fbox output also has a control queue. This queue is seven bits wide by seven entries deep. It contains information derived from the opcode; the result data length, the condition code map, whether the instruction writes a result or not, and how many transfer cycles will be required to retire the instruction. Since the opcodes will precede the data through the Fbox pipeline by one cycle, there is no need to have a bypass bus for the control queue. The output machine is always able to write the control information into this queue and read it back before it is needed. Like the data queue, the control queue is written every cycle. Its input pointer is advanced after a new instruction has been passed through the pipeline and written into this queue. Its output pointer is advanced after a valid entry has been read into the control latch (i.e., the control queue's output latch). Because the request information is needed early in the transfer cycles, the control queue often is running ahead of the data queue.

Result transfers to the Ebox can be initiated by one of three sources: from the Fbox stage 3 bypass request line, from a data valid in Fbox stage 4, or from the Fbox output queue. The output queue takes precedence over the Fbox core. If the queue is not empty then the current queue output is transferred to the Ebox, any concurrent results from the Fbox core are written into the output queue. Fbox stages 3 and 4 perform their own prioritization. If stage 4 is retiring an instruction then stage 3 will not attempt to bypass stage 4. Instead, stage 3 passes its unrounded result to stage 4 and stage 4 will retire that result in the next cycle.

11.4.5 Fbox Pipeline Stalls

The Fbox input can request to stall the Ebox for one of two reasons. The Ebox does not actually stall until the next time it is ready to transfer data to the Fbox.

Fbox Input Stall

1. Opcode Stalls
2. Fault Stalls

As was mentioned earlier at the end of Section 11.4.1, the implementation of some instructions requires more than one cycle of execution within some stages of the Fbox pipeline. These instructions require that they be followed by a sufficient number of bubbles in the pipeline such that they can not be overrun by succeeding instructions. In particular, MUL{D,G,L} require two cycles in the stage 2 multiply array, and DIV{F,D,G} require 10,21,20 cycles, respectively, in the divide array. In order to guarantee proper operation, the Fbox input generates an input stall of the appropriate length for each of these instructions. The multiply stalls are controlled by a simple state machine in the Fbox input, it starts when all of the multiply operands have been received and continues for one cycle. The divide stalls are started by the input interface, as soon as all of the divide operands are received, and ended by a divide done signal which is received from the Fbox divider stage.

Whenever the Fbox receives an operand from the Ebox for which the Ebox has signaled a fault, the Fbox will request an input stall. This is done because it is unpredictable whether or not the Ebox will complete any other outstanding data transfers for this instruction. Therefore, to prevent the Fbox from entering an unpredictable state, F%INPUT_STALL_H is asserted and any new data transfers after the faulting source operand are blocked. When the instruction with the faulting operand is retired the Ebox will flush the Fbox, this will release the fault stall condition.

The Fbox output can cause a stall at the Ebox for one of two reasons:

Fbox Output Stall

1. Result not ready
2. Stage 4 bypass abort

If the Fbox does not have any results ready to retire and it is the selected source for the RMUX in the Ebox, then the Ebox is stalled until the Fbox is ready to transfer the result.

Stage 3 in the Fbox has the ability to perform "mini-round" operations for floating ADD, SUB, and MUL instructions. When stage 3 detects that it may be possible to round its fraction result and bypass stage 4, then it makes a request to store data to the Fbox output interface. If the data queue is empty then this store request is passed on to the Ebox. Later in the same transfer cycle, stage 3 may detect a "mini-round" overflow or some other error condition. If this occurs then stage 3 signals an abort of the stage 4 bypass. If the data queue was empty then this abort causes F%STORE_STALL_H to be asserted to the Ebox. The current store is stalled, by the Fbox, for one cycle until the correct result can be obtained from stage 4.

11.4.6 Fbox Reset and Flush

The Fbox can be initialized by the assertion of two different signals. At powerup time K%RESET_H is asserted for several cycles. This signal initializes all of the instruction registers and the output queue pointers in the Fbox interface. Any outstanding transfers and all stalls are terminated. At the completion of reset the Fbox is properly initialized and ready to receive opcodes and operands.

The Ebox can also initialize the Fbox by asserting the E%FLUSH_FBOX_H signal. This has the same effect as resetting the Fbox, the Fbox pipeline is cleared of all operations. Operations already under way anywhere in the pipeline are lost. E%FLUSH_FBOX_H is updated during phase 1 and it is only asserted for one cycle. The Fbox is ready to receive new opcodes in the very next cycle.

11.4.7 Summary of Fbox-Ebox Signals

The following signals are driven by the Ebox to the Fbox.

- E%FLUSH_FBOX_H
  This signal causes the Fbox to clear its pipeline of all operations.
- E%FBOX_1ST_CYCLE_H
  This signal tells the Fbox that the opcode is valid.
- E%FOPCODE_H<8:0>
  This 9-bit opcode bus carries the 8-bit opcode byte of the macroinstruction along with a single bit that indicates G-type data.
- E%FDATA_VALID_H
  This signal tells the Fbox that all data on the operand busses is valid. The Fbox knows, from decoding the opcode, exactly what data to expect.
- E%ABUS_H<31:0> and E%BBUS_H<31:0>
  These 32-bit busses carry the source operand(s).
- E%A_SHLIT_H and E%B_SHLIT_H
  These signals indicate that the data on the E%ABUS_H or the E%BBUS_H, respectively, is a 6-bit short literal value extracted from the instruction stream. Special data formatting is required by the Fbox.
- E%PSL_FU_H
  The current PSL<FU> value for use by the Fbox in deciding whether to signal floating point underflow faults or not.
- E%F_MMGT_FLT_H, E%F_MEM_ERR_H, and E%F_RSVD_ADDR_MODE_H
  These signals tell the Fbox that there is a fault or error associated with the source operands. The Fbox carries this status down the pipeline so that it is handled after instructions which are already in the Fbox pipeline.
- E%FBOX_S4_BYPASS_ENB_H
  This signal is used to control the Fbox stage 4 bypass option. Assertion of this signal enables stage 3 to conditionally bypass stage 4. This signal is normally cleared at system startup, disabling the bypass option. This signal has the additional function of selecting between FD1R or FD2R to be output of Stage3 while the FBOX is in FBOX_Test mode.
- E%RETIRE_OK_H, E%STORE_OK_H
  These signals inform the Fbox of any stalls when attempting to transfer a result to the Ebox.

The following signals are driven by the Fbox to the Ebox.

- **F%INPUT_STALL_H**
  This control signal stalls the Ebox from issuing any more operands to the Fbox.
- **F%RETIRE_H**
  This control signal tells the Ebox the Fbox is attempting to retire an instruction in this cycle.
- **F%STORE_H**
  This control signal tells the Ebox the Fbox is attempting to store a result in this cycle.
- **F%STORE_STALL_H**
  This control signal tells the Ebox the Fbox is stalling the current store request this cycle.
- **F%RESULT_H<31:00>**
  This 32-bit bus carries Fbox results to the Ebox.
- **F%FBOX_DL_H<1:0>**
  This is the data length used by the Ebox for an Fbox store.
- **F%CC_N_H, F%CC_Z_H, F%CC_V_H**
  These 3 signals carry Fbox condition code bits to the Ebox. They are Negative, Zero, and Overflow.
- **F%CC_MAP_H<1:0>**
  This is the map specifier which tells the Ebox how to update the PSL condition code bits.
- **F%MMGT_FLT_H**
  Signals a memory management fault for one of the currently retiring instruction's source operands.
- **F%MERR_H**
  Signals a memory access hardware error for one of the currently retiring instruction's source operands.
- **F%RSVD_ADDR_MODE_H**
  Signals a reserved address mode fault for one of the currently retiring instruction's source operands.
- **F%RSV_H**
  Signals a reserved operand fault for one of the currently retiring instruction's source operands.
- **F%FOV_H**
  Signals a floating point overflow fault resulted from the currently retiring instruction.
- **F%FU_H**
  Signals a floating point underflow fault resulted from the currently retiring instruction.
- **F%FDBZ_H**
  Signals a floating point divide-by-zero fault resulted from the currently retiring instruction.

11.4.8 Fbox Instruction Set

The instructions listed in Table 11–3 constitute the VAX integer and floating point instructions supported by the Fbox datapath.

Table 11–3: Fbox Floating Point and Integer Instructions

Fbox Opc	Instruction	NZVC	CC MAP	DL	Exceptions
04C	CVTBF src.rb, dst.wf	**00	10	10	
06C	CVTBD src.rb, dst.wd	**00	10	11	
14C	CVTBG src.rb, dst.wg	**00	10	11	
04D	CVTWF src.rw, dst.wf	**00	10	10	
06D	CVTWD src.rw, dst.wd	**00	10	11	
14D	CVTWG src.rw, dst.wg	**00	10	11	
04E	CVTLF src.rl, dst.wf	**00	10	10	
06E	CVTLD src.rl, dst.wd	**00	10	11	
14E	CVTLG src.rl, dst.wg	**00	10	11	
048	CVTFB src.rf, dst.wb	***0	11	00	rsv, iov
049	CVTFW src.rf, dst.ww	***0	11	01	rsv, iov
04A	CVTFL src.rf, dst.wl	***0	11	10	rsv, iov
068	CVTDB src.rd, dst.wb	***0	11	00	rsv, iov
069	CVTDW src.rd, dst.ww	***0	11	01	rsv, iov
06A	CVTDL src.rd, dst.wl	***0	11	10	rsv, iov
148	CVTGB src.rg, dst.wb	***0	11	00	rsv, iov
149	CVTGW src.rg, dst.ww	***0	11	01	rsv, iov
14A	CVTGL src.rg, dst.wl	***0	11	10	rsv, iov
04B	CVTRFL src.rf, dst.wl	***0	11	10	rsv, iov
06B	CVTRDL src.rd, dst.wl	***0	11	10	rsv, iov
14B	CVTRGL src.rg, dst.wl	***0	11	10	rsv, iov
056	CVTFD src.rf, dst.wd	**00	10	11	rsv
199	CVTFG src.rf, dst.wg	**00	10	11	rsv
076	CVTDF src.rd, dst.wf	**00	10	10	rsv, fov
133	CVTGF src.rg, dst.wf	**00	10	10	rsv, fov, fuv
040	ADDF2 add.rf, sum.mf	**00	10	10	rsv, fov, fuv
041	ADDF3 add1.rf, add2.rf, sum.wf	**00	10	10	rsv, fov, fuv
060	ADDD2 add.rd, sum.md	**00	10	11	rsv, fov, fuv
061	ADDD3 add1.rd, add2.rd, sum.wd	**00	10	11	rsv, fov, fuv
140	ADDG2 add.rg, sum.mg	**00	10	11	rsv, fov, fuv
141	ADDG3 add1.rg, add2.rg, sum.wg	**00	10	11	rsv, fov, fuv

Table 11–3 (Cont.): Fbox Floating Point and Integer Instructions

Fbox			CC		
Opc	Instruction	NZVC	MAP	DL	Exceptions
042	SUBF2 sub.rf, dif.mf	**00	10	10	rsv, fov, fuv
043	SUBF3 sub.rf, min.rf, dif.wf	**00	10	10	rsv, fov, fuv
062	SUBD2 sub.rd, dif.md	**00	10	11	rsv, fov, fuv
063	SUBD3 sub.rd, min.rd, dif.wd	**00	10	11	rsv, fov, fuv
142	SUBG2 sub.rg, dif.mg	**00	10	11	rsv, fov, fuv
143	SUBG3 sub.rg, min.rg, dif.wg	**00	10	11	rsv, fov, fuv
0C4	MULL2 mulr.rl, prod.ml	***0	11	10	iov
0C5	MULL3 mulr.rl, muld.rl, prod.wl	***0	11	10	iov
044	MULF2 mulr.rf, prod.mf	**00	10	10	rsv, fov, fuv
045	MULF3 mulr.rf, muld.rf, prod.wf	**00	10	10	rsv, fov, fuv
064	MULD2 mulr.rd, prod.md	**00	10	11	rsv, fov, fuv
065	MULD3 mulr.rd, muld.rd, prod.wd	**00	10	11	rsv, fov, fuv
144	MULG2 mulr.rg, prod.mg	**00	10	11	rsv, fov, fuv
145	MULG3 mulr.rg, muld.rg, prod.wg	**00	10	11	rsv, fov, fuv
046	DIVF2 divr.rf, quo.mf	**00	10	10	rsv, fov, fuv, fdvz
047	DIVF3 divr.rf, divd.rf, quo.wf	**00	10	10	rsv, fov, fuv, fdvz
066	DIVD2 divr.rd, quo.md	**00	10	11	rsv, fov, fuv, fdvz
067	DIVD3 divr.rd, divd.rd, quo.wd	**00	10	11	rsv, fov, fuv, fdvz
146	DIVG2 divr.rg, quo.mg	**00	10	11	rsv, fov, fuv, fdvz
147	DIVG3 divr.rg, divd.rg, quo.wg	**00	10	11	rsv, fov, fuv, fdvz
050	MOVF src.rf, dst.wf	**0-	01	10	rsv
070	MOVD src.rd, dst.wd	**0-	01	11	rsv
150	MOVG src.rg, dst.wg	**0-	01	11	rsv
052	MNEGF src.rf, dst.wf	**00	10	10	rsv
072	MNEGD src.rd, dst.wd	**00	10	11	rsv
152	MNEGG src.rg, dst.wg	**00	10	11	rsv
051	CMPF src1.rf, src2.rf	**00	10	xx	rsv

Table 11–3 (Cont.): Fbox Floating Point and Integer Instructions

Fbox Opc	Instruction	NZVC	CC MAP	DL	Exceptions
071	CMPD src1.rd, src2.rd	**00	10	xx	rsv
151	CMPG src1.rg, src2.rg	**00	10	xx	rsv
053	TSTF src.rf	**00	10	xx	rsv
073	TSTD src.rd	**00	10	xx	rsv
153	TSTG src.rg	**00	10	xx	rsv

CC_MAP: Condition Code Map

00 = No Update
01 = MOV Floating
10 = All Other Floating
11 = Integer

DL: Result Data Length

00 = Byte
01 = Word
10 = Long
11 = Quad

11.5 DIVIDER

11.5.1 Introduction

The divider stage in the Fbox performs the floating point divide operations. The inputs to the divider stage are the divisor and the dividend operands, source data type, opcode, data valid, and abort from the input interface section. The divider computes the quotient, and outputs to stage 1 of the pipeline: the quotient as two vectors, the final remainder, also as two vectors, and division done signals. The divider also supplies the division done signal to the input interface section. The input interface stalls after issuing a divide instruction and defers further issue of instructions to Divider/Stage1 until the division is completed in the divider.

The final quotient and the final remainder are computed in the pipe stages. The sign of the final remainder is used for correcting the quotient. This correction is done in stage-3 of the pipeline. The terminal operations for floating point divide (quotient overflow, rounding), and the detection of floating overflow, underflow, and reserved operand are done in the pipeline stages.

The execution time within the divider stage is data independent for divide instructions. The table below lists execution time within the Fbox for divide instructions.

Table 11–4: Total Fbox execute cycles for Divide operation

Instruction	Execution time in cycles
DIVF	17
DIVD	30
DIVG	29

The execution cycles are counted beginning with the cycle in which Fbox receives the divide opcode through the cycle in which Fbox retires the result to EBOX.

A typical cycle count for DIVD instruction would have 1 opcode transfer cycle, 1 dead cycle, 2 operand transfer cycles, 1 divide pla cycle, 20 divider array cycles (retires 60 bits of quotient), 1 cycle each through stage1, stage2 and stage3 and finally 2 cycles for the result transfer from stage4 (lower longword) and output interface (upper longword) for a total count of 30 Fbox cycles.

11.5.2 Overview

The divider uses the Radix-2 SRT division algorithm using the following recursive relation:

```
NPR = 2*(PR - q*D)   where

NPR is the new partial remainder,
 PR is the partial remainder,
 q  is the quotient and
 D  is the divisor.  (assumed to be normalized.)
```

The partial remainder is computed using carry save addition and the quotient is selected using an estimate of the partial remainder. The boundary conditions for the partial remainder and the estimated partial remainder are as follows:

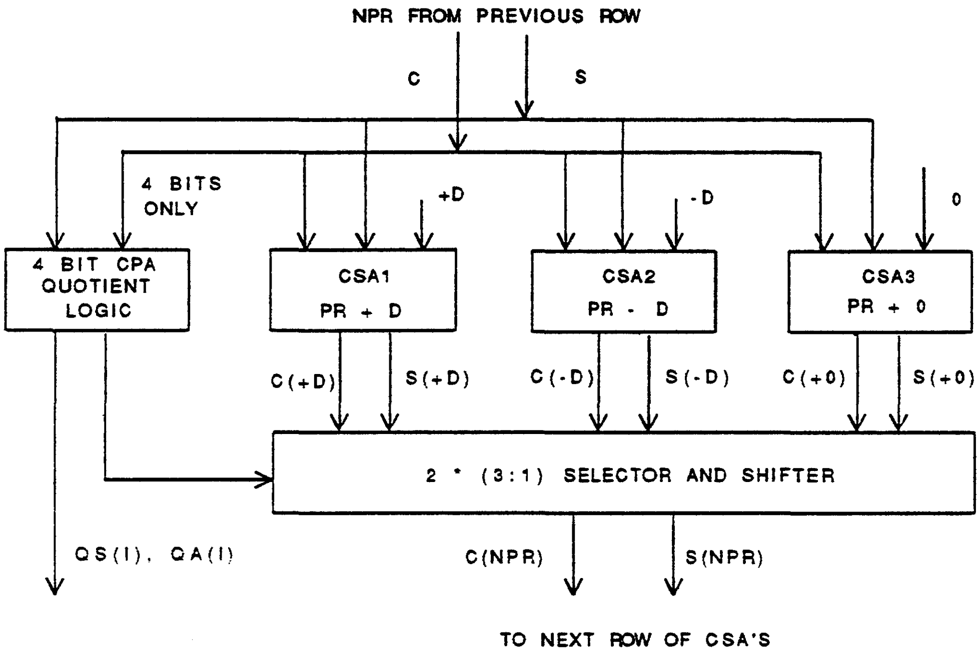
```
a.  -2D =< partial remainder < 2D
b.  0 =< Max. error < 1.0
c.  -2.5 =< estimated partial remainder < 2.0
d.  Quotient selection
     q = -1 if estimated partial remainder < (- 0.5)
     q =  0 if (- 0.5) =< estimated partial remainder < 0
     q = +1 if estimated partial remainder >= 0
```

To compute the estimated partial remainder the condition b) together with (c) above implies that a Carry Propagate Adder (CPA) of 4 bits (3 bits above the binary point and 1 bit below the binary point) is required.

The division process essentially consists of the following two steps to retire each bit:

- Compute estimated partial remainder using the CPA and the quotient
- Compute the new partial remainder using the CSA by adding +D, -D or 0 to the partial remainder based on the quotient from step 1.

Figure 11–4: Divider Array Block Diagram



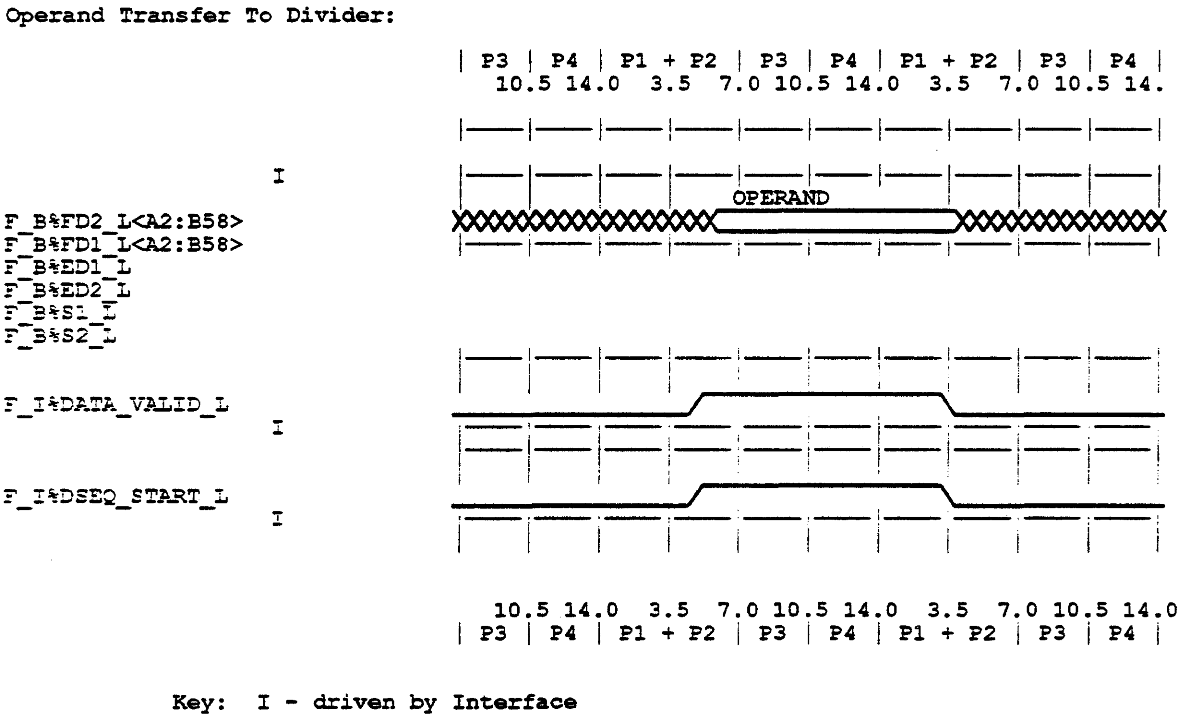
In order to speed up the time for retiring each bit, step 1 and step 2 are performed in parallel as there are only three choices for the quotient. As shown in the block diagram, Figure 11–4, the divider array computes (PR+1*D), (PR-1*D) and (PR+0*D) for all the possible values of quotient: q = -1, +1, and 0, in parallel while the quotient is being calculated. The correct new partial remainder is selected using the computed quotient. In the divide array, there are three rows of CSAs. Thus three bits are retired with each pass through the divide array.

11.6 Interface Signal Timing Diagrams

11.7 Divider Operation

For a valid divide operation, the divisor is loaded into Divisor (DVR) register and the dividend into Dividend Feedback (DFB) register, both during PHI_4. The CFB is initialized to zero. The control then sequences the datapath with appropriate control signals to load DFB and QM, for the required number of divide steps. For the DIVF instruction, the divide array generates 27 bits of quotient. For the DIVG instruction, the divide array produces 57 bits of quotient. For the

Figure 11–5: Input Signals from Input Interface

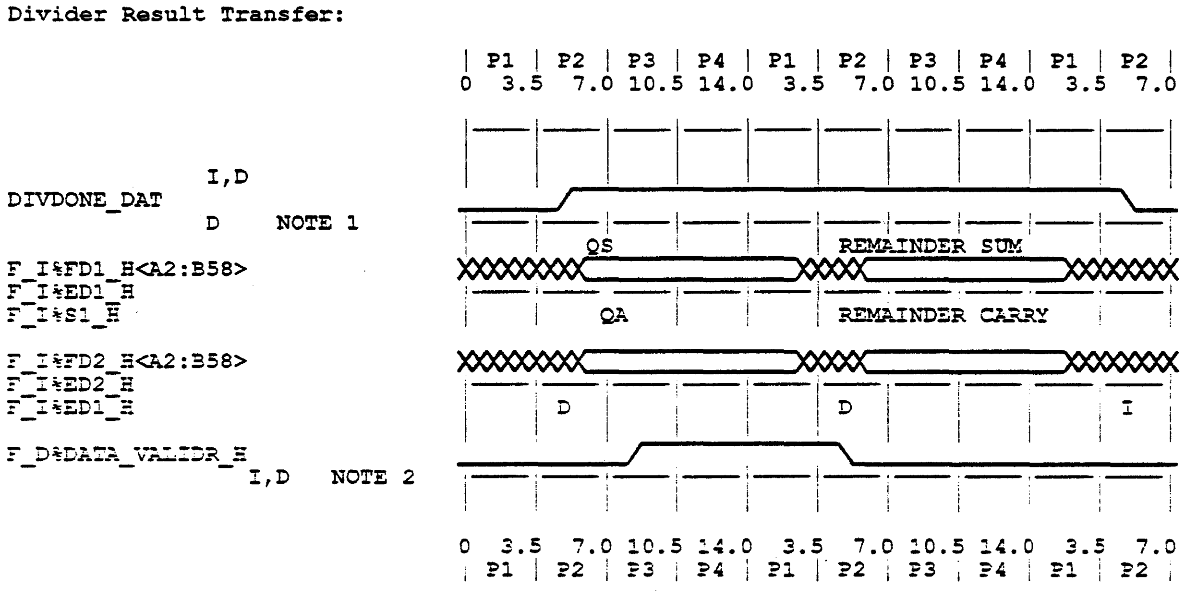


DIVD instruction, the divide array produces 60 bits of quotient. In general, since the quotient is greater than or equal to 0.5 and less than 2.0, the number of quotient bits to generate are the number of bits in the data type, one bit above the binary point and for rounding an additional bit in the least significant end. Since the divider array has three rows, one to two additional bits are generated.

The divider control receives the F_I%DSEQ_START_L signal from the input interface indicating a valid DIV instruction. This signal should remain valid from the trailing edge of PHI_2 (input to the Divider PLA) thru to the trailing edge of PHI_4 (Divisor and Dividend Latches) coming from the input interface. The divisor and dividend operand latches are conditioned by the F_I%DSEQ_START_L signal. The source data type field from the input interface determines whether the division is a DIVF or DIVD or DIVG.

At the conclusion of the required divider steps signals F_D_C2%DSEQ_DONEDAT4_H (to Input Interface) and F_D_C%DIVDONE_DAT_H (to Stage-1) are asserted. First the quotient components are driven on F_I%FD1R_H and F_I%FD2R_H together with the exponent and sign registers on respective buses. Then the sum and carry vectors are driven on F_I%FDR1_H and F_I%FD2R_H with exponents and signs.

Figure 11–6: Result Transfer to Stage-1



D - driven by Divider.

NOTE 1: divdone_dat with t_bypass_d deasserted.
NOTE 2: data valid only for quotient transfer.

The final quotient and the final remainder are computed in the pipeline stages. In stage 1, the two parts of the quotient and in the following cycle, the two parts of the remainder are added. The final quotient requires correction if the sign of the final remainder is negative as one too many subtractions were performed. Thus, if the sign of the final remainder is negative the quotient is decremented in stage 3. If the quotient is GEQ 1.0, it is shifted down and rounding constant is added in stage 4.

11.8 Divider Implementation

The divider stage consists of fraction data path, control, exponent and sign sections.

11.8.1 Divider Fraction Data Path

The divider fraction data path is composed of divisor register, divider array, quotient logic, quotient/remainder selector, and the fraction data path drivers. A block diagram of the divider fraction data path is shown in Figure 11–7. The divider fraction data path is shifted down by three bits relative to the interface and stage 1 fraction data path as shown in the figure.

11.8.1.1 Divisor Register - DVR

The divisor register DVR<B1:B55> stores the divisor from the interface for divide operations. The DVR register is loaded during PHI_4 when input interface asserts the F_I%DSEQ_START_L (asserted in PHI_2 and held by the divider through PHI_4) and the divider control asserts DVR_WR_FD1 (asserted in PHI_4). DVR<A2:A0> are forced to zero and DVR<B0> is forced to one. The output of this register is shifted down by two bits (for topological reasons to create space for Estimated Partial Remainder logic at the left of the datapath) and is used by the divider array to compute the partial remainder in DCSA cells. The dividend operand is also latched in PHI_4.

11.8.1.2 Divider Array

The divider array consists of three rows of carry save adders (CSA's), three carry propagate adders (CPAs), latches for the dividend and intermediate results. The various cells the divider array is composed of are DCSA, DSEL CPA, LAT1, R2D, DCSAF, DFB and CPA. The least significant bits of the array are different from the others and are described later.

11.8.1.2.1 DCSA and DSEL

The DCSA, the carry save adder cell computes in parallel the (partial remainder + divisor), (partial remainder - divisor) and (partial remainder + 0) corresponding to the quotient values of -1, 1, and 0 as sum (S) and carry (C). The correct new partial remainder is selected in DSEL using the three select lines from the CPA.

```
PR: Partial Remainder
S: sum input
C: carry input
D: divisor

S_PLUS0: sum output of PR+0       C_PLUS0: carry output of PR+0
S_PLUSD: sum output of PR+1*D     C_PLUSD: carry output of PR+1*D
S_MINUSD: sum output of PR-1*D    C_MINUSD: carry output of PR-1*D

SUM = S XOR C
SANDC_L = NOT(S AND C)
SORC_L  = NOT(S OR C)

S_PLUS0 = SUM
S_PLUSD = NOT((D AND SUM) OR (NOT D AND NOT SUM))
S_MINUSD = NOT(S_PLUSD)

C_PLUS0 = NOT(SANDC_L)
C_PLUSD = NOT((D AND SORC_L) OR (NOT D AND SANDC_L))
C_MINUSD = NOT((NOT D AND SORC_L) OR (D AND SANDC_L))
```

The inputs to the first row of the divider array are DVR, SFB_H, CFB_H. During the first step of the divide, the SFB and CFB contain the dividend and zero respectively and during subsequent steps they carry the outputs of the third row. The second row of the divider also uses the DCSA and DSEL cells.

In the least significant bits of the array, since the S vector is shifted left by 1 and C vector is shifted left by 2, except for the first step of the division, the S and C inputs to the DCSA are zero. For the first step of the division, the least significant bit contains dividend <B55>. For the computation of PR-1*D, the divisor is complemented and a one is forced in the C input position (to complete the 2's complement) as illustrated in Table 11–5.

Table 11–5: CSA Inputs

CSA Ports	PR+0	PR+1*D	PR-1*D
Input S	S	S	S
Input C	0	0	1
Input D	0	D	NOT D
Output S	S	S XOR D	S XOR D
Output C	0	S AND D	S OR (NOT D)

11.8.1.2.2 LAT1

The outputs of the first row are latched every cycle in the LAT1 cell to avoid corrupting the third row inputs. The LAT1 cell is also used to latch the select lines from the row 1 CPA in bit position <B56> for the formation of the quotient. The LAT1 outputs are shifted left - S by one and C by two, to form the 2*partial remainder for the second row DCSA. During reset, LAT1 is loaded with the row 1 outputs to prevent illegal data making multiple select lines valid in the second and third rows of the divider array.

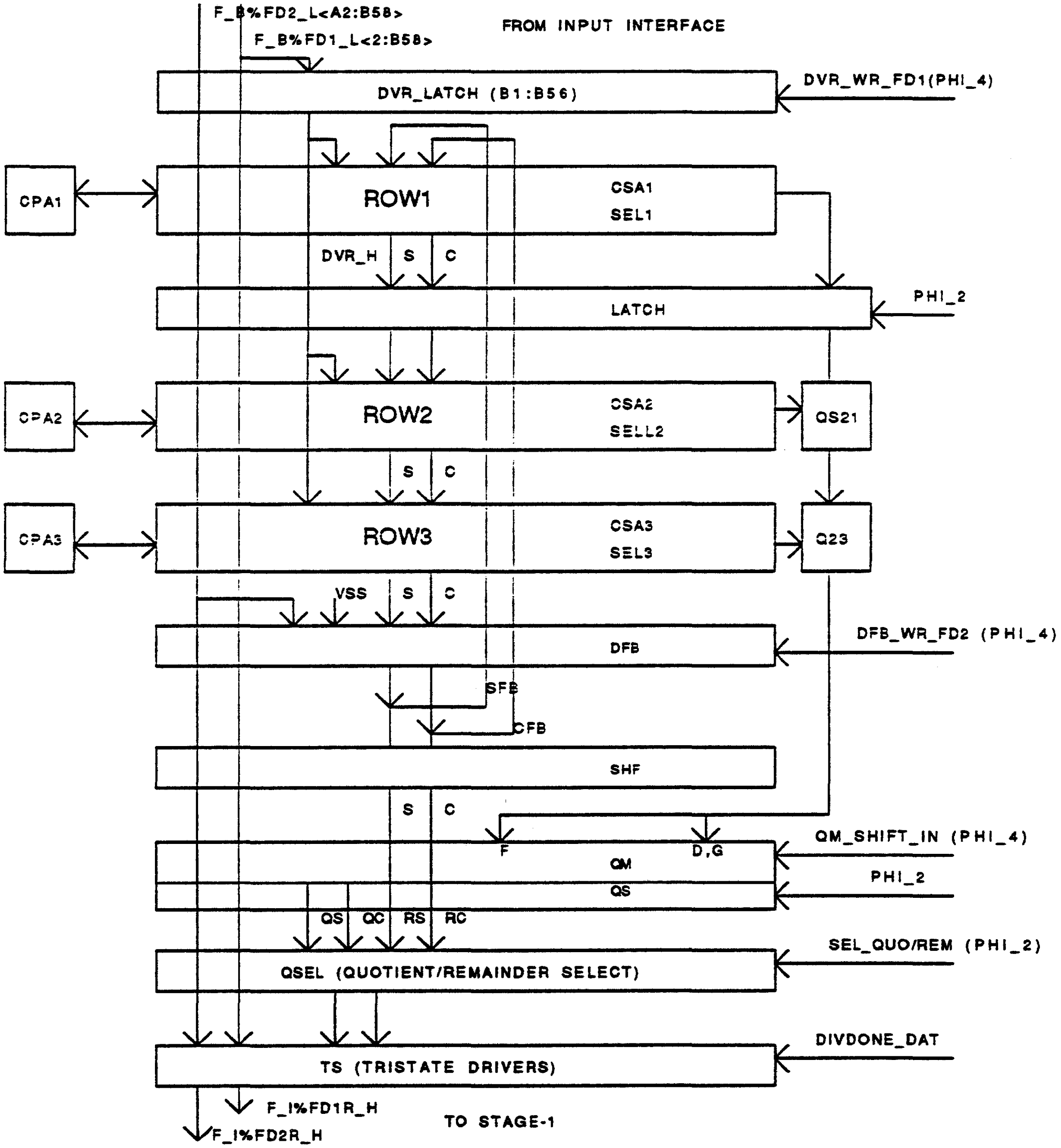
11.8.1.2.3 R2D and DCSAF

The cell R2D buffers the outputs of the second row and consequently the S and C vectors for the third row are asserted low. The cell DCSAF used for the third row is similar to the DCSA cell except that it takes S and C in complement form.

11.8.1.2.4 DFB and SHF

The DFB register contains static latches for the S and C outputs from the third row of the divider array and to store the dividend. The dividend is loaded into SFB from the input bus during PHI_4 using the control signal DFB_WR_FD2 and RESET_H, while the CFB is cleared. The S and C vectors from the third row are loaded into DFB using the control signal DFB_WR_R3 at the end of each pass through the array. The outputs of the DFB cell SFB_H and CFB_H are fed back to the first row of the array for the next pass. In addition, at the end of the required division steps, the DFB holds the final remainder to be transmitted to stage 1. The sign of the final remainder is used to correct the final quotient. Since the sign is derived from <A0> bit of the stage 1 adder, the final remainder is shifted down and buffered. The SHF cell accomplishes this and its outputs are RSR_L and RCR_L.

Figure 11–7: Divider Fraction Data Path



11.8.1.2.5 CPA

The CPA in each row of the divider array computes the estimated partial remainder(EPR) and generates the three select lines for selecting one of PR+0, PR+D and PR-D in the array. The inputs to the CPA are the four MSBs of S and C from the divider array. The CPA is implemented as a carry select adder as shown in Figure 11–8. The carry select adder computes the sign of the EPR, SIGN_H, and the zero detect logic detects if the 4-bit sum is exactly -0.5 (1111#2), Z_H. The three select lines are derived as follows:

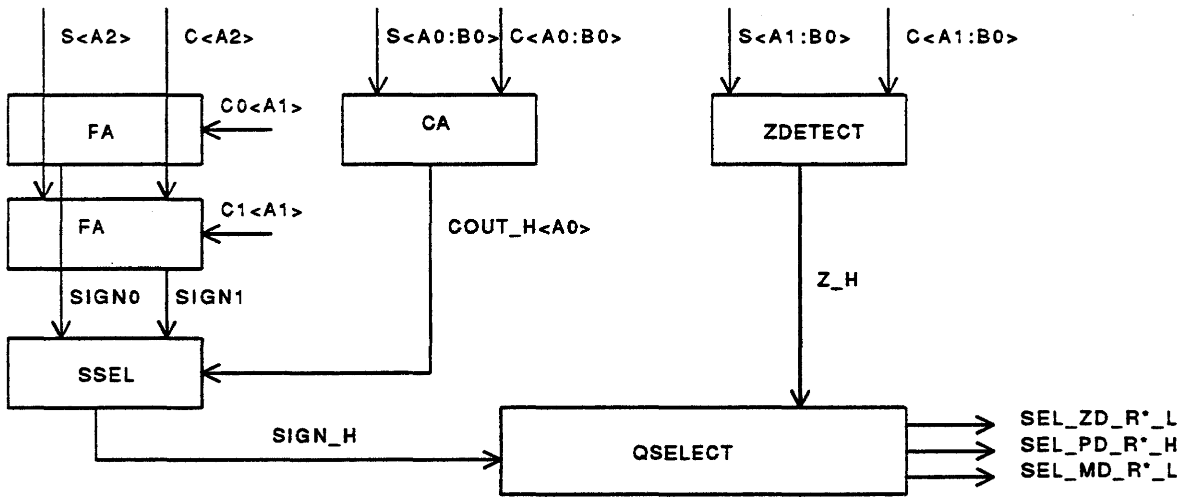
ESTIMATED PARTIAL REMAINDER	ACTION
011.1	SELECT PR+0 OUTPUT (NOT POSSIBLE)
010.0,010.1,011.0	SELECT PR-D OUTPUT (NOT POSSIBLE)
001.X	SELECT PR-D OUTPUT
000.X	SELECT PR-D OUTPUT
111.1	SELECT PR+0 OUTPUT
111.0	SELECT PR+D OUTPUT
110.X	SELECT PR+D OUTPUT
101.1	SELECT PR+D OUTPUT
101.0,100.X	SELECT PR+D OUTPUT (NOT POSSIBLE)

```
SEL_ZD_R*_H = select PR+0 output
SEL_PD_R*_H = select PR+D output
SEL_MD_R*_H = select PR-D output

SEL_ZD_R*_H = Z_H = SUM EQL X11.1
SEL_PD_R*_H = NOT Z_H AND SIGN_H
SEL_MD_R*_H = NOT SIGN_H
```

The three select lines are also used to form the quotient.

Figure 11–8: CPA Block Diagram



11.8.1.3 Quotient Recoding and Quotient Registers

11.8.1.3.1 QS21 and QREC

The select lines SEL_PD_R*, and SEL_MD_R* indicate the selected quotient value. Each pass through the array three pairs of quotient bits are generated. These can be expressed as the number of additions and the number of subtractions performed. These bits need to be accumulated in two shift registers. The final quotient is the total number of effective subtractions performed.

In order to minimize the number of bits to accumulate and to reduce the shift register bits, the three pairs of quotient bits from each pass through the divider array are encoded into four bits. The encoding is accomplished by generating the magnitude of the number of subtractions in each pass as three bits and a carry bit if the number of additions is greater than the number of subtractions. These four bits, instead of the six bits before the encoding, are accumulated in the shift register QM/QS. The carry vector after shifting left by 1 is subtracted from the number of effective subtractions to form the final quotient.

Since the row 3 computation is done last, two sets of quotient bits are generated from the first two rows - one for each possibility and the final quotient bits are selected based on the row 3 quotient bits. The cell QS21 performs recoding and generates the QSB21, QSB20, QSB10(QSB11) and QCA1 and QCA0.

```
S2 = subtract done in row 1
A2 = addition done in row 1
S1 = subtract done in row 2
A1 = addition done in row 2
S0 = subtract done in row 3
A0 = addition done in row 3

X2 = S2 XOR A2

QSB21 = NOT (X2 XOR S1)    QSB20 = X2 XOR A1
QSB11 = S1 XOR A1          QSB10 = NOT QSB11
QSB0  = S0 OR A0

QCA1  = A2 OR (NOT S2 AND NOT S1)
QCA0  = A2 OR (NOT S2 AND A1)
```

The cell QREC selects the final quotient bits and its outputs are QSUB_H<2:0> and QC_L<0> corresponding to the effective subtractions and the carry from one pass thru the array. These bits are shifted in to accumulate the final quotient in QM/QS cells.

11.8.1.3.2 QM and QS registers

The QM and QS is a master/slave shift register that holds the two components of the quotient - the number of subtractions performed and the carry vector respectively. After each pass through the array the quotient bits are loaded into QM at various positions depending on the data type. For the DIVF instruction the quotient bits are shifted into bit <B25>. For the DIVD instruction the quotient bits are shifted in at position <B58>. For the DIVG instruction the quotient bits are shifted in at position <B55>. The quotient carry component QC, is shifted left by one position when it is loaded into QM. The QM register is initialized to zero before beginning a new divide instruction so that the pipeline stages can operate on all the bits of the quotient. The QM register gets loaded either from the QSUB<2:0> and QC<0> or from the slave QS after a shift of three bits in PHI_4. The QS latch is loaded every PHI_2. The QM cells uses six control signals to clear, load or shift in the data. These control signals are derived as shown in Table 11–6.

Table 11–6: QM Cell Control Signals

	Operation					
Bit Positions	**INIT**	**DIVF**	**DIVD**	**DIVG**	**DONE**	**Cells**
A0:B22,B26:B52	CLEAR	SHFL	SHFL	SHFL	NOP	QMC,QMFC,QMGC
B23:B25	CLEAR	FLOAD	FSHF	FSHF	NOP	QMF
B53:B55	CLEAR	NOP	SHFD	GLOAD	NOP	QMG
B56:B58	CLEAR	NOP	DLOAD	NOP	NOP	QMD
Control Signals *						
DQM_SHFL	0	1	1	1	0	
DQM_FLD	0	1	0	0	0	
DQM_FSHF	0	0	1	1	0	
DQM_DLD	0	0	1	0	0	
DQM_GLD	0	0	0	1	0	
DQM_CLR	1	0	0	0	0	

* –asserted HIGH.
During reset, all the above control signals except DQM_CLR are deasserted.

In order to simplify the stage 1 control, the ones complement of the QC component is transferred to stage 1 so that stage 1 adder performs the same operation for both the final quotient and the final remainder computation.

11.8.1.3.3 QSEL and TSF

The QSEL selects the divider results to be driven to the stage 1 fraction data path. At the end of the required division steps, first the two components of the quotient are selected and in the following cycle the RSR and RCR of the final remainder are selected using the control signals DIV_SEL_REM_*. Since the carry component of the quotient is only one bit per three quotient bits, zeros are forced into the other two bits. The TSF cell consists of a tristate driver that drives the divider results on F_B%FD1_L and F_B%FD2_L busses during PHI_2 and PHI_3 using the control signal F_D_C2%DIVDONE_DATF_H. The TSF also contains buffers to drive F_I%FD1R_H and F_I%FD1R_H to stage 1.

11.8.2 Divider Control

The divider control is responsible for all sequencing and control of the divider data path. It gets F_I%DSEQ_START_L, SRC_DT_H<1:0> , F_I%DATA_VALIDR_H and F_I%ABORT_H from the Input Interface. The divider control generates all control signals for the data path, and F_D_C2%DSEQ_DONEDAT4_H signal for the input interface and F_D_C%DIVDONE_DAT_H to stage 1 of the pipeline. The early signal F_D_C2%DSEQ_DONEDAT4_H to the input interface stays valid thru two cycles for both the quotient and remainder transfers.

The F_I%DSEQ_START_L signal obtained from the input interface must be valid by the trailing edge of PHI_2. A latched version of this signal is used in PHI_4 to latch in the divisor and dividend.

11.8.2.1 Divider Control Blocks

The divider control consists of the control sequencer and miscellaneous logic, source data type latches, and buffers for driving the various control signals to the fraction data path.

11.8.2.1.1 Control Sequencer

The control sequencer is implemented as a PLA. The inputs to the PLA are the latched version of F_I%ABORT_H, F_I%DSEQ_START_L, F_I%SRC_DT_H<1:0> and state information. The PLA essentially implements a counter and a sequencer to control the data path. The divider control stays in the NOP state until a valid divide opcode and valid operands are received. The signal F_I%DSEQ_START_L obtained from the input interface combines these two conditions.
The state transition table shows the sequencer state, inputs and outputs.

Figure 11–9: Divider Sequencer State Transition Table

INPUTS				OUTPUTS								
ABORT	START_L	SRC_DT	STATE<4:0>	SEL_REM	BUSY	DONE_DAT	QSCR_EN	QCLR	STATE<4:0>	DONE_OPC	SEL_DOPC	DATA_VALID
1	X	X	X	0	0	0	0	1	NOP	0	0	0
0	1	X	NOP	0	0	0	0	1	NOP	0	0	0
0	0	X	NOP	0	1	0	1	0	PASS1	0	0	0
0	X	X	PASS1	0	1	0	1	0	PASS2	0	0	0
0	X	X	PASS2	0	1	0	1	0	PASS3	0	0	0
0	X	X	PASS3	0	1	0	1	0	PASS4	0	0	0
0	X	X	PASS4	0	1	0	1	0	PASS5	0	0	0
0	X	X	PASS5	0	1	0	1	0	PASS6	0	0	0
0	X	X	PASS6	0	1	0	1	0	PASS7	0	0	0
0	X	D/G	PASS7	0	1	0	1	0	PASS8	0	0	0
0	X	F	PASS7	0	1	0	1	0	LAST2	1	0	0
0	X	X	PASS8	0	1	0	1	0	PASS9	0	0	0
0	X	X	PASS9	0	1	0	1	0	PASS10	0	0	0
0	X	X	PASS10	0	1	0	1	0	PASS11	0	0	0
0	X	X	PASS11	0	1	0	1	0	PASS12	0	0	0
0	X	X	PASS12	0	1	0	1	0	PASS13	0	0	0
0	X	X	PASS13	0	1	0	1	0	PASS14	0	0	0
0	X	X	PASS14	0	1	0	1	0	PASS15	0	0	0
0	X	X	PASS15	0	1	0	1	0	PASS16	0	0	0
0	X	X	PASS16	0	1	0	1	0	PASS17	0	0	0
0	X	G	PASS17	0	1	0	1	0	LAST2	1	0	0
0	X	D	PASS17	0	1	0	1	0	PASS18	0	0	0
0	X	X	PASS18	0	1	0	1	0	LAST2	1	0	0
0	X	X	LAST2	0	1	0	1	0	SDONE	1	1	0
0	X	X	SDONE	0	1	1	0	0	QDONE	1	1	1
0	X	X	QDONE	1	1	1	0	0	RDONE	1	0	0
X	X	X	RDONE	0	0	0	0	1	NOP	0	0	0

The divider control PLA has 10 inputs, 14 outputs and 26 Minterms. These numbers include one spare input, one spare output and three spare minterms.

11.8.2.1.2 Opcode Information Latches

The divider latches the source data type signal from the input interface. If the divider is not busy, then the source data type information is latched into a static PHI_2 latch (cell LS). The output of this latch is used as an input to the sequencer PLA.

11.8.2.1.3 Divider Behavior during ABORT

Divider starts execution upon receipt of the F_I%DSEQ_START_L signal from the Input Interface. Assertion of F_I%ABORT_H from the Input Interface, while the divider is retiring quotient bits, will automatically force the divider to reset its control sequencer to its initial NOP state and to maintain the data valid enable in its deasserted state. It is expected that the Input Interface also deasserts the F_I%DATA_VALIDR_H signal during the ABORT cycle.

Assertion of the F_I%ABORT_H signal from the Input Interface during quotient and result transfers to Stage-1, also STOPS the divider from driving the F_D%DATA_VALIDR_H line to Stage-1. As above it is expected that the Input Interface also deasserts the F_I%DATA_VALIDR_H signal during the ABORT cycle.

11.8.2.1.4 Data path Control Drivers

The various control signals to the data path are combined with the appropriate clock signals and driven to the data path.

11.8.2.2 Summary of Divider Stage Outputs

The following table shows the divider stage outputs for the divide operations:

Table 11–7: Divider Output Stages

Instruction	Divider Outputs	
	Q(A,S)	R
DIVF	Q(C,S)<A0:B25>=Q	Remainder
	Q(C,S)<B26:B58>=0	
DIVD	Q(C,S)<A0:B58>=Q	Remainder
DIVG	Q(C,S)<A0:B55>=Q	Remainder
	Q(C,S)<B56:B58>=0	

Q(C,S)—Quotient Vectors QC, QS
R —— Remainder vectors, carry and sum

NOTE:

- The divider stage saves the exponent and the sign parts of the operands and passes them during the result transfer unchanged.
- Floating divide by zero, reserved operand, floating overflow and underflow are not detected by the divider stage. In these cases, the Q(C,S) and R outputs are undefined.

- The control outputs generated by the divider stage, DIVDONE_DAT_H and DSEQ_DONEDAT4_H signals are deasserted for non-divide operations.

11.8.2.3 Data Valid Logic

The divider output signal F_D%DATA_VALIDR_H driven to Stage-1 signal is a logical OR'ing of F_I%DATA_VALID_H signal from the Input Interface and the F_D_C_DV%EN_H signal from the Divider. These signals are mutually exclusive. The Input Interface deasserts its data valid after issuing a divide instruction and awaits the F_D_C2%DSEQ_DONEDAT4_H signal from the Divider before it asserts the data valid again. The presence of the global ABORT signal F_I%ABORT_H disables the driving of F_I%DATA_VALIDR_H signal by the Divider.

```
F_D%DATA_VALIDR_H = NOT (F_I%DATA_VALID_L AND F_D_C_DV%EN_L)
```

11.8.3 Exponent and Sign Data Path

The exponent data path in the divider consists of registers to save the exponents and signs of the divisor and the dividend. The divider does not operate on the exponent and sign parts of the divisor and the dividend. The exponents and signs are saved to pass them to stage 1 of the pipe along with the quotient and final remainder components so that for floating point divide operations, the exponent result and exception conditions can be detected.

The L1 cell is a static latch and is loaded with sign and exponent data from the interface during PHI_4 if a valid F_I%DSEQ_START_L is detected. At the end of divide operation the exponent and sign data is driven to stage 1 exponent data path. The cell TSE contains the tristate driver and the driver. The exponent and sign data, as in the case of the fraction data path, is actively driven during PHI_2 and PHI_3 using the control signal F_D_C2%DSEQ_DONEDAT_H and PHI_23.

11.9 Stage 1

Stage 1 of the pipeline is primarily used to perform the addition of the two inputs, or to compute the encoded shift amount, or to perform the recoding for the multiplier array, generate the initial partial product, select the row one input to the multiplier and the row two input to the multiplier in stage 2. Stage 1 receives its inputs from either the interface section or the divider section. All outputs of stage 1 are driven to stage 2 of the pipeline. The sign of the adder result is driven to stage 3 as well as stage 2. Stage 3 requires the sign of the remainder, for the divide operation, to determine if the quotient result should be incremented.

The fraction datapath portion of stage 1 primarily consists of an input selector, an adder, the multiplier recoder, and two output selectors. The adder in stage 1 is used for many functions. For multiply operations it is used to compute three times the multiplicand, for quotient operations it is used for adding the sum and carry vectors for the quotient; for other operations it is used to add two vectors.

The recoder logic is used to select the appropriate bits of the multiplier and recode them. The recoded bits are inputs to the multiplier array in stage 2.

The exponent datapath of stage 1 primarily consists of an input selector, two adders, detection logic, and an output selector. The main purpose of the exponent section in stage 1 is to compute the exponent difference. The detection logic is used to determine the range of the exponent difference.

The sign datapath portion in stage 1 performs no operation on the sign bits. They are passed unchanged to stage 2.

11.10 Section Implementation Description

11.10.1 Fraction Datapath

Figure 11–10 is a top level block diagram of the Fbox stage 1 fraction datapath.

Figure 11–10: Fraction Datapath Block Diagram

```
                                  F_I%FD2R   F_I%FD1R
                                      ||       ||       ZERO
                                      \/       \/        \/
+-----------------------------------------------------------------------------------------------------------+
|                                                 ISEL - IOVF                                               |---
+-------------------------------------++-------++-----------------------++--------++------------------------+
                                      ||       ||                       \/AIN     \/BIN
+-------------------------------------++-------++-----------------------------------------------------------+
|                                                  ADDER                                                    |
+----------------------++-------------++-------++-----------------------------------------------------------+
                       ||SUM          ||       ||FD1R
+----------------------++-------------++--------------------------------------------------------------------+
|                                        RECODER - MRECR - MTCR                                             |
+---++-------++--------++-------------++-------------------------------------------++-------------++--------+
    ||       ||        ||SUM          ||FD2R   ||FD1R                              ||             ||
+---++-------++--------++-------------++-------++----------------------------------++-------------++--------+
|                                       PHASE 4 LATCHES                                                     |
+---++-------++--------++-------------++-------++----------------------------------++-------------++--------+
    ||       ||        ||             ||       ||          ZERO                    ||             ||
    ||       ||        \/SUM          \/FD2_3R ||FD1_3R     \/                     \/MREC<17:0>    ||MRECR
+---++-------++--------++-------------++-------++----------------------------------++-------------++--------+
|                                        MIPPR SELECTOR and REGISTER                                        |
+---++-------++--------++-------------++-------++----------------------------------++-------------++--------+
    ||       ||        ||             ||       ||          ZERO                    ||             ||
    ||MIPPR  ||MTCR    \/SUM         \/ FD2_3R ||           \/                     \/MREC<17:0>    ||MRECR
+---++-------++--------++-------------++-------++----------------------------------++-------------++--------+
|                                        MRW2R SELECTOR and REGISTER                                        |
+---++-------++--------++-------------++-------++------------------------++--------++-------------++--------+
    ||       ||        ||             ||       ||          ZERO          ||        ||             ||
    ||MIPPR  ||MTCR    \/SUM         \/ FD2_3R ||           \/           ||MRW2R   \/MREC<17:0>    ||MRECR
+---++-------++--------++-------------++-------++------------------------++--------++-------------++--------+
|                                        MRW1R SELECTOR and REGISTER                                        |
+---++-------++--------++-------------++-------++------------------------++--------++-------------++--------+
    ||       ||        ||             ||       ||          ZERO          ||        ||             ||
    ||MIPPR  ||MTCR    \/SUM         \/ FD2_3R ||           \/           ||MRW2R   \/MRW1R        ||MRECR
+---++-------++--------++-------------++-------++------------------------++--------++-------------++--------+
|                                        FD1R SELECTOR and REGISTER                                         |
+---++-------++--------++-------------++-------++----------++------------++--------++-------------++--------+
    ||       ||       ZERO            ||       ||          ||            ||        ||             ||
    ||MIPPR  ||MTCR    \/            \/ FD2_3R ||          \/FD1R        ||MRW2R   \/MRW1R        ||MRECR
+---++-------++--------------------------------++------------------------++-----------------------++--------+
|                                        FD2R SELECTOR and REGISTER                                         |
+---++-------++--------------------------------++----------++------------++--------++-------------++--------+
    ||       ||                                ||          ||            ||        ||             ||
    \/       \/                                \/          \/            \/        \/             \/
  MIPPR_L  MTCR_L<18:0>                       FD2R_H       FD1R_H        MRW2R     MRW1R    MRECR_H[0:6]<5:0>
```

The Table 11–8 lists what is required to be loaded into the stage 1 fraction datapath registers, FD1R and FD2R, for each operation.

Table 11–8: Stage 1 Fraction Register Operations

Category	Operation	Condition
F0	FD1R <-- OP1 - (OP2) + 1	Effective SUB (DeltaE=0), CMP
F1	FD1R <-- OP1 -SHR1(OP2)+ 1	Effective SUB (DeltaE= +1)
F2	FD1R <-- -SHR1(OP1)+OP2+ 1	Effective SUB (DeltaE= -1)
F3	FD1R <-- OP1	Effective ADD or effective SUB (DeltaE > 1), and ED1R < ED2R
F4	FD1R <-- OP2	Effective ADD or effective SUB (DeltaE > 1), and ED1R >= ED2R
F5	FD1R <-- OP1 + OP2	DIV (after the divide array operation, done once for the quotient and one for the remainder) *
F6	FD1R <-- OP1 + 0	CVTfi, CVTff, MOV, MNEG, TST, and CVTif (if input integer is positive)
F7	FD1R <-- -(OP1) + 0 + 1	CVTif (if input integer is negative)
F8	FD1R <-- OP2 + SHL1(OP2)	MUL, MULL
F9	FD2R <-- OP1	Effective ADD or effective SUB (DeltaE > 1), and ED1R >= ED2R
F10	FD2R <-- OP2	Effective ADD or effective SUB (DeltaE > 1), and ED1R < ED2R, or MUL, MULL

*—The divider supplies stage 1 with QA . This allows the stage 1 adder to perform the same operation on the quotient and the remainder inputs.

11.10.2 Integer Overflow - IOVF

The integer overflow logic in stage 1 is used to help facilitate the detection of an integer overflow condition during a CVTFI operation.

11.10.3 Input Selector - ISEL

ISEL consists of two 3 to 1 selectors. The inputs to the A selector are FD1R%I<bI>, FD1R%I<bI+1>, and FD2R%I<bI-1>. The inputs to the B selector are FD2R%I<bI>, FD2R%I<bI+1>, and zero. Both selectors can invert the selected input.

11.10.4 Adder

The adder uses two 61-bit inputs to derive a 62-bit result. The 61-bit inputs have two bits above the binary point and 59 bits below; the 62-bit result has an additional bit above the binary point.

The main carry acceleration technique used is carry select. The adder is broken up into nine small groups, with all but the least significant group having duplicate carry chains. These carry chains operate in parallel in the first half of the stage 1 cycle. Propagate and generate logic operates before the carry chains. These parts of the adder are fully static.

In second half of the cycle, the sum logic executes. Just as for the carry logic, there is duplicate sum logic for all groups except the least significant one. These carry out signals are used to select the correct sum values. These parts of the adder are also fully static. The carry in to bit position <B58> is set directly by the stage 1 control.

11.10.5 Recoder Selector - RSEL

RSEL is a 2 to 1 selector which selects either F_I%FD1R_H<a0:b28> or F_I%FD1R_H<b26:b55>. When F_1_C%MRW_UPPER_H is asserted bits <a0:b28> are selected, and when F_1_C%MRW_UPPER_H is deasserted bits <b26:b55> are selected.

11.10.6 SRECODER

The srecoder uses the radix 8 modified Booth algorithm to compute the recoded sign bits of the partial products. The srecoder receives F_I%FD1R_H<a0:b26> as an input and outputs 9 recoded sign bits, F_1_R%SREC_H<8:0>. If either F_1_E%E1Z_H or F_1_E%E2Z_H is asserted, the srecoder will force the outputs to a one. The recoded sign bit is asserted when the partial product is positive.

11.10.7 Multiplier Two's Complement Register - MTCR<18:0>

The MTCR register is a 19 bit 2 to 1 selector and register. When F_1_C%MRW_UPPER_H is asserted the A inputs to the selector are selected and when F_1_C%MRW_UPPER_H is deasserted the B inputs to the selector are selected. Bit zero of the A input is tied to VDD, bits <9:1> are driven by the SRECODER, and bits <18:10> are tied to VDD. Bits <9:0> of the B input are driven by the RECODER and bits <18:10> are driven by the SRECODER.

11.10.8 Recoder

There are 31 inputs to the the recoder: F_1_R%RSEL_H<29:0> and zero. The least significant bit of the recoder input is always zero. The recoder performs the recoding using the radix 8 modified Booth algorithm. The recoder generates 60 recoded bits. They are F_1_R%MREC_H<59:0>. Of the 60 bits, 18 are used in stage 1. F_1_R%MREC_H<5:0> are used to select the MIPP. F_1_R%MREC_H<11:6> are used to select the row one input to the multiplier array. F_1_R%MREC_H are used to select the row two input to the multiplier array. If either F_1_E%E1Z_H or F_1_E%E2Z_H is asserted, the recoder will force the recoder outputs to recode zero.

11.10.9 PHI_4 LATCHES

The PHI_4 latches are used to latch F_I%FD1R<a2:b58>, F_I%FD2R<a2:b58>, F_1_R%MREC_H<59:0>, and F_1_R%SREC_H<8:0>.

11.10.10 Recoder Register - MRECR[0:6]<5:0>

The MRECR register is a 42 bit register. The latch is written every cycle with the upper 42 bits of the recoder output (F_1_R%MREC_3R_H<59:18>). The output of the register is driven to stage 2 as MRECR[0:6]<5:0>.

11.10.11 Multiplier Initial Partial Product Selector and Register - MIPPR

The MIPP selector is a 1 of 5 selector. It uses the RECODER output bits F_1%MREC_3R_H<5:0> to select the initial partial product. The inputs to the selector are: plus/minus (1X, 2X, 3X, 4X) the multiplicand, and zero. The selected input will be latched at the end of stage 1 execute cycle.

11.10.12 Multiplier Row 1 Selector and Register - MRW1R

The MRW1R selector is a 1 of 5 selector. It uses the RECODER output bits F_1%MREC_3R_H<11:6> to select the row 1 input to the multiplier array. The inputs to the selector are: plus/minus (1X, 2X, 3X, 4X) the multiplicand, and zero. The selected input will be latched at the end of stage 1 execute cycle.

11.10.13 Multiplier Row 2 Selector and Register - MRW2R

The MRW2R selector is a 1 of 5 selector. It uses the RECODER output bits F_1%MREC_3R_H<17:12> to select the row 2 input to the multiplier array. The inputs to the selector are: plus/minus (1X, 2X, 3X, 4X) the multiplicand, and zero. The selected input will be latched at the end of stage 1 execute cycle.

11.10.14 Selector and Register - FD1R

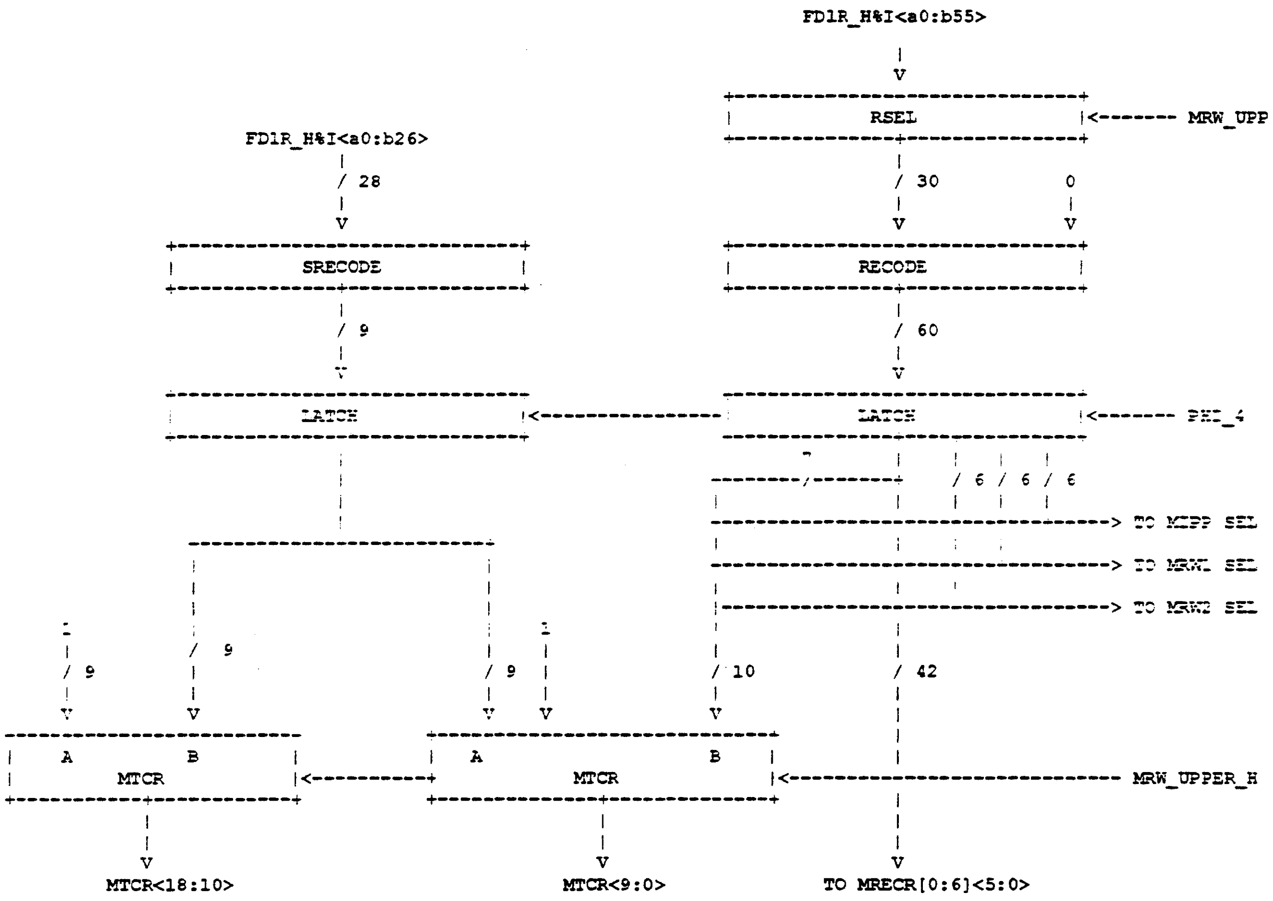
The FD1R selector is a 4 to 1 selector. The inputs to the selector are FD1_3R, FD2_3R, the output of the adder, and zero. The selected input is latched at the end of stage 1 execute cycle.

11.10.15 Selector and Register - FD2R

The FD2R selector is a 3 to 1 selector. The inputs to the selector are FD1_3R, FD2_3R, and zero. The selected input is latched at the end of stage 1 execute cycle.

Figure 11–11 is a block diagram of the recoder logic in stage 1 of the Fbox fraction datapath.

Figure 11–11: Recoder Block Diagram



11.11 Exponent Datapath

11.11.1 Stage 1 Exponent Processor Block diagram

Figure 11–12 is a block diagram of the exponent processor logic in stage 1.

Figure 11–12: Stage 1 Exponent Processor Block diagram

```
------------------------------------| |----| |-----------------------------------------| |---------------------
                       F_I_E%ED1R_H | |  K | |                                         | | F_I_E%ED2R_H    STN_EA1
                                    | |    | |                                         | |
                                    \ /    | |                                         \ /
                          +--------------------------------------------------------------------+
F_1_C%ENA_E1Z_L --------->|                          ZERO DETECTION                            |------> F_1_E%E1ZR_H
F_1_C%ENA_E2Z_L           +---------+-+----+-+-----------------------------------------+-+-----+       F_1_E%E2ZR_H
                       F_I_E%ED1R_H | |  K | |                                         | | F_I_E%ED2R_H
                                    \ /    \ /                                         \ /
                          +--------------------------------------------------------------------+
F_1_C%ISEL1_ED1R_A_H/L -->| AU                                                                 |
F_1_C%ISEL1_K_A_H/L ----->| NU                       INPUT SELECTOR                            |
F_1_C%ISEL1_ED2R_B_H/L -->|                           FOR  ADDER 1                             |
F_1_C%ISEL1_K_B_H/L ----->|                                                                    |
                          +---------+-+----+-+----------------------+-+--------+-+-----+-+-----+
                       F_I_E%ED1R_H | |  K | |               INP_1A | | INP_1B | |     | | F_I_E%ED2R_H
                                    | |    | |                      \ /        \ /     | |
                          +--------------------------------------------------------------------+
F_1_C%INVERT_EA_AD1_H/L ->|                         COMPLEMENT LOGIC                           |
F_1_C%INVERT_EB_AD1_H/L ->|                       FOR EXPONENT ADDER 1                         |
                          +---------+-+----+-+----------------------+-+--------+-+-----+-+-----+
                       F_I_E%ED1R_H | |  K | |               EA_AD1 | | EB_AD1 | |     | | F_I_E%ED2R_H
                                    | |    | |                      \ /        \ /     | |
                          +---------+-+----+-+-----------------------------------------+-+-----+
F_1_C%CIN_E_AD1_H  ------>|                         EXPONENT ADDER 1                           |
                          +---------+-+----+-+-----------+-+---------------------------+-+-----+
                       F_I_E%ED1R_H | |  K | |    E_AD1  | |                           | | F_I_E%ED2R_H     STN_EA1
------------------------------------\ /----\ /-----------\ /---------------------------\ /-------------------------
                          +--------------------------------------------------------------------+          F_1_E%E_N_H
F_1_C%ISEL2_ED2R_A_H/L -->|                                                                    |
F_1_C%ISEL2_K_A_H/L ----->|                         INPUT SELECTOR                             |
F_1_C%ISEL2_ED1R_B_H/L -->| AU                       FOR  ADDER 2                              |
F_1_C%ISEL2_K_B_H/L  ---->| NU                                                                 |
                          +---------+-+------------------+-+--------+-+--------+-+-----+-+-----+
                      F_I_E%ED1R_H  | |           E_AD1  | | INP_2A | | INP_2B | |     | | F_I_E%ED2R_H
                                    | |                  | |        \ /        \ /     | |
                          +---------+-+------------------+-+---------------------------+-+-----+
F_1_C%INVERT_EA_AD2_H/L ->|                       COMPLEMENT LOGIC FOR                         |
F_1_C%INVERT_EB_AD2_H/L ->|                         EXPONENT ADDER 2                           |
                          +---------+-+------------------+-+--------+-+--------+-+-----+-+-----+
                       F_I_E%ED1R_H | |           E_AD1  | | EA_AD2 | | EB_AD2 | |     | | F_I_E%ED2R_H
                                    | |                  | |        \ /        \ /     | |
                          +---------+-+------------------+-+---------------------------+-+-----+
F_1_C%CIN_E_AD2_H ------->| AU                      EXPONENT ADDER 2                           |
                          +---------+-+------------------+-+--------+-+----------------+-+-----+
                       F_I_E%ED1R_H | |           E_AD1  | |  E_AD2 | |                | | F_I_E%ED2R_H
                                    | |                  \ /        \ /                | |
                          +---------+-+-------------------------------------------------+-+-----+
                          |                        EXPONENT DIFFERENCE                         |----> F_1_E%E_DIFF_5_1_NEQ_0
                          |                               LOGIC                                |----> F_1_E%E_DIFFR_H<5:0>
                          +---------+-+------------------+-+--------+-+----------------+-+-----+
                       F_I_E%ED1R_H | |           E_AD1  | |  E_AD2 | |                | | F_I_E%ED2R_H            1_EA2
------------------------------------| |------------------| |--------| |----------------| |-----------------------------
                                    | |                  \ /        \ /                | |                         1_EOS
```

Figure 11–12 Cont'd on next page

Figure 11–12 (Cont.): Stage 1 Exponent Processor Block diagram

```
                           +-----------+-+-----------------------------------------------------+-+-----+
                           |                                                                           |----> F_I_E%E_DIFF_
                           |                            EXPONENT DIFFERENCE                            |----> F_I_E%E_DIFF_
                           |                                                                           |----> F_I_E%E_DIFF_
                           |                              DETECTION LOGIC                              |----> F_I_E%E_DIFF_
                           |                                                                           |----> F_I_E%E_DIFF_
                           +-----------+-+-------------------+-+-------------------------------+-+-----+
                          F_I_E%ED1R_H | |             E_AD1 | |                               | | F_I_E%ED2R_H
                                       \ /                   \ /                               \ /
                           +---------------------------------------------------------------------------+
F_1_C%OSEL1_ZERO_L ------->|                                                                           |
F_1_C%OSEL1_E_AD1_H/L --->|                       PHI_4 LATCHES, OUTPUT SELECT                         |
F_1_C%OSEL1_ED1R_H/L ---->|                           AND PHI_2 LATCHES                                |
F_1_C%OSEL1_ED2R_H/L ---->|                                                                            |
                           +-----------+-+-------------------------------------------------------------+           1_EOS
---------------------------------------| |-------------------------------------------------------------------------------
                           F_1_E%ED1R_H \ /
```

11.11.2 Exponent Adders

The operations performed by the adders in stage 1 are listed in Table 11–9. E1 refers to F_I_E%ED1R_H, E2 refers to F_I_E%ED2R_H, and K refers to the constants generated in the control section of stage 1.

Table 11–9: Exponent Adder Operations

Category	Adder #1	Adder #2	Condition
E0	—	—	CVTif, MULL
E1	E1 - E2	E2 - E1	SUBf, ADDf, CMPf
E2	- E1 + E2	—	DIVf
E3	- E1 + K	- K + E1	CVTfi
E4	E1 - K	—	CVTff, MOVf, MNEGf
E5	E1 + E2	—	MULf
E6	E1 + K	—	TSTf

11.11.3 Constants

The constants are driven from the control section into the exponent datapath. The constants needed for stage 1 are listed below.

0000010000000 = 0 ; TSTf
0000010000000 = 128 ; CVTff, MOV, MNEG {F,D}
0010000000000 = 1024 ; CVTff, MOV, MNEG {G}
0000010111000 = 184 ; CVTfi {F,D}
0010000111000 = 1080 ; CVTfi {G}

The Table 11–10 shows the required carry-in to the exponent adders.

Table 11–10: Exponent Adder Carry-In Operations

Category	Cin E_AD1	Cin E_AD2
E0	d	d
E1	1	1
E2	1	d
E3	1	1
E4	1	d
E5	0	d
E6	0	d

d = don't care

11.11.4 Zero Detection

The zero detectors check to see if an exponent operand has a value of zero. They are enabled by ENA_E1Z and ENA_E2Z. The detection is done in the second half of execute cycle and driven into the control as E1Z and E2Z. E1Z detects zero on ed1r and E2Z detects zero on ed2r.

11.11.5 Exponent Adder 1

The exponent adder is a 13-bit static adder used to add or subtract two inputs. Each input is passed through a 2 to 1 selector and inversion logic prior to the adder.

INP_1A can be selected from ED1R or K. If ISEL1_ED1R_A is asserted, then ED1R is passed through the selector. If ISEL1_K_A is asserted, then K is passed through the selector. Inversion of the adder input is then done based on the assertion of INVERT_EA_AD1.

INP_1B can be selected from ED2R or K. If ISEL1_ED2R_B is asserted, then ED2R is passed through the selector. If ISEL1_K_B is asserted, then K is passed through the selector. Inversion of the adder input is then done based on the assertion of INVERT_EB_AD1.

The adder also contains a carry-in to the LSB cell, CIN_E_AD1_H. The carry-in is primarily used for performing subtraction operations.

Since the adder is static, it begins its operation when the input data is valid at the start of the stage 1 execute cycle. Intermediate results in the exponent adder are latched in the second half of the execute cycle and sent to the detection logic and output selector.

11.11.6 Exponent Adder 2

Exponent adder 2 is almost identical to exponent adder 1. The only real difference is found in the input selection logic.

INP_2A can be selected from ED2R or K. If ISEL2_ED2R_A is asserted, then ED2R is passed through the selector. If ISEL2_K_A is asserted, then K is passed through the selector. Inversion of the adder input is then done based on the assertion of INVERT_EA_AD2.

INP_2B can be selected from ED1R or K. If ISEL2_ED1R_B is asserted, then ED1R is passed through the selector. If ISEL2_K_B is asserted, then K is passed through the selector. Inversion of the adder input is then done based on the assertion of INVERT_EB_AD2.

The adder also contains a carry-in to the LSB cell, CIN_E_AD2_H. The carry-in is primarily used for performing subtraction operations.

Since the adder is static, it begins its operation when the input data is valid at the start of the stage 1 execute cycle. Intermediate results in the exponent adder are latched in the second half of the execute cycle and sent to the detection logic and output selector.

11.11.7 Exponent Difference Detection

The exponent difference detection is used to detect certain exponent values. The detection is done on the output of both exponent adders, adder 1 and adder 2, and then selection of the exponent difference is based on E_N. E_N is bit 12 of adder 1. It is used to select the detection results from the positive adder output. The detection logic detects the following conditions:

Exponent Difference = 0 E_DIFF_EQL_0
Exponent Difference > 1 E_DIFF_GTR_1
Exponent Difference = 24 E_DIFF_EQL_24
Exponent Difference = 25 E_DIFF_EQL_25
Exponent Difference > 57 E_DIFF_GTR_57
E2 > E1 E_N
Exponent Difference <5:1> .NEQ. 0 E_DIFF_5_1_NEQ_0

The detection and latching is done at the start of the execute cycle.

In addition, the absolute value of the exponent difference is determined at the start of the stage 1 execute cycle. These lines, E_DIFFR<5:0>, are used to drive the inputs to the shift decoders in stage 2.

The exponent block also generates a signal called EDIFF_5_1_NEQ_0. This signal is asserted when bits <5:1> of the positive exponent difference are not equal to zero.

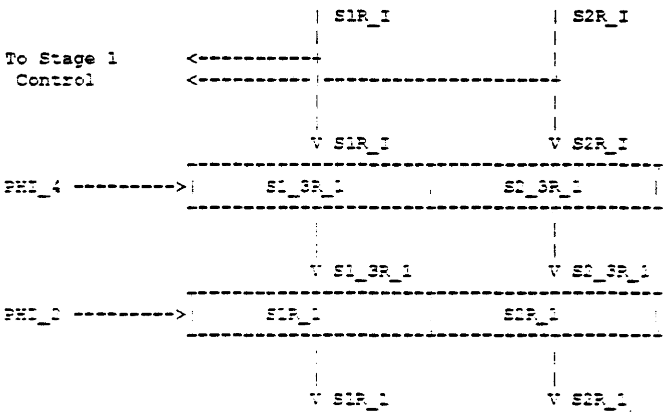
11.11.8 Output Selector

The output data (ED1R) can be selected from four sources: ed1r, ed2r, e_ad1 or it can be set to zero. The selection is done based on the assertion of the output select control signals. If OSEL1_ED1R is asserted, then ed1r is selected. If OSEL1_ED2R is asserted, then ed2r is selected. If OSEL1_E_AD1 is asserted, then e_ad1 is selected. If OSEL1_ZERO is asserted, then the output of the selector is zeroed. The output of the selector is latched every cycle at the end of the stage 1 execute cycle and driven into the following stage.

11.12 Sign Datapath

The sign bits of both the operands are not modified within stage 1. They are used by the stage 1 control. The two sign bits S1 and S2 are latched in stage 1 and are passed to stage 2 of the pipeline.

Figure 11–13: Sign Datapath Block Diagram



11.13 Stage 1 Control

The control section in stage 1 receives the opcode from the interface. The control section unconditionally decodes it every cycle. After a minimum of a one cycle delay, stage 1 will receive operands from the input interface. If it is a one operand instruction the input interface will assert data valid, and stage 1 will perform the instruction. If it is a two operand instruction, both operands are driven in the same cycle alongwith data valid.

11.13.1 Divide Instruction

During a divide operation, the opcode, data valid, and two operands are passed to the divider and stage 1 by the interface. The divider and stage 1 will perform their portion of the divide operation. Stage 1 will deassert data valid. When the divider completes the divide operation, stage 1 will again receive the opcode. The following cycle stage 1 will receive data valid, divdone_dat, and quotient bits QS and QA. Stage 1 will compute the quotient and pass data valid and the quotient result to stage 2. The next cycle stage 1 will receive divdone_dat and the sum and carry vectors for the remainder. Stage 1 will compute the remainder and pass the sign of the remainder to stage 3.

11.14 Fraction Datapath Operation Summary

Figure 11–14: Fraction Datapath Operation Table

	Condition					Data							
			EDIFF	IDIV									
OPERATION	E1Z	E2Z	5_1	ZERO	E_N	FD1R	FD2R	F_N	MRECR	MIPPR	MRWR	EDIFF	IOVF
EFF SUB EDIFF = 0	0	0	0	0	0	F1-F2	F1	0	X	X	X	X	X
EFF SUB DELTA E = +1	0	0	0	0	0	F1-F2/2	F1	0	X	X	X	X	X
EFF SUB DELTA E = -1	0	0	0	0	1	-F1/2+F2	F2	0	X	X	X	X	X
EFF ADD OR EFF SUB AND EDIFF > 1	0 0	0 0	1 1	0 0	0 1	F2 F1	F1 F2	X	X	X	X	V	X
CMP	0	0	X	0	V	F1-F2	0	X	X	X	X	X	X
MULf,d,g	0	0	X	0	X	F2+2F2	F2	0	V	V	V	X	X
DIVf,d,g	0	0	X	0	X	F1÷F2	0	V	X	X	X	X	X
MULL	0	0	X	0	X	F2+2F2	F2	V	V	V	V	X	X
MOV/MNEG	0	0	X	0	X	F1	0	V	X	X	X	X	X
TST	0	0	X	0	X	F1	0	V	X	X	X	X	X
CVTff	0	0	X	0	X	F1	0	0	X	X	X	X	X
CVTif S1 = 0	0	0	X	0	X	F1	0	0	X	X	X	X	X
CVTif S1 = 1	0	0	X	0	X	-F1	0	0	X	X	X	X	X
CVTfi	0	0	X	0	X	F1	0	0	X	X	X	V	V

X = Don't care
V = Valid

F1 = Fraction portion of operand 1
F2 = Fraction portion of operand 2

11.15 Fraction Datapath Exception Summary

Figure 11–15: Fraction Datapath Exception Summary

	Condition					Data							
OPERATION	E1Z	E2Z	EDIFF 5_1	IDIV ZERO	E_N	FD1R	FD2R	F_N	MRECR	MIPPR	MRWR	EDIFF	IOVF
EFF SUB	1	1	0	0	0	0	0	0	X	X	X	X	X
EDIFF = 0													
EFF SUB	1	0	X	0	1	0	F2	0	X	X	X	X	X
EDIFF = 1	0	1	X	0	0	0	F1	0	X	X	X	X	X
EFF ADD OR	1	0	X	0	1	0	F2	X	X	X	X	X	X
EFF SUB AND	0	1	X	0	0	0	F1	X	X	X	X	X	X
EDIFF > 1													
CMP	1	0	X	0	1	0	0	X	X	X	X	X	X
	0	1	X	0	0	0	0	X	X	X	X	X	X
	1	1	X	0	0	0	0	X	X	X	X	X	X
MULf,d,g	1	X	X	0	X	0	0	0	0	0	0	X	X
	X	1	X	0	X	0	0	0	0	0	0	X	X
DIVf,d,g	1	X	X	0	X	0	0	0	X	X	X	X	X
	X	1	X	0	X	0	0	0	X	X	X	X	X
MOV/MNEG	1	X	X	0	X	0	0	0	X	X	X	X	X
TST	1	X	X	0	X	0	0	0	X	X	X	X	X
CVTff	1	X	X	0	X	0	0	0	X	X	X	X	X
CVTfi	1	X	X	0	X	0	0	0	X	X	X	X	0

11.16 Exponent Datapath Operation Summary

Figure 11–16: Exponent Datapath Operation Table

	Condition					Data					
OPERATION	E1Z	E2Z	EDIFF 5_1	IDIV ZERO	E_N	ED1R	EDIFF = 0	EDIFF = 1	EDIFF > 1	EDIFF = 24	EDIFF = 25
EFF SUB EDIFF = 0	0	0	0	0	0	E1	1	0	0	0	0
EFF SUB DELTA E = +1	0	0	0	0	0	E1	0	1	0	0	0
EFF SUB DELTA E = -1	0	0	0	0	1	E2	0	1	0	0	0
EFF ADD OR EFF SUB AND EDIFF > 1	0 0	0 0	X X	0 0	0 1	E1 E2	0 0	0 0	1 1	X X	X X
CMP	0	0	X	0	V	0	V	X	X	X	X
MULf,d,g	0	0	X	0	X	E1+E2	X	X	X	X	X
DIVf,d,g	0	0	X	0	X	-E1+E2	X	X	X	X	X
MULL	0	0	X	0	X	0	X	X	X	X	X
MOV/MNEG	0	0	X	0	X	E1-K	X	X	X	X	X
TST	0	0	X	0	X	0	V	X	X	X	X
CVTff	0	0	X	0	X	E1-K	X	X	X	X	X
CVTif S1 = 0	0	0	X	0	X	0	X	X	X	X	X
CVTif S1 = 1	0	0	X	0	X	0	X	X	X	X	X
CVTfi	0	0	X	0	0	-E1+K	X	X	X	V	V

X = Don't care	E1 = Exponent portion of operand 1
V = Valid	E2 = Exponent portion of operand 2
K = Constant	MAX = Maximum exponent [E1,E2]

11.17 Exponent Datapath Exception Summary

Figure 11–17: Exponent Datapath Exception Table

	Condition					Data					
			EDIFF	IDIV			EDIFF	EDIFF	EDIFF	EDIFF	EDIFF
OPERATION	E1Z	E2Z	5_1	ZERO	E_N	ED1R	= 0	= 1	> 1	= 24	= 25
EFF SUB EDIFF = 0	1	1	0	0	0	0	1	0	0	0	0
EFF SUB DELTA E = +1	0	1	0	0	0	E1	0	1	0	0	0
EFF SUB DELTA E = -1	1	0	0	0	1	E2	0	1	0	0	0
EFF ADD OR	1	0	X	0	1	E2	0	0	1	X	X
EFF SUB AND	0	1	X	0	0	E1	0	0	1	X	X
EDIFF > 1											
CMP	1	0	X	0	1	0	0	X	X	X	X
	0	1	X	0	0	0	0	X	X	X	X
	1	1	X	0	0	0	1	0	0	0	0
MULf,d,g	1	X	X	0	X	0	X	X	X	X	X
	X	1	X	0	X	0	X	X	X	X	X
DIVf,d,g	1	X	X	0	X	0	X	X	X	X	X
	X	1	X	0	X	0	X	X	X	X	X
MOV/MNEG	1	X	X	0	X	0	X	X	X	X	X
TST	1	X	X	0	X	0	V	X	X	X	X
CVTff	1	X	X	0	X	0	X	X	X	X	X
CVTfi	1	X	X	0	1	0	X	X	X	0	0

Note - Stage 1 will not assert F_1%E2ZR_H during one operand instructions.

NOTE:

- The exponents and signs are driven to stage 1 during both quotient and final remainder transfers to stage 1.

11.17.1 Passthru Signals

MMGT_FLT_L, MEM_ERR_L, RSV_ADR_L and PSL_FU_H signals are simply passed through stage-1 without change. They are latched coming in from Input Interface during PHI_4 and driven to Stage-2 during PHI_2.

NEW_FOP_H signal also passes through to Stage-2 unaffected. It is latched during PHI_1 coming from Input Interface and driven to Stage-2 during PHI_3. This signal is gated with the global purge signal F_I%PURGE_H from the input interface which clears it on a PURGE from the input interface. This signal is used by the Output Interace to manipulate its control-queue and data-queue pointers.

11.18 STAGE 2

11.18.1 Introduction

Stage 2 of the Fbox pipeline is composed of a fraction datapath, an exponent datapath, a sign datapath and a control block. Stage 2 receives all its data inputs from stage 1 and passes all its data outputs to stage 3. Stage 2 receives control inputs from stage 1 and the interface section, and passes control information to stage 3. The stage 2 fraction datapath has an array multiplier, a right shifter and detection logic. The detection logic is used to detect the bit position of the most significant bit in a number and if a number is equal to zero. The detection logic is also used to generate the sticky bit associated with the right shifter. The exponent datapath is composed of the standard exponent block, of which only the adder and the output selector are used, and an additional 6 bit data register. The sign bits are passed from stage 1 to stage 3 unchanged.

The stage 2 fraction datapath performs operations on its input data for the following instructions: ADDf, SUBf, CMPf, TSTf, MULf, MULL, CVTif, CVTfi and CVTRfi. The ADDf and SUBf instructions use the output of the right shifter and the detection logic. CMPf, TSTf, CVTif use the output of the detection logic. The MULf and MULL instructions use the output of the array multiplier. The CVTfi and CVTRfi instructions use the output of the right shifter. For all other instructions the stage 2 fraction output registers are either written with the unchanged input data passed from stage 1 or the contents are undefined.

The stage 2 exponent datapath performs operations on its input data for the MULf and DIVf instructions. The adder in the exponent datapath is used to either add or subtract the appropriate exponent bias from the exponent data passed from stage 1. The output selector selects between the adder output, the input data from stage 1, and zero. For all instructions other than MULf and DIVf, the output selector passes the data passed from the stage 1 exponent datapath.

The stage 2 control block generates all conditional datapath control signals and passes control information to stage 3. The control block must sequence the fraction multiplier for MULD/G and MULL instructions which require two consecutive cycles of execution in the stage 2 fraction datapath for generating the two vectors (carry and sum) used for forming the final product in stage 3.

11.18.2 MUL Instruction Flows

Stage 2 is the stage of the Fbox pipeline that executes most of the computation needed for MULf and MULL instructions. To clarify the need for the multiply hardware in stage 2, the basic MUL flow is described. The multiplication algorithm implemented in the FBOX is the modified Booth algorithm which retires 3 multiplier bits at a time. The steps for calculating the product, or the fraction portion of the product in the case of floating point operands is as follows. First, the multiples of the multiplicand that are required by the Booth algorithm are calculated and the multiplier is recoded. Then the summands (a summand must be one of the calculated multiples of the multiplicand) which are to be added together to form the product are selected based on the recoded bits of the multiplier. Finally, the summands are added together and all terminal operations (rounding, etc.) are performed as required by the particular instruction and datatype.

The stage 1 fraction datapath basically calculates the required multiples of the multiplicand and recodes the multiplier. The stage 1 exponent datapath adds the exponents of the operands for floating point instructions. The sign of the operands are passed from stage 1 to stage 2 unchanged. The stage 2 fraction datapath selects the summands and performs carry-save addition on the summands. The stage 2 exponent datapath subtracts the appropriate exponent bias from the sum of the exponents calculated in stage 1. The signs of the operands are passed from stage 2 to stage 3 unchanged. The stage 3 fraction datapath forms the final product by doing a carry-propagate addition of the carry and sum vectors output from stage 2. The stage 3 exponent datapath decrements the exponent of the product if the fraction portion of the product needs to be normalized. Stage 3 also checks for potential data dependent stage-4 bypassable cases by carrying out a miniround on the lower 3 bits and the round bit. If the rounding operation doesn't carry past the 4 bits then stage-4 is bypassed. This bypass is aborted should stage-3 detect any exception or potential exception conditions. For a more detailed explanation refer to stage-3 specifiction.

For all non stage-4 bypassable instructions, Stage 3 passes the signs of the operands unchanged to stage 4. The stage 4 fraction datapath performs all terminal operations on the product. For MULf instructions stage 4 rounds the fraction of the product and increments the exponent if the fraction overflows. Stage 4 also checks for floating overflow and underflow. For MULL instructions stage 4 checks for integer overflow and forms and aligns the product for outputing to the interface. Stage 4 generates the correct sign bit for floating and integer MUL instructions.

Two consecutive cycles of execution in stage 2 are needed to complete all MUL instructions except MULF. This is due to the fact that 1 cycle is required for each pass through the multiply hardware in stage 2 and only F floating datatype multipliers can be completely retired in one pass. A more detailed description of the operations executed in the fraction datapaths of stages 1 through 4 is given below.

Stage 1 passes the recoded multiplier, +1*multiplicand and +3*multiplicand to stage 2. The multiples of the multiplicand required by the Booth algorithm are 0, +/-1*multiplicand, +/-2*multiplicand, +/-3*multiplicand and +/-4*multiplicand. Stage 1 only calculates +3*multiplicand because the other multiples are obtained by a simple shift of +1*multiplicand, and all negative multiples are generated by two's complementing the positive multiples. In order to reduce the number of computations executed in stage 2 for MULD, stage 1 also passes the summand selected from the recoded 3 LSB's (assuming D datatype) of the multiplier. The initial partial product is zero for all MUL instructions except for MULD. Stage 1 also passes two summands which are input to two rows of CS adders in stage 2 called MROW1 and MROW2, and a vector which facilitates generating the two's complement of the selected summands. The logic in stage 1 which determines the summands for MROW1 and MROW2 examine different multiplier bits depending on the operand datatype and whether it needs to output summands for the first or the second pass through the stage 2 multiply hardware. The initial partial product is latched in the MIPPR, and the two summand inputs to the MROW1 and the MROW2 are latched in the MRW1R and the MRW2R, respectively. The vector used in stage 2 for two's complementing the selected summands is latched in the MTCR.

Stage 2 selects all the summands which are needed to form the product, with the exception of the summands provided by stage 1. Stage 2 performs carry-save addition on the summands and outputs a carry and a sum vector to stage 3 for the formation of the final product. The multiply hardware in Stage 2 can be thought of as a 9 row, 3 bit retirement, carry-save multiply array which is capable of feeding its outputs back to its inputs for executing MULD/G and MULL instructions. Each multiply array cell is composed of a selector which selects a summand and a carry save adder which adds the summand to the partial product. The first two physical rows of the array are called MROW1 and MROW2, and are different from the other 7 rows in that they have no selector. The selected summands for MROW1 and MROW2 are passed from stage 1 in the MRW1R and MRW2R. During the first execute cycle of a MUL instruction, MROW1 adds the following three inputs from stage 1: the MIPPR output, the MRW1R output, and the MTCR output. During the second execute cycle, MROW1 adds the MRW1R output and the fed back MARRAY sum and carry outputs.

Stage 3 does the carry propagate addition on the carry and sum vectors passed from stage 2 to form the final product and normalizes the product if necessary. Note that a left shift of 1 bit position is the maximum normalization possible. Actually two separate carry propagate additions are performed in stage 3. A 60 bit carry propagate addition is performed to form the fraction portion of floating point products and the high order 58 bits of integer products. A separate 6 bit carry propagate addition is performed to form the 6 least significant bits of integer products. The carry out generated from the 6 bit addition is accounted for in the 60 bit addition so the 6 bit sum can be concatenated to the high order 58 bits. Stage 3 passes the results of both additions to stage 4.

Stage 4 performs all the terminal operations (rounding, etc.) on the final product (except when stage-3 bypasses stage-4 operations) before passing the product to the interface section. Stage 4 handles detection of floating underflow, floating overflow, integer overflow, and the proper alignment of the product.

11.19 Stage 2 Implementation Description

11.19.1 Fraction Datapath

Figure 11–18: Stage 2 Fraction Datapath Block Diagram

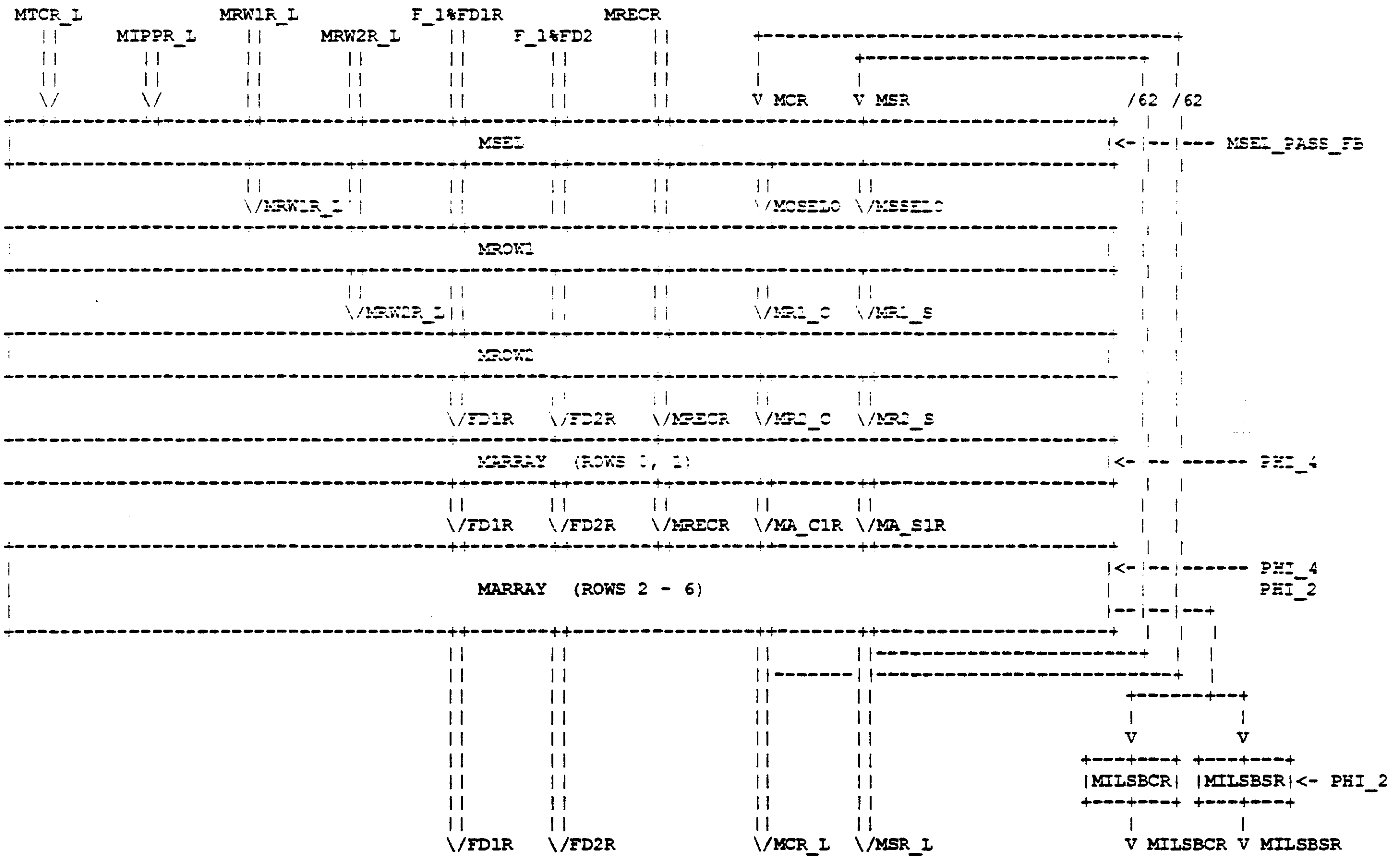


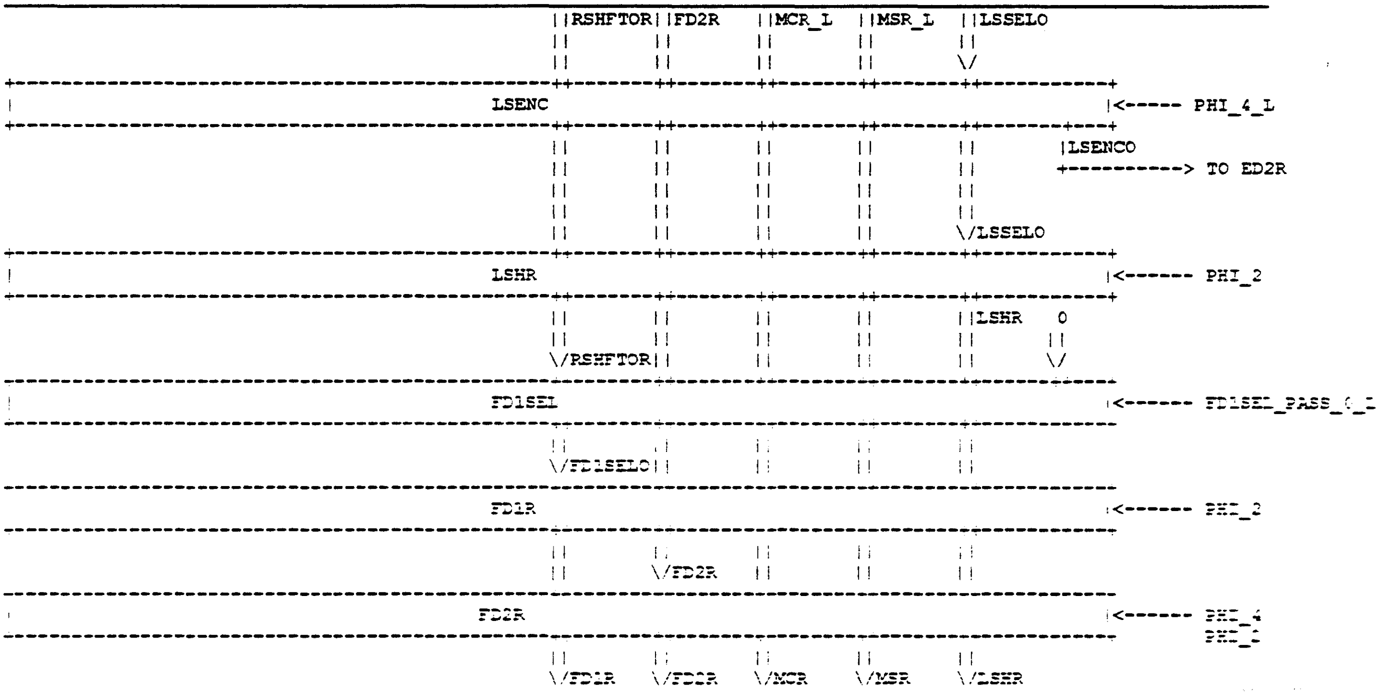
Figure 11–18 Cont'd on next page

Figure 11–18 (Cont.): Stage 2 Fraction Datapath Block Diagram

```
                                            ||FD1R              ||FD2R    ||MCR_L   ||MSR_L
                                            ||                  ||        ||        ||
                                            \/                  ||        ||        ||
+-------------------------------------------++------------------++--------++--------++-----------------------+        PHI_3
|                                              RSHIFT                                                        |<------ PHI_41
+-----------------------++------------------++--------++--------++--------++--------++-----------------------+        FORCE_SH
                        /\SDECO             ||        ||        ||        ||        ||
                        ||                  ||        \/RSHFTO  ||        ||        ||
+-----------------------++------------------++--------++--------++--------++--------++-----------------------+
|                                              RSHFTOR                                                       |<------ PHI_4
+-----------------------++------------------++--------++--------++--------++--------++-----------------------+
                        ||SDECO             ||        ||RSHFTOR||         ||        ||
                        ||                  ||        ||        ||        ||        ||
+-----------------------++------------------++--------++--------++--------++--------++-----------------------+
|                                              SDEC                                                          |<------ PHI_2
+-----------------------++------------------++--------++--------++--------++--------++-----------------------+
                        ||                  ||        ||        ||        ||        ||
                        \/SDECO             \/FD1R    ||        ||        ||        ||
--------------------------------------------------------------------------------------------------------------
|                                               DETL                                                         |------> F_Z, STK
|                                                                                                            |        PHI_3
|                                                                                                            |<------ DETL_EN_
--------------------------------------------------------------------------------------------------------------        SET_STKY
                        ||                            ||        ||        ||        ||        ||DETLO
                        ||                            ||        ||        ||        ||        \/
--------------------------------------------------------------------------------------------------------------
|                                               DETLOR                                                       |<------ PHI_41
--------------------------------------------------------------------------------------------------------------
                        ||                            ||        ||        ||        ||        ||DETLOR
                        ||                            ||        ||        ||        ||        \/
+-----------------------++----------------------------++--------++--------++--------++--------++-------------+
|                                               L1DETL                                                       |<------ ELZ_EDI
--------------------------------------------------------------------------------------------------------------
                        ||                            ||        ||        ||        ||        ||L1DETLO
                        \/SDECO                       ||        ||        ||        ||        ||
+-----------------------++----------------------------++--------++--------++--------++--------++-------------+
|                                              SDECOR                                                        |<------- PHI_4
+-----------------------++----------------------------++--------++--------++--------++--------++-------------+
                        ||                            ||        ||        ||        ||        ||
                        \/SDECOR                      ||        ||        ||        ||        \/L1DETLO
+-----------------------++----------------------------++--------++--------++--------++--------++-------------+
|                                               LSSEL                                                        |<------ LSSEL_PA
+-----------------------------------------------------++--------++--------++--------++--------++-------------+
                                                      ||        ||        ||        ||        ||
                                                      ||        ||        ||        ||        ||
                                                      \/RSHFTOR\/FD2R     \/MCR_L   \/MSR_L   \/LSSELO
```

Figure 11–18 Cont'd on next page

Figure 11–18 (Cont.): Stage 2 Fraction Datapath Block Diagram



11.19.2 MSEL - Multiplier Selector

The MSEL is composed of two 62 bit, 2 to 1 selectors. One selector selects the carry input to the MROW1, the other selects the sum input. The two possible carry inputs to the MROW1 are the F_1%MTCR_L and zero in some bit positions, or the MCR fed back from the bottom of the MARRAY. The two possible sum inputs to MROW1 are the F_1%MIPPR_L or the MSR fed back from the bottom of the MARRAY. If the signal MSEL_PASS_FB_H is asserted, then the MCR and MSR outputs are passed to the MSEL outputs, MCSELO and MSSELO, respectively. Otherwise the F_1%MTCR_L and the F_1%MIPPR_L are passed to MCSELO and MSSELO, respectively. The MSEL_PASS_FB_H signal is asserted during the second execute cycle of MULD/G and MULL. MCSELO<A2:B58> and MSSELO<A2:B58> are driven to the MROW1.

11.19.3 MROW1 - Multiplier Row 1

The MROW1 is composed of a row of 59 CS adders and a 3 bit carry propagate adder. The MROW1 is actually the first physical row of the multiplier array but since the summand selection is performed in stage 1, the MROW1 has no summand selector. The CS adders perform a carry-save addition on MRW1R_L<A2:B55> (the summand), MCSELO<A2:B55> and MSSELO<A2:B55>. The 3 bit carry propagate adder adds MCSELO<B56:B58> and MSSELO<B56:B58> and is needed to maintain a correct partial product. The 3 bit carry propagate adder insures that if the bits of the C and the S vector which are shifted out of the array cause a carry of bit significance

B55, that carry is correctly added to the partial product. The MROW1 generates a carry output, MRW1_C, and a sum output, MRW1_S, which are input to the MROW2.

11.19.4 MROW2 - Multiplier Row 2

The MROW2 is composed of a row of 59 CS adders and a 3 bit carry propagate adder. The MROW2 is the second physical row of the multiplier array, and like the MROW1, it has no summand selector. The summand selection for the MROW2 is done in stage 1. The CS adders perform a carry-save addition on MRW2R_L<A2:B55> (the summand), and sign extended MRW1_C<A2:B53> and MRW1_S<A2:B52>. The 3 bit carry propagate adder adds MRW1_C<B54:B55>, MRW1_S<B53:B55>, and the carry out of the 3 bit carry propagate adder in the MROW1, MRW1_C<B56>. This 3 bit carry propagate adder is needed to maintain a correct partial product. The 3 bit carry propagate adder insures that if the bits of the C and the S vector which are shifted out of the array cause a carry of bit significance B55, that carry is correctly added to the partial product. The MROW2 generates a carry output, MRW2_C, and a sum output, MRW2_S, which are input to the first row of the MARRAY.

11.19.5 MARRAY - Multiplier Array

The MARRAY is a 3 bit retirement per row multiplier array which has 7 rows of multiplier cells. The MROW1, MROW2, and the MARRAY are used together to generate a carry and a sum vector which are added in stage 3 to produce the final product. The inputs to MARRAY are F_1%FD1R, F_1%FD2R, MRECR, MRW2_C, and MRW2_S. The F_1%FD2R and F_1%FD1R contains 1*multiplicand and 3*multiplicand respectively for MUL instructions. The MRECR contains the recoded multiplier bits. The MRW2_C and MRW2_S signals are the carry and sum outputs of the MROW2. Each multiplier cell is composed of a selector and a CS adder. The selector selects the summand input and the CS adder adds the summand to the partial product. The MRECR[0:6]<5:0> control the summand selectors. The selector inputs are F_1%FD2R, F_1%FD2R left shifted by 1 bit position, F_1%FD1R, F_1%FD2R left shifted by 2 bit positions, or zero. The selector can generate the ones complement of any of the previously mentioned inputs for generating negative summands. The ones complement of zero is never generated. The MARRAY selector outputs are unconditionally latched in PHI_4. The least significant bit positions <B56:B58> in MARRAY, as in the MROW1 and MROW2, are populated by three bit carry propagate adder cells which are used to calculate carrys which have the weight of the <B55> bit position. The carry and sum outputs from the second row of the MARRAY, MA_C[1] and MA_S[1], are latched unconditionally in PHI_4. The least significant 5 carry and 6 sum outputs of the MARRAY cells in the fifth and sixth rows of the MARRAY are latched in the MILSBCR and MILSBSR. The MILSBCR and the MILSBSR are used in stage 3 to form the 6 least significant bits of longword products. The carry and sum outputs from the last row of the MARRAY, MA_C[6] and MA_S[6], are latched unconditionally in PHI_2. The latched versions of MA_C[6] and MA_S[6] are the MCR and MSR signals and are driven to the MSEL and stage 3.

11.19.6 MILSBSR<5:0> - Multiplier Integer LSB Sum Register

The MILSBSR is a 6 bit register. This register holds a 6 bit sum vector which is used to form the least significant 6 bits of the 64 bit product of longword operands. MILSBSR<5:3> are written with MA_S[5]<B53:B55>, and MILSBSR<2:0> are written with MA_S[4]<B53:B55> uncondtionally in PHI_2. The contents of this register are undefined for instructions other than MULL. The MILSBSR is driven to stage 3.

11.19.7 MILSBCR<4:0> - Multiplier Integer LSB Carry Register

The MILSBCR is a 5 bit register. This register holds a 5 bit carry vector which is used to form the least significant 6 bits of the 64 bit product of longword operands. MILSBCR<4:3> are written with MA_C[5]<B54:B55>, and MILSBCR<2:0> is written with MA_C[4]<B54:B56> uncondtionally in PHI_2. The contents of this register are undefined for instructions other than MULL. The MILSBCR output is driven to stage 3.

11.19.8 RSHIFT - Right Shifter

The RSHIFT shifts F_1%FD1R to the right by 0 to 57 bit positions depending on the control signal FORCE_SHFT_0 and the output of the shift decoder, SDECO. The RSHIFT is used for pre-aligning operands in ADD and SUB instructions under certain conditions (for details see the description of the stage 2 control) and right shifting the fraction of a floating point operand in CVTFI/CVTRFI instructions. If the signal FORCE_SHFT_0 is asserted, the RSHIFT will pass F_1%FD1R<A0:B58> to its output RSHFTO<A0:B58> unshifted. If FORCE_SHFT_0 is deasserted, F_1%FD1R is passed to RSHFTO right shifted by 0 to 57 bit positions, depending on the state of SDECO<57:0> which has exactly 1 bit asserted. F_1%FD1R<A0> is always passed to the RSHFT<A0> output. The RSHFTO<B0:B57> bits which become vacant due to the right shift of F_1%FD1R<B0:B58> are zero filled. The RSHIFT output, RSHFTO, is driven to the RSHFTOR.

11.19.9 RSHFTOR<A0:B58> - Right Shifter Output Register

The RSHFTOR is a 60 bit register which is written with RSHFTO<A0:B58> unconditionally in PHI_4. The RSHFTOR is driven to the FD1SEL.

11.19.10 SDEC - Shift Decoders

The SDEC decodes the F_1%E_DIFFR_H<5:0> from the stage 1 exponent datapath to a 58 bit output, SDECO<57:0>, which has exactly one bit asserted. The SDECO is the fully decoded right or left shift amount which is used to control the RSHIFT or the normalizer in stage 3, for ADD, SUB and CVTFI instructions under certain conditions (for details see the description of the stage 2 control). The assertion of SDECO<57> corresponds to a shift of zero. The assertion of SDECO<56> corresponds to a shift of 1, and so on. The SDEC output is driven to the RSHIFT, the SDECOR and the DETL.

11.19.11 SDECOR<57:0> - Shift Decoder Output Register

The SDECOR is a 58 bit register which is written with SDECO<57:0> unconditionally in PHI_4. The SDECOR is driven to the LSSEL.

11.19.12 DETL - Detection Logic

The DETL detects if F_1%FD1R is equal to zero, generates outputs which are used by the leading one detection logic, L1DETL, and calculates the sticky bit for the adder in stage 3. The sticky bit is needed for ADD and SUB instructions under certain conditions (for details see the description of the stage 2 control). If the control signal DETL_EN_STKY_L is asserted, the DETL takes F_1%FD1R and SDECO as its inputs and calculates the sticky bit, STKYR. The sticky bit is set if a one in the F_1%FD1R is right shifted out of the B58 bit position by the RSHIFT. If F_2%SET_STKYR_H is asserted, STKYR is set independent of the DETL inputs. The STKY latch is written unconditionally in PHI_2 and is driven to stage 3. If DETL_EN_STKY_L is deasserted, the DETL takes only the F_1%FD1R as its input and it generates outputs which are used by the L1DETL. The DETL has two outputs, FZ and DETLO<65:0>. The FZ is the zero detection output and is driven to the stage 2 control block. The DETLO<65:0> outputs are driven to the DETLOR. FZ is asserted if F_1%FD1R<B0:B57> are all zeros. The FZ output is conditionally loaded in PHI_2 in the stage 2 control.

11.19.13 DETLOR<B0:B57> - Detection Logic Output Register

The DETLOR is a 58 bit register which is written with DETLO unconditionally in PHI_41. The DETLOR is driven to the L1DETL.

11.19.14 L1DETL - Leading 1 Detection Logic

The L1DETL is used to determine the bit position of the leading or most significant bit of the F_1%FD1R<B0:B57>. If F_1%FD1R<A0> is a 1 then leading 1 detection is performed on the ones complement of F_1%FD1R<B0:B57>, otherwise leading 1 detection is performed on F_1%FD1R<B0:B57>. The L1DETL output, L1DETLO, is 58 bits wide and has exactly one bit asserted. The L1DETLO output determines the shift required to normalize (the normalizer is in stage 3) the F_1%FD1R in CVTIF and under certain conditions ADD and SUB instructions (for details see the description of the stage 2 control). If L1DETLO<B0> is set, the left shift amount is zero. If L1DETLO<B1> is set, the left shift amount is one, and so on. If the signal E1Z_E2Z is asserted, L1DETO<B0> is set independent of the DETLOR outputs. If E1Z_E2Z is deasserted, the L1DETL outputs depend on the DETLOR outputs. The L1DETLO is driven to the left shift selector LSSEL.

11.19.15 LSSEL - Left Shift Selector

The LSSEL is a 58 bit 2 to 1 selector which selects between L1DETLO<B0:B57> and SDECOR<57:0>. If the signal LSSEL_PASS_SDECOR_H is asserted, then SDECOR<57:0> is passed to LSSELO<B0:B57>. Otherwise L1DETLO<B0:B57> is passed to LSSELO<B0:B57>. LSSEL_PASS_SDECOR_H is asserted if a CVTFI instruction is decoded, or if and an effective subtraction and exponent difference greater than 1 is detected. LSSELO is driven to the LSHR.

11.19.16 LSENC - Left Shift Encoder

The LSENC does a binary encoding of the LSSELO<B0:B57> and drives the encoded signal, LSENCO_DYN<5:0>, to the ED2R. The LSENCO_DYN signal is used in CVTIF and under certain conditions ADD and SUB (for details see the description of the stage 2 control). LSENCO_DYN is used to form the result exponent in CVTIF. LSENCO_DYN is used to correct the result exponent due to normalizing the result in ADD and SUB. LSENCO_DYN<5:0> is driven to the ED2R.

11.19.17 LSHR<57:0> - Left Shifter Control Register

The LSHR is a 58 bit register which is written with LSSELO<B0:B57> unconditionally in PHI_2. The contents of this register determine the number of bit positions the normalizer in stage 3 will shift its input data. Exactly one bit of LSHR<57:0> is asserted. The LSHR output is driven to stage 3.

11.19.18 FD1SEL - Fraction Data 1 Selector

The FD1SEL is a 60 bit 2 to 1 selector that selects the input to the stage 2 FD1R. If FD1_SEL_PASS_0_1 is asserted, zero is passed to the FD1SEL output, FD1SELO. If FD1_SEL_PASS_0_1 is deasserted, then the RSHFTOR is passed to FD1SELO. The FD1SEL output is driven to the FD1R.

11.19.19 FD1R<A0:B58> - Stage 2 Fraction Data 1 Register

The FD1R is a 60 bit register which is written with FD1SELO unconditionally in PHI_2. The contents of this register for all instruction flows are given in the description of the stage 2 control. The FD1R output is driven to stage 3.

11.19.20 FD2R<A0:B58> - Stage 2 Fraction Data 2 Register

The FD2R is a 60 bit register master/slave register. The master register is written with F_1%FD2R unconditionally in PHI_4, and the slave register is written with the output of the master unconditionally in PHI_2. The contents of this register for all instruction flows are given in the description of the stage 2 control. The FD2R output is driven to stage 3.

11.19.21 Exponent Datapath

Figure 11–19: Stage 2 Exponent Datapath Block Diagram

```
                           F_1_E%ED1R_H | |   K | |                                       | | F_1_E%ED2R_H
                                        \ /     | |                                       \ /
                                +---------------+-+----------------------------------------------+
F_2_C%ENA_E12_L,    (NU) ------>|                        ZERO DETECTION                          |------> F_2_E%E
F_2_C%ENA_E22_L     (NU)        +-------+-+-----+-+---------------------------------------+-+----+        F_2_E%E
                           F_1_E%ED1R_H | |   K | |                                       | | F_1_E%ED2R_H
                                        \ /     \ /                                       \ /
                                +----------------------------------------------------------------+
F_2_C%ISEL1_ED1R_A_H/L(AU )  -->|                                                                |
F_2_C%ISEL1_K_A_H/L (NU)    --->|                        INPUT SELECTOR                          |
F_2_C%ISEL1_ED2R_B_H/L(NU)   -->|                         FOR  ADDER 1                           |
F_2_C%ISEL1_K_B_H/L (AU)   ---->|                                                                |
                                +----------------------------------------------------------------+
                           F_1_E%ED1R_H | |   K | |              INP_1A | | INP_1B  | |   | | F_1_E%ED2R_H
                                        | |     | |                     \ /         \ /   | |
                                +----------------------------------------------------------------+
F_2_C%INVERT_EA_AD1_H/L   (NU) ->|                      COMPLEMENT LOGIC                         |
F_2_C%INVERT_EB_AD1_H/L  ------>|                     FOR EXPONENT ADDER 1                       |
                                +----------------------------------------------------------------+
                           F_1_E%ED1R_H | |   K | |              EA_AD1 | | EB_AD1  | |   | | F_1_E%ED2R_H
                                        | |     | |                     \ /         \ /   | |
                                +----------------------------------------------------------------+
F_2_C%CIN_E_AD1_H   ----------->|                       EXPONENT ADDER 1                         |
                                +----------------------------------------------------------------+
                           F_1_E%ED1R_H | |   K | |   E_AD1 | |                           | | F_1_E%ED2R_H
                                        \ /                 | |
                              +-----------------------------+-+----------------------------------+
PHI_4_H       --------------->|                        PHI  LATCHES                              |
                              +---------+-+-----------------+-+----------------------------------+
                          F_2_E%ED1_H   | |           E_AD1 | |
                                        \ /                 \ /
                              +------------------------------------------------------------------+
F_2_C%EN_G_TYPE_L (AU)   ---->|                          OVERFLOW AND                            |
F_2_C%EN_FD_TYPE_L (NU)   --->|                            UNDERFLOW                             |
F_2_C%OSEL1_ZERO_H   -------->|                          DETECTION LOGIC                         |
                              +---------+-+-----------------+-+-----------------+-+--------------+
                            F_2_E%ED1_H | |           E_AD1 | | F_OVF1, F_OVF2  | |
                                        \ /                 \ /                 \ /
                              +------------------------------------------------------------------+
F_2_C%OSEL1_ZERO_L   -------->|                       OUTPUT SELECT                              |
F_2_C%OSEL1_E_AD1_H   ------->|                       AND PHI  LATCHES                           |
(F_2_C%OSEL1_ED1R_H)   ------>|                                                                  |
                              +---------+-+-------------------------------------+-+--------------+
                     +----+             | |                                     | |
F_2_L%LSENCO_H   --->|ED2R|-------------| |-----------------+                   | |
                     +----+             | |                 |                   | |
                       |                | |                 |                   | |  F_2_E%F_OVFR_H (NU)
PHI_2   ---------------+                \ / F_2_E%ED1R_H    V ED2R              \ /  F_2_E%F_UNFR_H (NU)
```

11.19.22 Zero Detection

This functional block is not used in stage 2.

11.19.23 Exponent Adder 1

The exponent adder is a 13-bit static adder used to add or subtract two inputs. Each input is passed through a 2 to 1 selector and inversion logic prior to the adder. INP_1A can be selected from ED1R or K. If ISEL1_ED1R_A is asserted, then ED1R is passed through the selector. If ISEL1_K_A is asserted, then K is passed through the selector. Inversion of the adder input is then done based on the assertion of INVERT_EA_AD1. INP_1B can be selected from ED2R or K. If ISEL1_ED2R_B is asserted, then ED2R is passed through the selector. If ISEL1_K_B is asserted, then K is passed through the selector. Inversion of the adder input is then done based on the assertion of INVERT_EB_AD1. The adder also contains a carry-in to the LSB cell, CIN_E_AD1_H. The carry-in is primarily used for performing subtraction operations. Since the adder is static, it begins its operation when the input data is valid near the falling edge of phase 1. Intermediate results in the exponent adder are latched in phase 3 and sent to the detection logic and output selector.

For stage 2, INP_1A always selects ED1R not inverted. INP_1B always selects K. Inversion of INP_1B is done based on the assertion of CIN_E_AD1. In other words, in stage 2 INVERT_EB_AD1 is shorted to and named CIN_E_AD1.

11.19.24 Floating Overflow and Underflow Detection

This functional block is not used in stage 2.

11.19.25 Output Selector

The output selector is used to select the output data from three different sources: ed1r, ead1 or zero. This selection is done for the exponent output data (ED1R), the floating overflow (F_OVFR) and the floating underflow (F_UNFR). The selection is based on the assertion of two control signals, OSEL1_ZERO and OSEL1_E_AD1. OSEL1_E_AD1 if asserted, selects the output from E_AD1; for overflow and underflow, OSEL1_E_AD1 selects E_AD1_UNF and E_AD1_OVF. If OSEL1_E_AD1 is deasserted, then the output is selected from ED1R; for overflow and underflow, OSEL1_E_AD1 deasserted selects ED1R_OVF and ED1R_UNF. This selection is done using a 2 to 1 selector. The selection of zero is done prior to the 2 to 1 selector described above. If OSEL1_ZERO is asserted, then the inputs from E_AD1 and ED1R entering the 2 to 1 selector are both forced to zero. Then, since only one select line is used to control the selector, the zero value will be transferred to the output regardless of the assertion of OSEL1_E_AD1. The output of the selector is latched every phase 1 and driven into the following stage.

11.19.26 ED2R<5:0> - Exponent Data 2 Register

The ED2R is a 6 bit register which is written with LSENCO_DYN<5:0> unconditionally in PHI_2. The ED2R output is driven to the stage 3 exponent datapath.

11.19.27 Sign Datapath

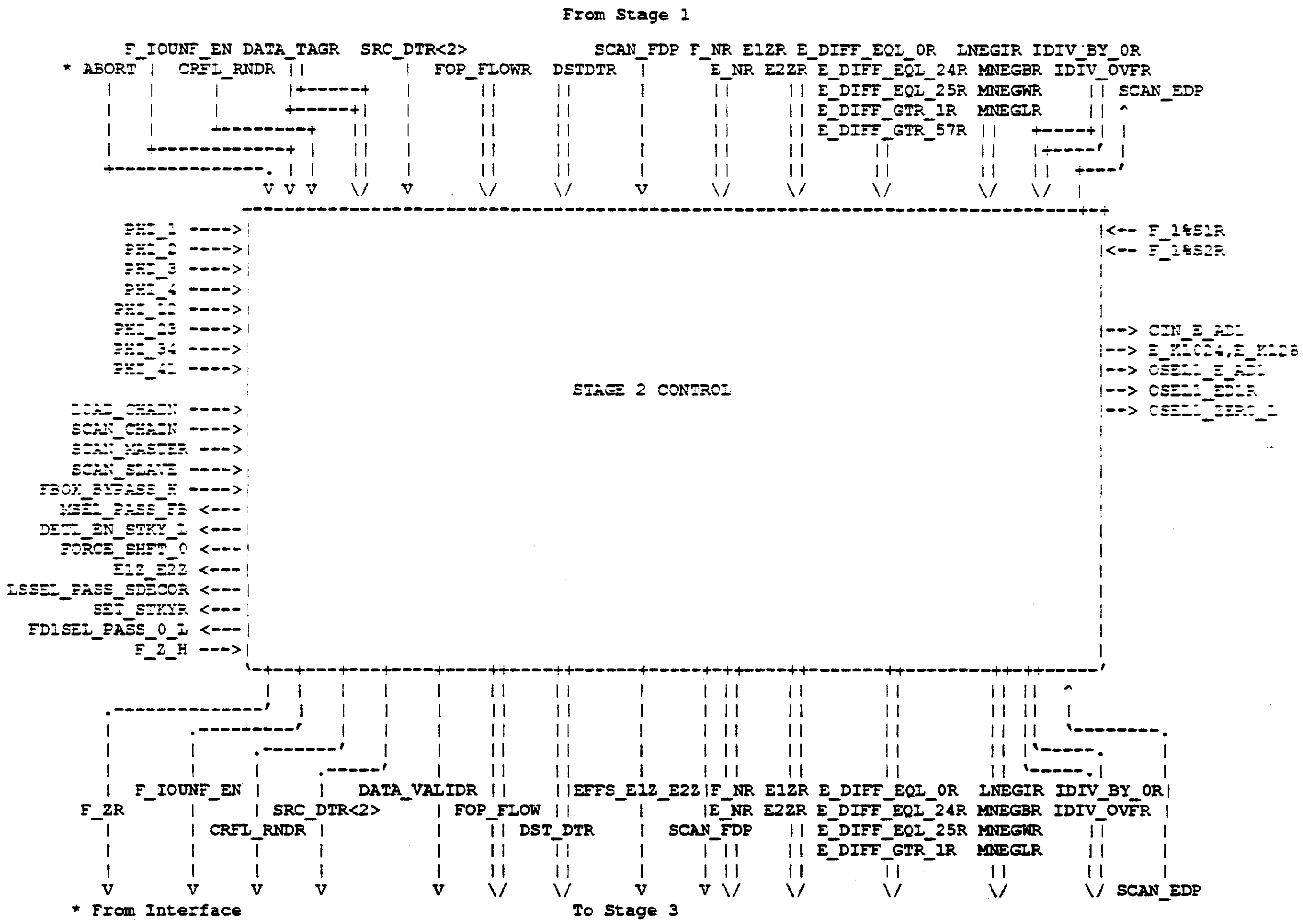
Figure 11–20: Sign Datapath Block Diagram

```
                          | F_1  %S1R_H       | F_1%S2R_H
                          |                   |
                <---------+                   |
To Stage 2      <---------|-------------------+
 Control                  |                   |
                          V                   V
                +-------------------+-------------------+
PHI_4 --------->|        S1R        |        S2R        |
PHI_2           +-------------------+---------+---------+
                          |                   |
                          |                   |
                          V F_2%S1R_H         V F_2%S2R_H
```

The 2 inputs to the stage 2 sign datapath are F_1%S1R_H and F_1%S2R_H These bits correspond to the sign of operand 1 and the sign of operand 2, respectively. The stage 2 datapath does not perform any operations on the sign bits. S1R and S2R are 1 bit master-slave registers. The master latches are written unconditionally in PHI_4 and the slave latches are written with the master latch outputs unconditionally in PHI_2. The register outputs F_1%S1R_H and F_2%S2R_H are driven to stage 3.

11.19.28 Control

Figure 11–21: Control Block Diagram



The stage 2 control block generates control signals for the stage 2 fraction and exponent datapaths based on opcode and control information passed from stage 1. The control block decodes the datapath control signals one cycle prior to the cycle in which they are needed in the datapaths. The control signals are latched in master slave latches to allow control decoding to overlap with datapath execution and to prevent races. The control block also loads control information output from stage 1 into master slave registers and passes the information to stage 3. The master slave latches which hold the SRC_DTR<2>, FOP_FLOWR<5:0>, DATA_VALIDR, and DST_DTR<2:0> signals are written in PHI_1 (master strobe) and PHI_3 (slave strobe). All other master slave latches are written unconditionally in PHI_4 and PHI_2. If the interface section asserts F_I%ABORT_H the signals F_2%LAT_MUL2_H and F_2%DATA_VALIDR_H are deasserted.

The internal signal F_2%LAT_MUL2_h is used to facilitate the sequencing of the fraction multiplier and stalling of the DATA_VALIDR bit transfer for MULD/G, and MULL instructions. F_2%LAT_MUL2_H is asserted during the second decode cycle of MULD, MULG, and MULL instructions. If F_2%LAT_MUL2_H is asserted, the DATA_VALIDR bit will not be passed to stage 3, and the multiplier will select fed-back outputs as its inputs. F_2%LAT_MUL2_H is unconditionally deasserted one cycle after it is asserted.

The stage 2 control block also modifies one bit of the internal opcode encoding, F_1%FOP_FLOWR_H<1>, before passing it to the stage 3 control if the conditions effective sub and exponent difference greater than 1 are detected. It also contains the FZR bit latch and some logic to conditionally clear the latch. If an effective subtraction is decoded, and E1ZR XOR E2ZR is true, the FZR bit latch will be cleared. If this condition is not true, the FZR bit latch will loaded with the FZ output of the DETL in the fraction datapath.

11.19.28.1 Datapath Control Signals Output from Control Block

CIN_E_AD1_H : This is the carry-in to the LSB position of the exponent datapath adder, E_AD1. This signal also controls the ones complementing of the exponent bias. If asserted the ones complement of the exponent bias is passed to the EB_AD1 output of the exponent complement logic. If deasserted, the true exponent bias is passed to the EB_AD1 output unchanged. F_2_C%CIN_E_AD1_His asserted if a MULf instruction is decoded by the stage 2 control.

F_2_C%DETL_EN_STKY_L :This enables the DETL to detect conditions for setting the sticky bit which is used by the stage 3 fraction adder. This signal is asserted if an effective subtraction is decoded and the exponent difference between the operands is greater than one.

F_2_C%E_K_H<7> : This signal is an exponent bias which is driven to the INP_1B input of the exponent complement logic. This signal is the complement of E_K_H%F_2_C<10>.

F_2_C%E_K_H<10> : This signal is an exponent bias which is driven to the INP_1B input of the exponent complement logic. This signal is asserted if the F_1_C%DST_DTR_H<2:0> decodes to G datatype.

F_2_C%E1Z_E2Z_H : If this signal is asserted, the L1DETLO<B0> bit will be set (which indicates the contents of the F_1%FD1R is a normalized number) independent of the other inputs to the L1DETL. This signal is asserted if (F_1%E1ZR OR F_1%E2ZR) AND (effective sub) is detected.

F_2_C%FD1SEL_PASS_0_L : If this signal is asserted then the FD1SEL will pass zeros to it's outputs and the stage 2 FD1R will load in all zeros. This signal is asserted if F_1_E%EDIFF_GTR_57_H is asserted, and ADDf or SUBf or CVTfi or CVTRfi is decoded.

F_2_C%LSSEL_PASS_SDECOR_H : If this signal is asserted F_2_P%SDECOR_H is passed to the LSSEL output, F_2%LSSELO_H. If deasserted, F_2_L%L1DETLO_H is passed to F_2_L%LSSELO_H. LSSEL_PASS_SDECOR_H%F_2_C is asserted if a CVTfi or CVTRfi instruction is decoded, or if an effective subtraction and exponent difference greater than 1 is detected.

F_2_C%MSEL_PASS_FB_H : If this signal is asserted F_2_M%MCR_L is passed to the MSEL carry output, MCSELO, and the F_2_M%MSR_L is passed to the MSEL sum output. If deasserted, the MTCR%_1 is passed to MCSELO, with zeros in the vacant bit positions, and the F_1%MIPPR is passed to MSELO. F_2_C%MSEL_PASS_FB_H is asserted if the internal signal F_2_C%LAT_MUL2_H is asserted.

F_2_C%OSEL1_E_AD1_H / F_2_C%OSEL1_ED1R_H : The OSEL1_E_AD1_H and the OSEL1_ED1R_H signals are complementary signals. If OSEL1_E_AD1_H is asserted, the exponent output selector, OSEL1, passes E_AD1, E_AD1_OVF and E_AD1_UNF to its outputs. If OSEL1_ED1R_H is asserted, ED1R, ED1R_OVF and ED1R_UNF are passed to the OSEL1 outputs. OSEL1_E_AD1_H is asserted if a MUL or DIV is decoded by the stage 2 control.

F_2_C%OSEL1_ZEROS_L : If this signal is asserted the exponent output selector, OSEL1, passes zero to its output. If deasserted, zero is not passed to the OSEL1 outputs. This signal is asserted if a MUL or a DIV is decoded, and F_1_E%E1ZR_H or F_1_E%E2ZR_H is asserted.

F_2_C%SET_STKYR_H : If this signal is asserted the STKYR is forced to 1, independent of the state of the F_1%FD1R and the SDECOR. If deasserted, the state of the STKYR depends on the instruction flow and the data. SET_STKYR_H%F_2_C is asserted if F_1_E%E_DIFF_GTR_57R_H AND NOT(F_1%E1ZR OR F_1%E2ZR%) is true.

F_2_C%FORCE_SHFT_0_H : This signal forces the RSHIFT to pass the FD1R%_1 to its output unshifted. If this signal is deasserted, then the RSHIFT shifts the F_1%FD1R by the number of bit positions decoded by the SDEC. This signal is deasserted if an effective sub is decoded and the F_1_E%E_DIFF_GTR_1R_H is asserted, or if an effective add is decoded, or a CVTfi or a CVTRfi is decoded and F_1_E%E_NR_H is low.

11.19.29 Stage 2 Fraction Datapath Operation Summary

The following tables summarize the operation of the stage 2 fraction and exponent datapaths.

Figure 11–22: Fraction Datapath Operation Summary

Conditions							Fraction Datapath Registers and Outputs							
OPC	EFF A/S	E1Z	E2Z	E_N	EDIFF	M2	FD1R	FD2R	F_ZR	LSHR	STKYR	MC/SR	MILSBC/
ADDf	A	0	0	X	ED<58	X	R(FD1R_1)	FD2R_1	UD	UD	UD	UD	UD
SUBf	A	0	1	X	ED<58	X	R(FD1R_1)*	FD2R_1	UD	UD	UD	UD	UD
	A	1	0	X	ED<58	X	R(FD1R_1)*	FD2R_1	UD	UD	UD	UD	UD
	A	1	1	X	ED<58	X	R(FD1R_1)*	FD2R_1	UD	UD	UD	UD	UD
	A	X	X	X	ED>57	X	0	FD2R_1	UD	UD	UD	UD	UD
	S	0	0	X	ED=0	X	FD1R_1	FD2R_1	0	L1DETLOR	UD	UD	UD
	S	0	0	X	ED=0	X	FD1R_1	FD2R_1	1	0	UD	UD	UD
	S	0	1	X	ED=0	X	UD	UD	UD	UD	UD	UD	UD
	S	1	0	X	ED=0	X	UD	UD	UD	UD	UD	UD	UD
	S	1	1	X	ED=0	X	FD1R_1*	FD2R_1	1	LSHR<57>=1	UD	UD	UD
	S	0	0	X	ED=1	X	FD1R_1	FD2R_1	0	L1DETLOR	UD	UD	UD
	S	0	1	X	ED=1	X	FD1R_1*	FD2R_1	0	LSHR<57>=1	UD	UD	UD
	S	1	0	X	ED=1	X	FD1R_1*	FD2R_1	0	LSHR<57>=1	UD	UD	UD
	S	1	1	X	ED=1	X	UD	UD	UD	UD	UD	UD	UD
	S	0	0	X	ED>57	X	0	FD2R_1	UD	UD	1	UD	UD
	S	0	1	X	ED>57	X	0	FD2R_1	UD	UD	0	UD	UD
	S	1	0	X	ED>57	X	0	FD2R_1	UD	UD	0	UD	UD
	S	1	1	X	ED>57	X	UD	UD	UD	UD	UD	UD	UD
	S	0	0	X	1<ED<58	X	R(FD1R_1)	FD2R_1	UD	UD	STKY	UD	UD
	S	0	1	X	1<ED<58	X	R(FD1R_1)*	FD2R_1	UD	UD	STKY	UD	UD
	S	1	0	X	1<ED<58	X	R(FD1R_1)*	FD2R_1	UD	UD	STKY	UD	UD
	S	1	1	X	1<ED<58	X	UD	UD	UD	UD	UD	UD	UD
CMPf, TSTf	X	X	X	X	X	X	FD1R_1	FD2R_1*	F_Z	UD	UD	UD	UD
DIVf, MOVf, MNEGf, CVTff	X	X	X	X	X	X	FD1R_1	FD2R_1*	UD	UD	UD	UD	UD
CVTfi,	X	X	X	0	ED<58	X	R(FD1R_1)	FD2R_1*	UD	UD	UD	UD	UD
CVTRfi	X	X	X	1	ED<58	X	FD1R_1	FD2R_1*	UD	SDECOR	UD	UD	UD
	X	X	X	X	ED>57	X	0	FD2R_1*	UD	UD	UD	UD	UD
CVTif	X	X	X	X	X	X	FD1R_1	FD2R_1*	F_Z	L1DETLOR	UD	UD	UD

Figure 11–22 Cont'd on next page

Figure 11–22 (Cont.): Fraction Datapath Operation Summary

Conditions							Fraction Datapath Registers and Outputs						
OPC	EFF A/S	E1Z	E2Z	E_N	EDIFF	M2	FD1R	FD2R	F_ZR	LSHR	STKYR	MC/SR	MILSBC/SR
MULF	X	0	0	X	X	0	FD1R_1#	FD2R_1##	UD	UD	UD	C/S	UD
MULF	X	1	0	X	X	0	FD1R_1*	FD2R_1*	UD	UD	UD	0	UD
MULF	X	0	1	X	X	0	FD1R_1*	FD2R_1*	UD	UD	UD	0	UD
MULF	X	1	1	X	X	0	FD1R_1*	FD2R_1*	UD	UD	UD	0	UD
MULF	X	X	X	X	X	1	UD	UD	UD	UD	UD	UD	UD
MULDG	X	0	0	X	X	0	FD1R_1#	FD2R_1##	UD	UD	UD	UD	UD
MULDG	X	0	0	X	X	1	FD1R_1#	FD2R_1##	UD	UD	UD	C/S	UD
MULDG	X	1	X	X	X	0	FD1R_1*	FD2R_1*	UD	UD	UD	UD	UD
MULDG	X	X	1	X	X	0	FD1R_1*	FD2R_1*	UD	UD	UD	UD	UD
MULDG	X	1	X	X	X	1	FD1R_1*	FD2R_1*	UD	UD	UD	0	UD
MULDG	X	X	1	X	X	1	FD1R_1*	FD2R_1*	UD	UD	UD	0	UD
MULL	X	X	X	X	X	0	FD1R_1#	FD2R_1##	UD	UD	UD	UD	UD
MULL	X	X	X	X	X	1	FD1R_1#	FD2R_1##	UD	UD	UD	C/S	C/S

OPC - OPCODE
EFF A/S - Denotes that the operation is an effective ADD or SUB.
ED - Exponent difference.
M2 - Denotes the second execution cycle of a MUL instruction in stage 2.
C/S - Denotes valid Carry and Sum vectors.
R(AAA) - Denotes that the stage 2 right shifter input is AAA
UD - Undefined
* - Stage 1 forces this register to be all zeros.
- Contains 3*multiplicand generated in stage 1.
- Contains 1*multiplicand.

Figure 11–23: Stage 2 Exponent Datapath Operation Summary

Conditions								Exponent Datapath Registers	
OPC	EFF A/S	E1Z	E2Z	E_N	EDIFF	M2	K	ED1R	ED2R
ADDf	A	X	X	X	X	X	X	ED1R_1	UD
SUBf	S	0	0	X	ED=0,1	X	X	ED1R_1	LSENCO
	S	0	1	X	ED=0,1	X	X	ED1R_1	0
	S	1	0	X	ED=0,1	X	X	ED1R_1	0
	S	1	1	X	ED=0,1	X	X	ED1R_1	0
	S	X	X	X	ED>1	X	X	ED1R_1	UD
CMPf, TSTf	X	X	X	X	X	X	X	ED1R_1	UD
DIVF/D	X	0	0	X	X	X	128	E_AD1=ED1R_1+K	UD
DIVG	X	0	0	X	X	X	1024	E_AD1=ED1R_1+K	UD
DIVf	X	1	X	X	X	X	X	0	UD
DIVf	X	X	1	X	X	X	X	0	UD
MOVf, MNEGf, CVTff, CVTfi, CVTRfl	X	X	X	X	X	X	X	ED1R_1	UD
CVTif	X	X	X	X	X	X	X	UD	LSENCO
MULF/D	X	0	0	X	X	X	128	E_AD1=ED1R_1-K	UD
MULG	X	0	0	X	X	X	1024	E_AD1=ED1R_1-K	UD
MULf	X	1	X	X	X	X	X	0	UD
MULf	X	X	1	X	X	X	X	0	UD
MULL,	X	X	X	X	X	X	X	UD	UD

```
    OPC  - OPCODE
EFF A/S  - Denotes that the operation is an effective ADD or SUB.
     ED  - Exponent difference.
     UD  - Undefined
```

11.19.30 Passthru Signals

MMGT_FLT_L, MEM_ERR_L, RSV_ADR_L and PSL_FU_H signals are simply passed through stage-2 without change. They are latched coming in from Stage-1 during PHI_4 and driven to Stage-3 during PHI_2.

NEW_FOP_H signal also passes through to Stage-3 unaffected. It is latched during PHI_1 coming from Stage-1 and driven to Stage-3 during PHI_3. This signal is gated with the global purge signal F_I%PURGE_H from the input interface which clears it on a PURGE from the input interface. This signal is used by the Output Interace to manipulate its control-queue and data-queue pointers.

11.20 STAGE 3

11.20.1 Introduction

Stage 3 of the pipeline is used primarily to left shift an input, or to perform the addition of two inputs. Stage 3 contains a control section and portions of the fraction, exponent, and sign datapaths. In addition, stage 3 has the capability to bypass stage 4 rounding operation for certain instructions. Stage 3 takes virtually all of its inputs from stage 2 of the pipeline, and drives it's outputs either to stage 4 or to the output interface directly.

The fraction datapath portion of stage 3 consists of a left shifter, an instantiation of the generic adder and some mini-rounding incrementers. The left shifter is used for convert and effective subtraction-like operations. The adder is used by all other operations either to pass an input to the output (by adding zero), or to add two vectors--for example, the two input operands (correctly aligned) for addition/subtraction, or the sum and carry vectors for multiplication. The mini-rounding incrementers are used to round the fraction result during a stage 4 bypass operation. Stage 3 also performs the injection of the sticky bit and increments the quotient, dependent on the sign of the remainder. The output of stage 3 is always normalized, where relevant.

The exponent datapath consists of the generic exponent block. In this stage, the input selector, adder, and output selector are primarily used. For addition, subtraction, multiplication, and division, the adder is used to increment/decrement the input exponent according to whether the fraction addition can overflow/underflow. It also subtracts the left shift amount when the fraction portion performs a left shift.

The sign datapath portion in stage 3 will generate the correct sign for the result during a successful stage 4 bypass. No operation is performed on the sign bit that is sent to stage 4.

Some integer overflow detection logic is included in the control path. Additionally the six LSB's generated for MULL are combined, and a few stage 4 signals are generated.

11.20.2 Stage 4 Bypass

For a specific set of instructions and conditions, stage 3 can supply a result to the output interface directly. This is referred to as a "stage 4 bypass" and improves Fbox latency by supplying a result one full cycle earlier than the stage 4 supplied result. In order to bypass stage 4, stage 3 must perform the required operations that stage 4 would normally perform under the same conditions. This includes rounding the fraction, supplying the correct exponent and generation of the condition codes and status information that is related to the result for floating ADD, SUB and MUL instructions.

Stage 3 performs the rounding operation through the use of incrementers. These incrementers are much smaller in width than the number of fraction bits for a particular data type due to timing constraints. Because of the limited size of the incrementers not all fraction datums can be correctly rounded by stage 3. (The mini-round succeeds if the selected incrementer for a bypassable instruction does not generate carry out.) If the mini-round fails, the unmodified fraction is driven and the stage 4 bypass is aborted.

Stage 3 and stage 4 share common busses to drive results to the output interface. Stage 4 will drive the busses, during phi3, if it has a valid data. Stage 3 will drive the busses, during phi3, if it can successfully bypass an instruction and stage 4 does not have a valid data.

The stage 3, stage 4 common busses are listed below:

```
f_b%f_out_l<b0:b55>  fraction result bus
f_b%e_out_l<10:0>  exponent result bus
f_b%s_out   sign result bus
f_b%n_out   psln result bus
f_b%z_out   pslz result bus
f_b%v_out   pslv result bus
```

11.20.2.1 Stage 4 Bypass Request

When stage 3 has detected that a stage 4 bypass may be possible it signals the output interface by asserting the signal **F_3%S4_BYPASS_REQR_H** during phi4.

All of the following conditions must be met in order to generate a stage 4 bypass request.

o *The signal* **F_I%S4_BYPASS_ENB_H** *must be asserted.*

o *The FOP meets one of the following conditions.*

1. ADD F,D,G
2. MUL F,D,G
3. SUB F,D,G (except when ediff=0 AND fraction result is negative)

o *The signal* **F_2%DATA_VALIDR_H** *is asserted indicating that the data present at stage 3's input is valid.*

o *The signal* **F_3%DATA_VALIDR_H** *is NOT asserted indicating that a result was not sent to stage 4 in the previous cycle.*

o *There are no faults associated with the data. (mmgt_flt, mem_err, rsv_adr)*

o *Neither of the two input operands are reserved operands.*

11.20.2.2 Stage 4 Bypass Abort

In order to abort a stage 4 bypass, the signal **F_3%S4_BYPASS_ABORTR_H** must be asserted during phi2. Either of the two following conditions must be met in order to abort a stage 4 bypass assuming the bypass request was generated.

o *Mini-round failure. The selected mini-round incrementer carried out of it's most significant bit position.*

o *Exponent overflow or underflow is detected on either of the two exponent results in stage 3's exponent section, irrespective of the possible 1-bit left or right shift required for the fraction adder result.*

11.20.2.3 Stage 3 Response to FBOX Purge

Stage 3 responds to the FBOX purge by clearing from stage 3, the data_valid flag and also the new_fop flag.

11.20.3 Section Implementation Description

11.20.3.1 Block Diagrams

Stage 3 is made up of three sections: control, fraction, and exponent. On the following pages, block diagrams of the fraction and exponent datapaths are shown.

Figure 11–24: Stage 3 Fraction Datapath Block Diagram

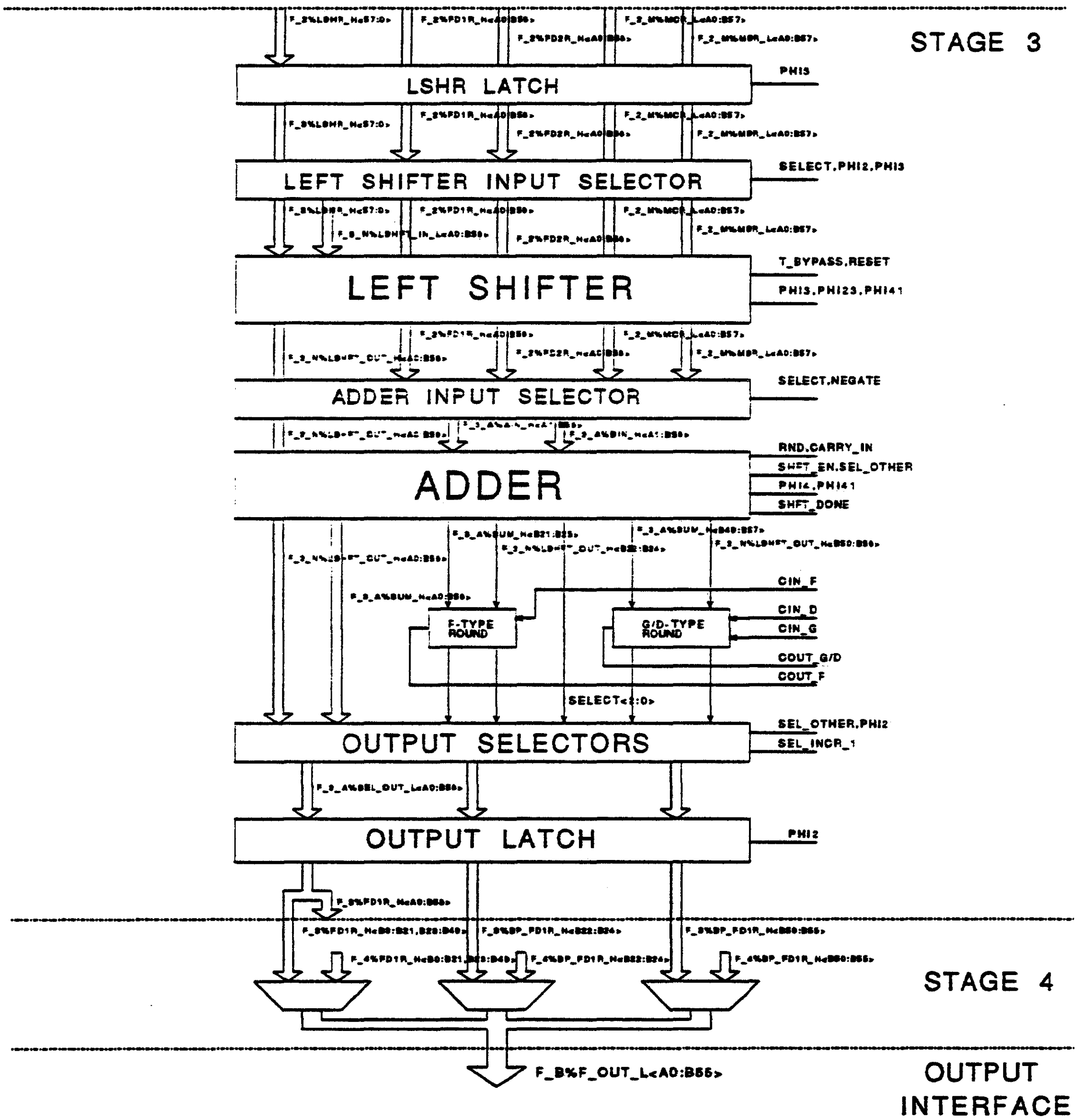


Figure 11–25: Stage 3 Fraction Mini-round Block Diagram

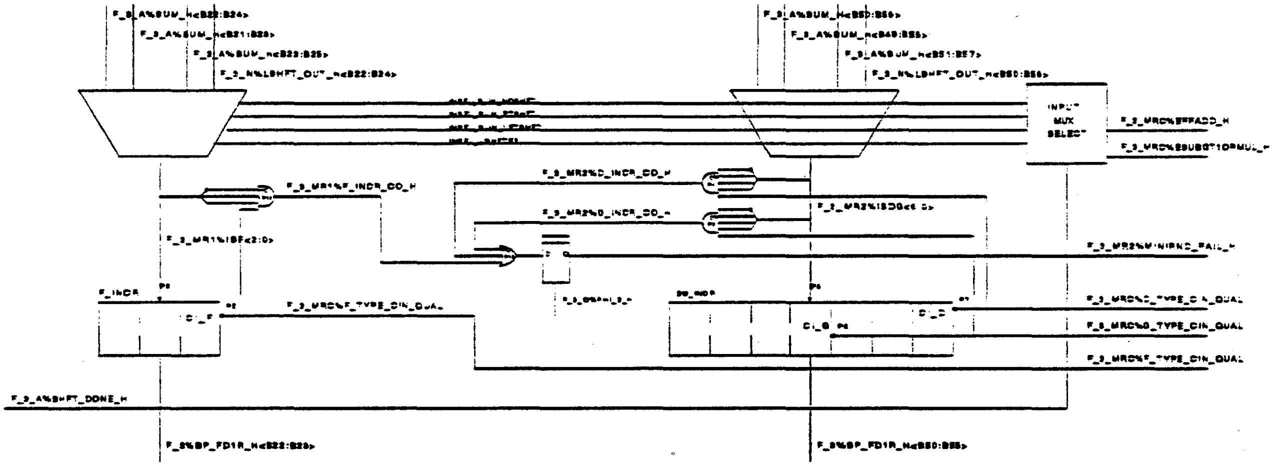
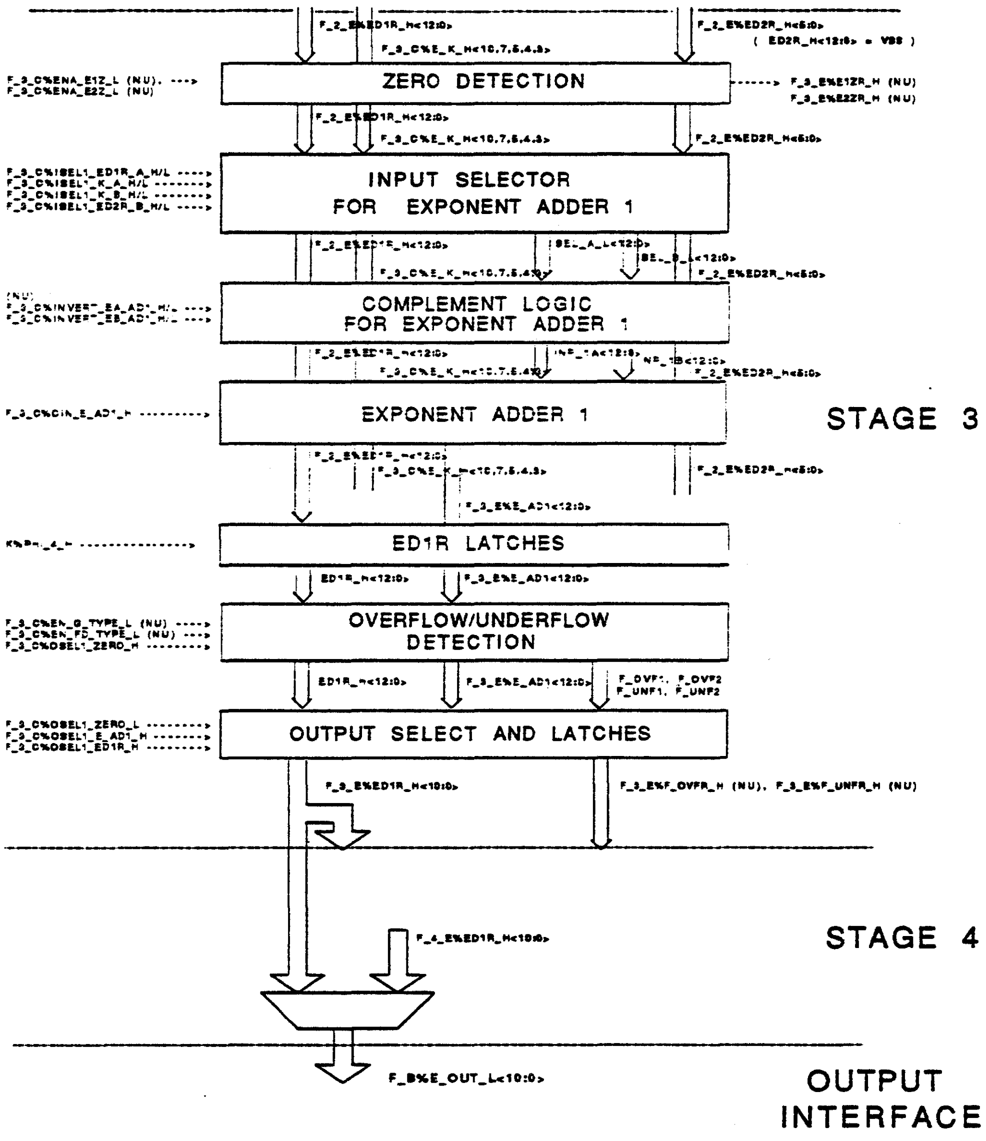


Figure 11–26: Stage3 Exponent Datapath Block Diagram



11.20.4 Fraction Datapath

The operations performed in the fraction datapath in this stage are shown in the following table.

Table 11–11: Stage 3 Fraction Datapath Operations

Category	Operation	Condition
F0	LSHFT_OUT <– FD1R.SHL.[LSHR]	EFF.SUBf, deltaE < 2, neither operand = 0; CVTif; CVTfi, left shift; CVTRfL, left shift
F1	LSHFT_OUT <– FD2R.SHL.[LSHR]	EFF.SUBf, deltaE < 2, operand(s) = 0
F2	SUM <– FD2R + FD1R	EFF.ADDf; CVTff; MOVf; MNEGf; CMPf; TSTf; CVTfi, right shift
F3	SUM <– FD2R + .not.FD1R + .not.STKYR	EFF.SUBf, deltaE > 1
F4	SUM <– FD2R + FD1R + Rnddi	CVTRfL, right shift
F5	SUM <– FD2R + FD1R + .not.F_NR	DIVf
F6	SUM <– .not.MCR + .not.MSR	MULf; MULL;

11.20.4.1 Normalizer Input Selection

The data to be left-shifted may be contained in F_2%FD1R_H or F_2%FD2R_H. The normalizer input selector is used to select between these two input registers.

11.20.4.2 Left Shifter

The left shifter is capable of performing zero to fifty-seven bit left shifts. The shift amount is driven on the LSHR lines in decoded form. The output of the left shifter is driven on LSHFT_OUT to the stage 3 output selector. For effective subtraction, exponent difference equal to zero, the output of the left shifter may be negative. The shift amount is forced to "shift of zero" if stage 3 is in FBOX_Test mode or the chip is reset.

11.20.4.3 Adder Input Selection

The adder is driven with two input vectors: AIN and BIN. AIN can be FD2R or MSR; BIN can be FD1R or MCR. Note that for several operations, either FD1R or FD2R must be zero; the data is contained in the other register.

These operations are:

CVTff
MOVf
MNEGf
CMPf
TSTf
CVTfi, right shift
CVTRfL, right shift
DIVf

11.20.4.4 Adder

The adder uses two 61-bit inputs to derive a 62-bit result. The 61-bit inputs have two bits above the binary point and 59 bits below; the 62-bit result has an extra bit above the binary point. In this stage, the most significant bit of each input is not used; neither are the two most significant bits of the output.

The main carry acceleration technique used is carry select. The adder is broken up into nine small groups, with all but the least significant group having duplicate carry chains. These carry chains operate in parallel during the early part of the execute cycle. Propagate and generate logic operates before the carry chains. These parts of the adder are fully static.

During the late part of the execute cycle, the sum logic executes. Just as for the carry logic, there is duplicate sum logic for all groups except the least significant one. In addition, logic to derive the true group carry out signals executes in these phases. These carry out signals are used to select the correct sum values. These parts of the adder are also fully static.

NOTE FOR MULL:

> The adder in stage 3 adds the 58 MSB's generated by the multiplier array. <B58> of AIN and BIN is forced to zero for multiply operations.

Shift Detection Logic:

The most significant group of adder bits, bit positions <A2:B1>, is different from the groups below it. In this group, both the carry and sum logic execute during the early part of the execute cycle. Late in the execute cycle, shift detection logic executes. If enabled, it examines the sum bits <A0:B1> to determine whether a one bit shift right or left is needed to normalize the result. The possible values of sum bits <A0:B1> are given in the table below for each operation which may yield a non-normalized adder result.

Table 11–12: Possible Values For Sum Bits <A0:B1>

Result #1	Result #2	Result #3	Operation
0.1X	1.XX	0.00	EFF.ADDf
0.1X	0.01	0.00	EFF.SUBf, deltaE>1
0.1X	0.01	0.00	MULf
0.1X	1.XX	0.00	DIVf

If the shift detection logic is disabled, then the signal indicating "no shift needed" will be asserted.

This logic is also conditioned with another signal (sel_other) which is used to deassert all of the shift detection signals. Since these shift detection signals are used to drive the output selector for the stage, this feature permits the selection of a stage output other than the shifted or unshifted adder result.

The logic used to control the shifting is as follows:

$$\mathit{DET\_SHR1\_H} = \mathit{A0} * \mathit{SHIFT\_EN} * \overline{\mathit{SEL\_OTHER}}$$

$$\mathit{DET\_PASS\_H} = (((\overline{\mathit{A0}}*\mathit{B0} + \overline{\mathit{A0}}*\overline{\mathit{B0}}*\overline{\mathit{B1}}) * \mathit{SHIFT\_EN}) + \overline{\mathit{SHIFT\_EN}}) * \overline{\mathit{SEL\_OTHER}}$$

$$\mathit{DET\_SHL1\_H} = \overline{\mathit{A0}}*\overline{\mathit{B0}}*\mathit{B1} * \mathit{SHIFT\_EN} * \overline{\mathit{SEL\_OTHER}}$$

Det_shr1 detects the case in which the fraction result is 1.XX..XX, and thus the fraction must be shifted right by one bit to be normalized. Det_pass detects several cases: first, the case in which the fraction result is 0.1XX..XX; second, the case in which the fraction result is zero (0.00XX..XX); last, the case in which shifting is disabled. Det_shl1 detects the case in which the fraction result is 0.01XX..XX, and the fraction must thus be shifted left by one bit to be normalized.

The detection logic is duplicated, with one copy for each of the two sets of sum bits. This logic is fully static. The correct shift signals are selected dynamically by the true group carry out of the previous group, and driven out of the adder. A signal indicating whether a shift was done is driven to the exponent section, where it is used in selecting the proper exponent output.

Bit Injection Within Adder:

The adder performs rounding and two's complementing for all datatypes. The following table shows the bit positions into which injection is done. The bit positions are defined as c(y), meaning the carry in of the yth bit position. This carry in is derived by forcing a carry out to be generated in bit position (y-1). Only Rnddi is used in this stage.

Table 11–13: Bit Injection Within Adder

Type of Injection			
Rndf	Rnddi	Rndg	Cinb55_one
c(B24)			
	c(B56)		
		c(B53)	
			c(B55)

The carry in to bit position <B58> is set directly by the stage's control section.

11.20.4.5 Mini-Round Incrementers

These incrementers are used to round the fraction result supplied by either the left shifter or the adder. The incrementer for D and G type is four bits wide while the incrementer for F type is three bits wide.

11.20.4.6 Output Selector

The output selector is a precharged 1-of-4 selector. It selects either the left shifter output or the adder output (shifted left one bit position, passed unshifted, or shifted right one bit position). Three of the four selector control signals (the three adder output selection signals) are driven from the adder to the output selector; the fourth (the left shifter output selection signal) is driven from the control section.

11.20.4.7 Fraction Datapath Operation Summary (Normal Operating Mode):

Figure 11–27: Fraction Datapath Operation Summary

Conditions						FDP Inputs							FDP Output
OPC	EFF A/S	E_N	EDIFF	E1Z	E2Z	FD1R	FD2R	MCR	MSR	LSHR	STKYR	F_N	FD1R
ADDf,	A	x	x	x	x	V	V	x	x	x	x	x	SUM
SUBf	S	x	ED<2	0	0	V	x	x	x	V	x	x	LSHFT_OUT
	S	x	ED<2	1	x	x	V	x	x	V	x	x	LSHFT_OUT
	S	x	ED<2	x	1	x	V	x	x	V	x	x	LSHFT_OUT
	S	x	1<ED<58	x	x	V	V	x	x	x	V	x	SUM
	S	x	ED>57	0	0	V	V	x	x	x	1	x	SUM
	S	x	ED>57	0	1	V	V	x	x	x	0	x	SUM
	S	x	ED>57	1	0	V	V	x	x	x	0	x	SUM
CMPf	x	x	x	x	x	V	V	x	x	x	x	x	SUM
TSTf	x	x	x	x	x	V	V	x	x	x	x	x	SUM
DIVf	x	x	x	x	x	V	V	x	x	x	x	V	SUM
MOVf	x	x	x	x	x	V	V	x	x	x	x	x	SUM
MNEGf	x	x	x	x	x	V	V	x	x	x	x	x	SUM
MULf	x	x	x	x	x	x	x	V	V	x	x	x	SUM
MULL	x	x	x	x	x	x	x	V	V	x	x	x	SUM
CVTff	x	x	x	x	x	V	V	x	x	x	x	x	SUM
CVTfi	x	1	x	x	x	V	x	x	x	V	x	x	LSHFT_OUT
CVTfi	x	0	x	x	x	V	V	x	x	x	x	x	SUM
CVTif	x	x	x	x	x	V	x	x	x	V	x	x	LSHFT_OUT
CVTRfL	x	1	x	x	x	V	x	x	x	V	x	x	LSHFT_OUT
CVTRfL	x	0	x	x	x	V	V	x	x	x	x	x	SUM

OPC - Opcode
EFF A/S - Effective Addition (A) or Effective Subtraction (S)
SUM - Adder Output, shifted left/passed unshifted/shifted right as needed
ED - Exponent Difference
V - Valid data
x - Don't care

11.20.5 Exponent Datapath

The operations performed in the stage 3 exponent datapath adder are shown in the table below. Note that the exponent operation category numbers are unrelated to the fraction operation category numbers.

Table 11–14: Exponent Datapath Operation Summary

Category	Operation done in Adder	Condition
E0	NOOP	CMPf; TSTf
E1	E_AD1 <-- ED1R + K + 1 (ER + 0 + 1)	DIVf; EFF.ADDf
E2	E_AD1 <-- ED1R + .not.K (ER - 1)	MULf; MULL; EFF.SUBf, deltaE>1
E3	E_AD1 <-- ED1R + .not.ED2R + 1 (ER - NORM)	EFF.SUBf, deltaE<2
E4	E_AD1 <-- K + .not.ED2R + 1 (BIAS1 - NORM)	CVTif
E5	E_AD1 <-- ED1R + .not.K + 1 (ER - BIAS2)	CVTfi, CVTRfL
E6	E_AD1 <-- ED1R + K (ER + BIAS3)	CVTff, MOVf, MNEGf

11.20.5.1 Constants

Five bits (bits <BITMAP>(10), <BITMAP>(7), and <5:3>) of the exponent constants are driven from the control section into the exponent section. The other eight constant bits are hardwired to ground within the exponent block. The constants needed in stage 3 are:

```
K0 = 0000000000000     =    0
     1111111111111     =   -1   = NOT(K0)
K1 = 0000010100000     =  160   ; CVT{B,W,L}{F,D}
K2 = 0010000100000     = 1056   ; CVT{B,W,L}G
K3 = 0000000011000     =   24   ; CVT{F,D,G}L/CVTR{F,D,G}L
K4 = 0000000101000     =   40   ; CVT{F,D,G}W
K5 = 0000000110000     =   48   ; CVT{F,D,G}B
K6 = 0000010000000     =  128   ; CVT{D,G}F/CVTFD
K7 = 0010000000000     = 1024   ; CVTFG
```

K1 and K2 are the BIAS1 constants, used in CVTif; K3, K4, and K5 are the BIAS2 constants, used in CVTfi and CVTRfL; K6 and K7 are the BIAS3 constants used in CVTff, MOVf, and MNEGf.

11.20.5.2 Zero Detection

The zero detectors are not used in stage 3.

11.20.5.3 Exponent Adder 1

The exponent adder is a 13-bit static adder used to add or subtract two inputs. Each input is passed through a 2 to 1 selector and inversion logic prior to the adder.

INP_1A can be selected from ED1R or K. If ISEL1_ED1R_A is asserted, then ED1R is passed through the selector. If ISEL1_K_A is asserted, then K is passed through the selector. Inversion of the adder input is then done based on the assertion of INVERT_EA_AD1. INP_1A is never inverted in this stage.

INP_1B can be selected from ED2R or K. If ISEL1_ED2R_B is asserted, then ED2R is passed through the selector. If ISEL1_K_B is asserted, then K is passed through the selector. Inversion of the adder input is then done based on the assertion of INVERT_EB_AD1.

The adder also contains a carry-in to the LSB cell, CIN_E_AD1_H. The carry-in is primarily used for performing subtraction operations. The table below gives the carry value for each exponent operation category:

Table 11–15: LSB Carry-In Values

Category	Carry In
E0	d
E1	1
E2	0
E3	1
E4	1
E5	1
E6	0

Since the adder is static, it begins its operation when the input data is valid near the falling edge of phase 2. Intermediate results in the exponent adder are valid by the middle part of the execute cycle and sent to the detection logic and output selector.

11.20.5.4 Output Selector

The output selector is used to select the output data from three different sources: ed1r, e_ad1 or zero. This selection is done for the exponent output data (ED1R), the floating overflow (F_OVFR) and the floating underflow (F_UNFR). The selection is based on the assertion of two control signals, OSEL1_ZERO and OSEL1_E_AD1.

OSEL1_E_AD1 if asserted, selects the output from E_AD1; for overflow and underflow, OSEL1_E_AD1 selects E_AD1_UNF and E_AD1_OVF. If OSEL1_E_AD1 is deasserted, then the output is selected from ED1R; for overflow and underflow, OSEL1_E_AD1 deasserted selects ED1R_OVF and ED1R_UNF. This selection is done using a 2 to 1 selector.

The selection of zero is done prior to the 2 to 1 selector described above. If OSEL1_ZERO is asserted, then the inputs from E_AD1 and ED1R entering the 2 to 1 selector are both forced to zero. Then, since only one select line is used to control the selector, the zero value will be transferred to the output regardless of the assertion of OSEL1_E_AD1.

The output of the selector is latched every cycle and driven into the following stage.

The selection of the exponent output is shown in the following table.

Table 11–16: Exponent Output Selection

Exponent Operation Category	Select ED1R if:	Select E_AD1 if:	Force Zeros if:
E0	always select	—	—
E1	fraction passed unshifted	fraction shifted	DIV: (e1zr + e2zr) EFF.ADDf: (e1zr * e2zr)
E2	fraction passed unshifted	fraction shifted	MUL: (e1zr + e2zr) EFF.SUBf: (e1zr * e2zr)
E3	—	always select	EFF.SUBf, deltaE=0: (f_zr) EFF.SUBf: (e1zr * e2zr)
E4	—	always select	(f_zr)
E5	—	always select	—
E6	—	always select	(e1zr)

As shown in the table above, some selection operations are dependent only on the operation category, while others also depend on whether the fraction adder result needed a one bit normalization. The control section implements the following equation:

$$
\begin{array}{ll}
\mathit{OSEL1\_E\_AD1} = 1 & \mathit{if}\ \mathit{GSEL1\_E\_AD1} \\
\mathit{OSEL1\_E\_AD1} = 0 & \mathit{if}\ \mathit{GSEL1\_ED1R} \\
 & \\
\mathit{OSEL1\_E\_AD1} = \mathit{SHFT\_DONE} & \mathit{if}\ (\overline{\mathit{GSEL1\_E\_AD}} * \overline{\mathit{GSEL1\_ED1R}}) + \mathit{RESET}
\end{array}
$$

GSEL1_E_AD1 and GSEL1_ED1R are generated in the control section, based on the opcode. SHFT_DONE is generated in the adder, based on the value of the adder output. If RESET is asserted, SHFT_DONE selects the exponent output.

The overflow and underflow outputs that are selected are never used.

11.20.5.5 Exponent Datapath Operation Summary (Normal Operating Mode):

Conditions							EDP Inputs			EDP Output
OPC	EFF A/S	EDIFF	E1Z	E2Z	F_Z	SHFT_DONE	K	ED1R	ED2R	ED1R
ADDf,	A	x	0	x	x	0	V	V	x	ED1R
SUBf	A	x	x	0	x	0	V	V	x	ED1R
	A	x	0	x	x	1	V	V	x	E_AD1
	A	x	x	0	x	1	V	V	x	E_AD1
	A	x	1	1	x	x	V	V	x	0
	S	ED=0	0	x	0	x	x	V	V	E_AD1
	S	ED=0	x	0	0	x	x	V	V	E_AD1
	S	ED=0	1	1	x	x	x	V	V	0
	S	ED=0	x	x	1	x	x	V	V	0
	S	ED=1	x	x	x	x	x	V	V	E_AD1
	S	ED>1	x	x	x	0	V	V	x	ED1R
	S	ED>1	x	x	x	1	V	V	x	E_AD1
CMPf	x	x	x	x	x	x	x	V	x	ED1R
TSTf	x	x	x	x	x	x	x	V	x	ED1R
DIVf,	x	x	0	0	x	0	V	V	x	ED1R
MULf,	x	x	0	0	x	0	V	V	x	ED1R
MULL,	x	x	0	0	x	1	V	V	x	E_AD1
CVTff,	x	x	0	x	x	x	V	V	x	E_AD1
MOVf,	x	x	1	x	x	x	V	V	x	0
MNEGf										
CVTfi	x	x	x	x	x	x	V	V	x	E_AD1
CVTif	x	x	x	x	0	x	V	x	V	E_AD1
	x	x	x	x	1	x	V	x	V	0
CVTRfL	x	x	x	x	x	x	V	V	x	E_AD1

11.20.6 Sign Datapath

The operation done in the sign datapath portion of stage 3 is shown in the table below.

Table 11–17: Stage 3 Sign Datapath Operations/sign_dp_oper

Category	Operation	Condition
S0	f_3%s1r_h <– f_2%s1r_h f_3%s2r_h <– f_2%s2r_h	always performed
S1	f_b%s_out_l <– f_3%bp_plsn	performed during stage 4 bypass

11.20.7 Control

The control section generates all the control signals needed for stage 3, based on the opcode and several condition signals, such as E1ZR and F_ZR. It sends the opcode and necessary condition signals to stage 4. In addition, it contains some integer overflow detection logic, a 6-bit adder used in MULL, and logic to generate some control signals needed by stage 4.

The following table shows which categories of operations are performed in the fraction, exponent, and sign datapath portions of stage 3 for each opcode. Each category indicates a unique set of control signals to be driven. The control section generates these combinations of categories.

Table 11–18: Categories of Datapath Operations

Opcode Operation	Categories of Datapath Operations		
	Fraction	Exponent	Sign
CVTff, MOVf, MNEGf	F2	E6	S0
CVTif	F0	E4	S0
CVTfi, right shift	F2	E5	S0
CVTfi, CVTRfL: left shift	F0	E5	S0
CVTRfL, right shift	F4	E5	S0
DIVf	F5	E1	S0
EFF.ADDf	F2	E1	S0
EFF.SUBf, deltaE<2, opnds <> 0	F0	E3	S0
EFF.SUBf, deltaE<2, opnd(s) = 0	F1	E3	S0
EFF.SUBf, deltaE>1	F3	E2	S0
CMPf; TSTf	F2	E0	S0
MULf, MULL,	F6	E2	S0

11.20.7.1 Miscellaneous Control Signals

Most of the stage 3 control signals are generated in the control decoders, but some are generated or conditioned external to the decoders. These signals are described in this section.

11.20.7.2 Data_Valid

The data_valid signal sent to stage 4 is received from stage 2 and is enabled when there is no FBOX flush occurring and a stage 4 bypass is also not occurring. The equation for enabling **F_3_C%3_DATA_VALIDR_H** is as follows:

```
f_3%s3_dv_enb              =    NOT f_1%abort_h AND (f_3%s4_bypass_abortr_h OR
                                NOT ( f_3%s4_bypass_enb AND f_3%s4_bypass_reqr_h ))
```

This operation is performed before the end of the execute cycle.

11.20.7.3 Fault Bits and NEW_FOP

There are three fault signals associated with each valid data that flows through the FBOX pipe. In addition to these three fault signals there is one more signal (new_fop) which indicates that there is a new FBOX operation is coming through the FBOX pipe. The three fault signals are named **F_3%MMGT_FLT_L**, **F_3%MEM_ERR_L**, **F_3%RSV_ADR_L**. A stage 4 bypass request can not be generated if any of the fault lines are asserted. The new_fop signal is cleared out of the FBOX pipe whenever an FBOX purge occurs.

11.20.7.4 Signs_Not_Eql, Fb_Neg4

Stage 3 generates two signals for use in stage 4. These signals are signs_not_eqlr_h and fb_neg4r_h The equation for signs_not_eql is:

$$\mathit{SIGNS\_NOT\_EQL} = \mathit{S1}\ \mathit{XOR}\ \mathit{S2}$$

FB_NEG4 is the signal used to negate the B input to the stage 4 fraction adder. The input is negated if stage4 needs to perform a two's complement. The equation implemented is:

$$\begin{aligned}\mathit{FB\_NEG4} = [&(\mathit{EFFSUB} * \mathit{E\_DIFF\_EQL\_0} * \mathit{F\_2\&F\_N}) + \\ &(\mathit{CVTFI} * \mathit{S1R}) + \\ &(\mathit{SIGNS\_NOT\_EQL})] * \overline{\mathit{F\_I\&FBOX\_BYPASS\_H}}\end{aligned}$$

11.20.7.5 Integer Overflow Logic

Some of the logic used to detect the integer overflow condition for CVTfi and CVTRfL is located in stage 3. This static logic operates unconditionally. and its outputs are used by stage 4 when needed.

The first function is IOVFL3. It implements the equation

$$\begin{aligned}\mathit{IOVFL3} \leftarrow \mathit{S1R} * [&(\mathit{DESTDT{<}BYTE{>}} * \overline{\mathit{MNEGBR}}) + \\ &(\mathit{DESTDT{<}WORD{>}} * \overline{\mathit{MNEGWR}}) + \\ &(\mathit{DESTDT{<}LONG{>}} * \overline{\mathit{MNEGLR}})]\end{aligned}$$

IOVFL3 detects integer overflow for CVTfi and CVTRfL (no round up), in the case where the hidden bit of the fraction becomes the MSB of the integer, and the sign is negative. In this case, a two's complement must be performed on the integer. If the integer is 100...00, no overflow will occur since the result of the two's complement will be 100...00, a negative number. This happens because in N bits, more negative numbers (one more) can be represented using two's complement than positive numbers. Thus, there is no positive equivalent of the most negative number (100...00). If the integer is not 100...00, overflow will occur since the result of the two's complement will be 0XX..XX, a positive number.

The second function is IOVFL4. It implements the equation

$$\begin{aligned}&\mathit{IOVFL4} \leftarrow (\mathit{IOVFL4A} + \mathit{IOVFL4B}) * \mathit{CVTRfL} \\ &\mathit{IOVFL4A} \leftarrow \mathit{LNEGIR} * \overline{\mathit{S1R}} * \mathit{E\_DIFF\_EQL\_25R} \\ &\mathit{IOVFL4B} \leftarrow \mathit{S1R} * \mathit{E\_DIFF\_EQL\_24R} * \mathit{CRFL\_RNDR} * \mathit{MNEGLR}\end{aligned}$$

IOVFL4 detects integer overflow for CVTRfL, in the case when rounding causes the integer to be incremented. IOVFL4A detects the case where the integer is 011...11, the result should be positive, and a round up occurs. IOVFL4B is used to detect a case not covered by IOVFL3. In general, if the hidden bit of the fraction becomes the MSB of the integer and the sign bit is negative, overflow will occur unless the integer is 100...00. However, for CVTRfL, overflow will also occur for an integer equal to 100...00 if the integer must be rounded up. IOVFL4B covers this case.

IOVFL3 and IOVFL4 are sent to stage 4, which calculates the final integer overflow result.

11.20.7.6 Cin_B58

The carry in to bit position <B58> of the fraction adder is generated outside the control decoders, using control signals generated by the decoders. (See Fraction Datapath Operation Summary.)

$$\texttt{CIN\_B58} = \overline{\texttt{F\_1\%F\_N}} \quad \texttt{if operation is DIVIDE}$$

$$\texttt{CIN\_B58} = \overline{\texttt{STKY}} \quad \texttt{if operation is EFFSUB, EDIFF>1}$$

$$\texttt{CIN\_B58} = \texttt{0} \quad \texttt{otherwise}$$

The decoders generate the signals indicating the operation type.

11.20.7.7 Sel_Other

The sel_other signal is used in the adder and output selector in order to permit selection of the normalizer output as the stage output. For all operations except CVTfi, the value of this signal is determined by the operation. For CVTfi, it is determined by the sign of the exponent difference obtained in stage 1. If the exponent difference is negative, a left shift is performed on the fraction, and stage 3 must select the normalizer output. If the exponent difference is positive, a right shift is performed, and stage 3 selects the adder output. (See Fraction Datapath Operation Summary.) Finally, the normalizer output is always selected in FBOX_Test mode and when the chip is reset. The equation implemented is the following:

$$\begin{aligned}\texttt{SEL\_OTHER} = {} & \texttt{(CVTFI * E\_N)} + \\ & \texttt{(}\overline{\texttt{CVTFI}}\texttt{ * F\_3\%SEL\_OTHER\_H)} + \\ & \texttt{F\_1\%FBOX\_BYPASS\_H} + \\ & \texttt{RESET}\end{aligned}$$

The control decoders generate the signal **F_3_C%SEL_OTHER_H**, used for all operations except CVTFI.

11.20.7.8 Left Shifter Input Selection Signals

There are two left shifter input selection signals: **F_3%LSHFT_FD1R_H** and **F_3%LSHFT_FD2R_H**. Either F_2%FD1R or F_2%FD2R may hold the input to be left-shifted. (See Fraction Datapath Operation Summary.) F_2%FD2R holds the input if the operation is effective subtraction, with either input equal to zero. For all other operations, F_2%FD1R holds the input to be shifted. The equations implemented are the following:

$$\begin{aligned}\texttt{LSHFT\_FD1R} = {} & \texttt{[}\overline{\texttt{F\_1\%FBOX\_BYPASS\_H}}\texttt{ * }\overline{\texttt{(EFFSUB * (E1Z + E2Z))}}\texttt{]} + \\ & \texttt{[F\_1\%FBOX\_BYPASS\_H * }\overline{\texttt{F\_1\%S4\_BYPASS\_ENB\_H}}\texttt{]} \\ \texttt{LSHFT\_FD2R} = {} & \texttt{[}\overline{\texttt{F\_1\%FBOX\_BYPASS\_H}}\texttt{ * (EFFSUB * (E1Z + E2Z))]} + \\ & \texttt{[F\_1\%FBOX\_BYPASS\_H * F\_1\%S4\_BYPASS\_ENB\_H]}\end{aligned}$$

11.20.7.9 Osel1_Zero

This signal is used to force the stage 3 exponent output to zero. The equation implemented is as follows (see description of the output selector in the exponent section):

```
OSEL1_ZERO = {[DIV    * (E1Z + E2Z)] +
              [MUL    * (E1Z + E2Z)] +
              [EFFADD * (E1Z * E2Z)] +
              [EFFSUB * (E1Z * E2Z)] +
              [EFFSUB * E_DIFF_EQL_0 * F_Z] +
              [CVTif  * F_Z] +
              [CVTff  * E1Z] +
              [MOVf   * E1Z] +
              [MNEGf  * E1Z]}  * F_I&FBOX_BYPASS_H
```

11.20.7.10 Osel1_Ed1r

This signal is used to select the stage 3 exponent output. If it is asserted, the contents of F_SG2%ED1R are chosen as the stage output; otherwise, the exponent adder output is chosen as the stage output.

```
OSEL1_ED1R = 0           if [(EFFSUB * E_DIFF_GTR_1) +
                              CVTff  -
                              MOVf   +
                              MNEGf  +
                              CVTfi  +
                              CVTif  ] * F_I&FBOX_BYPASS_H  * RESET
OSEL1_ED1R = 1           if [ CMPf  +
                              TSTf  +
                              F_I&FBOX_BYPASS_H ] * RESET
OSEL1_ED1R = SHFT_DONE   if [(EFFSUB * E_DIFF_GTR_1) +
                              EFFADD  +
                              MUL     +
                              DIV     ] * F_I&FBOX_BYPASS_H +
                              RESET
```

11.20.7.11 MULL Adder

The multiplier array in stage 2 generates 64-bit sum and carry vectors for MULL. The 58 MSB's are combined in the fraction adder in stage 3. The 6 LSB's (<B58:B63>) of each vector must be added together in the control section of stage 3. The six sum bits generated are sent to stage 4 (as are the MSB sum bits). Any carry out of the six LSB's has been previously incorporated in the MSB's in stage 2.

11.21 STAGE 4

Stage 4 of the pipe is used to do various terminal operations of an instruction. It does round or a 2's complement on the result of stage 3. The result of stage 4 is the final result which is sent to the interface section. Stage 4 finds the sign of the final floating result and outputs it to the interface. Stage 4 also detects the following conditions: integer overflow, floating overflow, floating underflow, zero result, negative result, reserved operand and floating divide by zero. In addition to this, it sets the correct condition codes (PSL.Z AND PSL.N). Stage 4 also checks whether the condition for CMP and TST instruction is met or not. For CMP, the correct condition codes are set. During any CMP instruction, stage 4 forces the fraction and exponent datapath output to zero. When reset is asserted only one path of selector will be enabled in the fraction adder selector logic.

11.22 FRACTION DATAPATH

Figure 11–28: Fraction Datapath Block Diagram

```
                                     F_B%F_OUT_L<B1:B58> ||    || F_3%FD1R<A0:B58>
                                                         \/    \/
+--------------------------------------------------------------------------------------------+
|            ZERO DETECTION LOGIC/MULL OVERFLOW LOGIC    ||                                  |--> TO CONTROL
---------------------------------------------------------------++----------------------------
                                                         ||    ||                 |<--------- F_3%MILSBR<5:0>
                                                         \/    \/                 V
----------------------------------------------------------------------------------------------
|            ADDER INPUT SELECTOR AND NEGATION           ||                                  |<---SEL_MULL
|                                                        ||                                  |<---FB_NEG
----------------------------------------------------------------------------------------------
                                                         ||    ||
                                         || ZERO         ||    || F_4_A%BIN_H
                                         \/              \/    \/
----------------------------------------------------------------------------------------------
|                                        FA              ||    FB                            |<- RNDf,g,dl
|                                     61-BIT ADDER       ||                                  |<- CINB55,CIN_B58
|                                                        ||                                  |<- SEL_OTHER,SHFT_EN
|                                                        ||                                  |-> SHFT_DONE (TO EXP.)
----------------------------------------------------------------------------------------------
    |            |             ||                        ||                            |
    |            |             ||------------------------||-------> SUM<b48>,SUM<b40>, | SUM<b24> (TO MISC LOGIC)
    |            |             \/ SUM  (ADDER RESULT)    \/                            |
+---+------------+---------------------------------------------------------------------+-----+
|   |            |                  1-BIT RIGHT SHIFTER  ||                            |     |
+---+------------+---------------++-------------++-------------------------------------+-----+
    |DET_SHR1    | DET_PASS      ||             ||       ||                            |SEL_OTHER
    |            |               ||             ||       ||    || ZERO                 |
    V            V               \/ RESN        \/ RESS  \/    \/                      V
+--------------------------------------------------------------------------------------------+
|                                   RSELECTOR            ||                                  |
+------------------------------------++------------------------------------------------------+
                                     ||                  ||
                                     \/                  \/
+--------------------------------------------------------------------------------------------+
|                              FD1R<B0:B55>              ||                                  |<-- PHI_2
+------------------------------------++------------------------------------------------------+
                                     || F_4%FD1R         ||
                                     \/                  \/
+--------------------------------------------------------------------------------------------+
|                                                                                            |<-- PHI_3
|                                        BUS DRIVERS                                         |<-- DATA_VALID
|                                                                                            |<-- F_UNFR_H
+--------------------------------------------------------------------------------------------+
                                                         ||
                                                         \/ F_B%F_OUT_L<B1:B58> ( To output interface.)
```

11.22.1 Fraction Implementation Description

FRACTION DETECTION LOGIC

The detection logic in the fraction datapath is connected directly to the output from stage 3. The F_3%FD1R_H and F_3%MILSBR_H outputs from stage 3 are necessary for the detection logic. The detection logic works unconditionally and no control signals are provided to the logic except clocks. The detection logic detects a zero result for MULL and CVTfi instructions. It also detects overflow for the MULL instruction. Overflow for MULL occurs when the 32 msb's of the 64-bit result are not equal to the sign extension of the low half (32 lsb).

SELECTOR

The selector drives the selected input into the adder. The selector either selects F_3%FD1R_H unshifted or shifted left by eight bits. It can also negate the selected input. The control input to the selector is SEL_MULL_L, SEL_MULL_H, FB_NEGR_H, and FB_NEGR_L. If the SEL_MULL_H is high then it is a MULL instruction and the F_3%FD1R_H and F_3_C%MILSBR_H is selected, shifted left by eight bits. If SEL_MULL_H is low, then the F_3%FD1R_H is selected without any shifting. If FB_NEGR_H is high then the selected input is complemented. The complementing is necessary for doing a 2's complement if certain conditions are satisfied for EFFSUB and CVTfi instructions.

ADDER

The adder is used for the terminal operation of the result, i.e. for rounding, to find the 2's complement of the result and to add zero to the input. The last case is used when the input to stage 4 is to be passed as output of stage 4. The adder also drives the result selection signals. One input (FB) to the adder is F_4_A%BIN_H and the other input (FA) is always zero. The RND*, CIN_B58 and CINB55_ONE signals are driven to the adder by the control of stage 4.

SHIFT DETECTION LOGIC OF ADDER

If enabled, the adder examines the sum bits <A0:B1> to determine whether a one bit shift right is needed to normalize the result. The instructions which may require a one bit right shift are: EFF.ADDf, EFF.SUBf, MULf, DIVF, CVTif and CVTff. For all these instructions the result from stage 4 fraction adder could be of the form 0.1XX.., 0.00..., or 1.XX.. .

If the shift detection logic is disabled, then the signal indicating "no shift needed" will be forced valid. This logic is also conditioned with another signal, which is used to force all of the shift detection signals to their invalid value. Since these shift detection signals are used to drive the output selector for the stage, this feature permits the selection of a stage output other than the shifted or unshifted adder result.

The logic used to control the shifting is as follows:

$$\texttt{f\_*\_a\%det\_shrl\_h = A0 * shift\_en * sel\_other}$$

$$\texttt{f\_*\_a\%det\_pass\_h = \{[(}\overline{\texttt{A0}}\texttt{*B0 + }\overline{\texttt{A0}}\texttt{*}\overline{\texttt{B0}}\texttt{*}\overline{\texttt{B1}}\texttt{) * shift\_en] + }\overline{\texttt{shift\_en}}\texttt{\} * }\overline{\texttt{sel\_other}}$$

DET_SHR1 detects the case in which the fraction result is 1.XX..XX, and thus the fraction must be shifted right by one bit to be normalized. DET_PASS detects several cases: first, the case in which the fraction result is 0.1XX..XX; second, the case in which the fraction result is zero (0.00XX..XX); last, the case in which the shifter is disabled.

1-BIT RIGHT SHIFTER

The input to the 1-bit shifter is the adder result. The output of the 1-bit shifter is the adder result unshifted (RESN) and the adder result shifted right by 1-bit (RESS). The 1-bit shifter works unconditionally. The shifter is used to right shift a fraction overflow result in case of fraction overflow for floating point instructions. If fraction overflow has occured then the shifted result is used, otherwise the unshifted result is used.

RSELECTOR

The RSELECTOR selects the final result for an instruction. The output of the selector is latched in PHI_2 which is passed to the interface. The inputs to the RSELECTOR are the two outputs from the 1-bit shifter and zero. For the CMP instruction and for floating destination type instructions if the final result is zero then it selects zero. For all other instructions the selector selects the 1-bit right shifter output (RESN or RESS).

BUS DRIVERS

The BUS DRIVER section drives the final stage 4 result to the output interface on an active-low precharged bus, F_B%F_OUT_L<B1:B55>. This bus is shared with stage 3 which uses it to bypass stage 4 for certain instructions. The input to the BUS DRIVER section is F_4%FD1R_H<B1:B55>. During PHI_3, if stage 4's data_valid bit is set and the underflow condition is not detected, the inverted value of F_4%FD1R_H<B1:B55> is driven onto the bus. If underflow is detected then the bus is not driven. This represents a zero being driven to the output interface. The fraction sign bit (S1R),the PSL.N bit, and the exponent data bits are all driven to the output interface in the same manner.

11.22.2 Fraction Operation

The operations performed in the fraction datapath are shown in the table below.

Table 11–19: Fraction Datapath Operations

Condition	Floating Operation	ADDER
		SHIFT_EN
EFF_SUB AND FN=1 AND DeltaE=0	FD1R <– 0 + NOT FB + 1	Y
EFF_SUB AND FN=0 AND DeltaE=0	FD1R <– 0 + FB	Y
EFF_ADD OR (EFF_SUB AND NOT DelatE=0)	FD1R <– 0 + FB + Rndx	Y
MULf	FD1R <– 0 + FB + Rndx	Y
DIVf	FD1R <– 0 + FB + Rndx	Y
CVTif	FD1R <– 0 + FB + Rndx	Y
CVTff/MOV	FD1R <– 0 + FB + Rndx	Y
MNEG instruction	FD1R <– 0 + FB	N
CVTfi AND S1R=0	FD1R <– 0 + FB	N
CVTfi AND S1R=1	FD1R <– 0 + NOT FB + 1	N
CMP/TST and PIPELINED CMP inst.	FD1R <– 0	N
MULL	FD1R <– 0 + FB	N

Table 11–20: Fraction Datapath Operation Summary

	Conditions				Inputs		Output
OPC	EDIFF Value	E1Z	E2Z	F_Z	FD1R	MILSBR	FD1R
E_A	X	0	X	X	V	X	SUM
E_A	X	X	0	X	V	X	SUM
E_A	=0	1	1	X	X	X	0
E_S	=0	X	X	0	V	X	SUM
E_S	=0	X	X	1	X	X	0
E_S	=0	1	1	X	X	X	0
E_S	>0	0	X	X	V	X	SUM
E_S	>0	X	0	X	V	X	SUM
MULf	X	0	0	X	V	X	SUM
MULf	X	1	X	X	X	X	0
MULf	X	X	1	X	X	X	0
DIVf	X	X	0	X	V	X	SUM
DIVf	X	X	1	X	X	X	0
CVTif	X	X	X	0	V	X	SUM
CVTif	X	X	X	1	X	X	0
CVTff	X	0	X	X	V	X	SUM
CVTff	X	1	X	X	X	X	0
MOV/N	X	0	X	X	V	X	SUM
MOV/N	X	1	X	X	X	X	0
CVTfi	X	0	X	X	V	X	SUM
CVTfi	X	1	X	X	X	X	0
MULL	X	X	X	X	V	V	SUM
CMP	V	V	V	V	X	X	0

```
E_A/E_S = Eff add/Eff substarct.
MOV/N   = MOV/MNEG instruction
0       = Zero result.
X = Don't care
V = Valid
```

The "=0" and ">0" under the EDIFF value column for E_A or E_S refers to the exponent difference value being equal to zero or greater than zero respectively.

11.23 EXPONENT DATAPATH

Figure 11–29: Block Diagram of Exponent Processor

```
                        F_3_E%ED1R_H | |   K | |   | | F_B%E_OUT_L<10:0>                 | | F_3_E%ED2R_H -------
                                     \ /     | |   \ /                                   \ /              F_4_EA1
                           +-----------------+-+------------------------------------------------+
F_4_C%ENA_E1Z_L, --------->|                       | |  ZERO DETECTION                          |------> F_4_E%E1ZR_
F_4_C%ENA_E2Z_L            +---------+-+-----+-+-----------------------------------------+-+----+        F_4_E%E2ZR_
                        F_3_E%ED1R_H | |   K | |   | |                                   | | F_3_E%ED2R_H
                                     \ /     \ /   \ /                                   \ /

                           +--------------------------------------------------------------------+
F_4_C%ISEL1_ED1R_A_H/L -->|                       | |                                          |
F_4_C%ISEL1_K_A_H/L ----->|                       | |  INPUT SELECTOR                          |
F_4_C%ISEL1_ED2R_B_H/L -->|                       | |   FOR  ADDER 1                           |
F_4_C%ISEL1_K_B_H/L ----->|                       | |                                          |
                           +---------+-+-----+-+---------------------+-+---------+-+-----+-+----+
                        F_3_E%ED1R_H | |   K | |   | |        INP_1A | |  INP_1B | |     | | F_3_E%ED2R_H
                                     | |     | |   \ /               \ /         \ /     | |
                           +---------+-+-----+-+---------------------+-+---------+-+-----+-+----+
F_4_C%INVERT_EA_AD1_H/L ->|                       | |  COMPLEMENT LOGIC                        |
F_4_C%INVERT_EB_AD1_H/L ->|                       | | FOR EXPONENT ADDER 1                     |
                           +---------+-+-----+-+---------------------+-+---------+-+-----+-+----+
                        F_3_E%ED1R_H | |   K | |   | |        EA_AD1 | | EB_AD1 | |      | | F_3_E%ED2R_H
                                     | |     | |   \ /               \ /        \ /      | |
                           +--------------------------------------------------------------------+
F_4_C%CIN_E_AD1_H -------->|                       | |  EXPONENT ADDER 1                        |
                           +---------+-+-----+-+-----------------------------------------+-+----+
                        F_3_E%ED1R_H | |   K | |   | |   | | E_AD1                       | | F_3_E%ED2R_H F_4_EA1
                      ---------------| |-----------| |---| |---------------------------------------------------
                                     \ /           \ /   | |                                              F_4_EOS
                           +-----------------------------+-+------------------------------------+
K_F%PHI_4_H ------------->|                       | |   | | PHI4 LATCHES                       |
                           +---------+-+-----------------+-+------------------------------------+
                         F_4_E%ED1_H | |           | |   | | E_AD1
                                     \ /           \ /   \ /
                           +--------------------------------------------------------------------+
F_4_C%EN_G_TYPE_L  ------->|                       | |  OVERFLOW AND                            |
F_4_C%EN_FD_TYPE_L ------->|                       | |    UNDERFLOW                             |
F_4_C%OSEL1_ZERO_H ------->|                       | |  DETECTION LOGIC                         |
                           +---------+-+-----------------+-+-----------------+-+----------------+
                         F_4_E%ED1_H | |           | |   | | E_AD1           | | F_OVF1, F_OVF2
                                     \ /           \ /   \ /                 \ /
                           +--------------------------------------------------------------------+
F_4_C%OSEL1_ZERO_L  ----->|                       | |  OUTPUT SELECT                           |
F_4_C%SHFT_DONE_H   ----->|                       | |  AND PHI2 LATCHES                        |
                           |                       | |                                          |
                           +---------+-+-------------------------------------+----+-------------+         F_4_EOS
                      ---------------| |-----------| |-----------------------|    |------------------------------
 F_UNFR_H                            | |           | |                       |    |
<-------------------+----------------|-|-----------|-|-----------------------|    |
 To FRAC. D.P.      |   F_4_E%ED1R_H \ /           \ /             F_UNFR_H  V    V  F_OVFR_H
                    |      +--------------------------------------------------------------------+
                    +----->|                                                                    |
         PHI_3 ----------->|                            BUS DRIVERS                             |
         DATA_VALID ------>|                                                                    |
                           +--------------------------------------------------------------------+
                                                   | |                       |    |
                                                   \ / F_B%E_OUT_L<10:0>     V    V  F_4%FOV_OUT_L
                                                                        F_4%FUN_OUT_L
```

11.23.1 Exponent Block Description

The exponent block can be used for various functions. In stage 4 it is used to increment the stage 3 exponent result. It is also used to detect the floating underflow and floating overflow conditions on the final result. The zero detector result is used for CVTfi overflow detection logic. The final exponent result is either the stage 3 result, or the stage 3 result incremented by one, (if there is overflow) or zero. As the selection of the final result is done near the end of a cycle, floating overflow and underflow are computed for all possible results and the correct one is chosen with the result.

11.23.2 Exponent Operation

In the exponent datapath, the stage 3 exponent result is incremented unconditionally for each instruction. Then, depending on the instruction and the fraction result, the correct exponent is selected. The three possible exponent results are: the stage 3 exponent result, the stage 3 exponent result incremented by one, and zero. For instructions having integer as the final output, the exponent is a don't care.

11.23.3 Floating Overflow and Underflow Detection

Floating point overflow and underflow is detected on the output of the exponent adder as well as the exponent data (ED1R).

Floating point overflow requires detecting a case when the exponent is larger than the largest biased exponent of 255 for F and D, and 2047 for G. The overflow is detected as follows, where e<12:0> represents the exponent:

$$\texttt{For F and D: Overflow = }\overline{\texttt{e<12>}}\texttt{ * (e<11> + e<BITMAP>(10) + e<9> + e<8>)}$$

$$\texttt{for G: overflow = }\overline{\texttt{e<12>}}\texttt{ * e<11>}$$

The floating overflow signals, ED1R_OVF and E_AD1_OVF, are only asserted if an overflow is detected and the appropriate enable signal is asserted. The enable signals are en_fd_type_l and en_g_type_l, they signal whether a floating point operation is being performed and what the data type is.

Floating point underflow requires detecting the case when the exponent is smaller than the minimum exponent. Since the smallest biased exponent is 1 for F, D and G, the following logic detects underflow:

```
for F,D and G:

underflow = e<12> + NOR (e<0> to e<12>), which reduces to,
          = e<12> + NOR (e<0> to e<11>)
```

As with overflow, the underflow signals, ED1R_UNF and E_AD1_UNF, are asserted only if an underflow is detected and one of the enable signals is asserted.

The overflow and underflow signals are selected as described in the output selector section.

11.23.4 Output Selector

The output selector is used to select the output data from three different sources: ed1r, e_ad1 or zero. This selection is done for the exponent output data (ED1R), the floating overflow (F_OVFR) and the floating underflow (F_UNFR). The selection is based on the assertion of two control signals, OSEL1_ZERO and SHFT_DONE.

SHFT_DONE, if asserted, selects the output from E_AD1; for overflow SHFT_DONE selects E_AD1_UNF and E_AD1_OVF. If SHFT_DONE is deasserted, then the output is selected from ED1R; for overflow and underflow, SHFT_DONE deasserted selects ED1R_OVF and ED1R_UNF. This selection is done using a 2 to 1 selector.

The selection of zero is done prior to the 2 to 1 selector described above. The selection for the exponent result is done as follows. If the final result is know to be zero then a zero result is selected. The PSL.Z bit (see below under miscellaneous logic) is asserted if the final result is zero, which asserts OSEL1_ZERO. If OSEL1_ZERO is asserted, then the inputs from E_AD1 and ED1R entering the 2 to 1 selector are both forced to zero. Then, since only one select line is used to control the selector, the zero value will be transferred to the output regardless of the assertion of SHFT_DONE.

The output of the selector is latched during PHI_2 of every cycle and driven to the BUS DRIVER section.

BUS DRIVERS

The BUS DRIVER section drives the final stage 4 result to the output interface on an active-low precharged bus, F_B%E_OUT_L<10:0>. This bus is shared with stage 3 which uses it to bypass stage 4 for certain instructions. The input to the BUS DRIVER section is F_4_E%ED1R_H<10:0>. During PHI_3, if stage 4's data_valid bit is set and the underflow condition is not detected, the inverted value of F_4_E%ED1R_H<10:0> is driven onto the bus. If underflow is detected, the bus is not driven. This represents a zero being driven to the output interface.

Table 11–21: Exponent Datapath Operation Summary

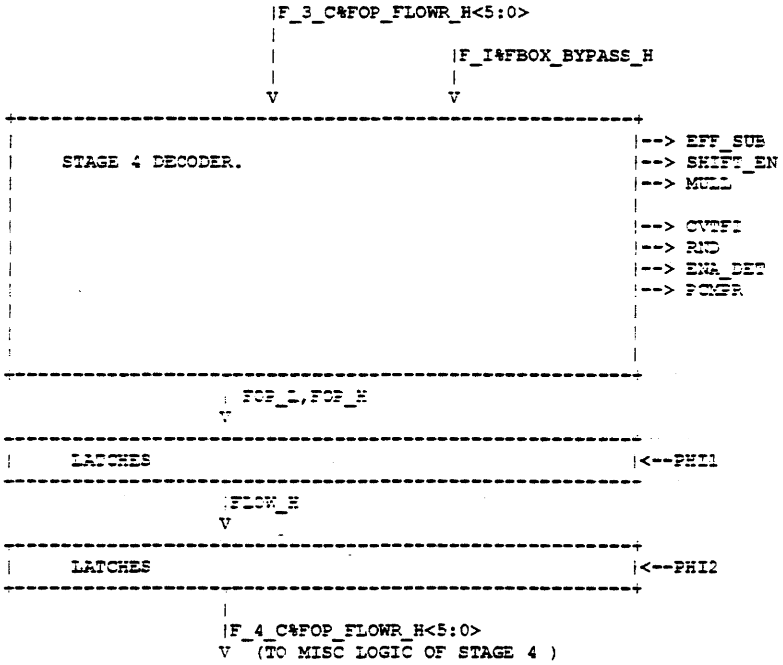
	Conditions				Input	Output
OPC	EDIFF Value	E1Z	E2Z	F_Z	ED1R	ED1R
E_A	X	0	X	X	V	V
E_A	X	X	0	X	V	V
E_A	=0	1	1	X	X	0
E_S	=0	X	X	0	V	V
E_S	=0	X	X	1	X	0
E_S	=0	1	1	X	X	0
E_S	>0	0	X	X	V	V
E_S	>0	X	0	X	V	V
MULf	X	0	0	X	V	V
MULf	X	1	X	X	X	0
MULf	X	X	1	X	X	0
DIVf	X	X	0	X	V	V
DIVf	X	X	1	X	X	0
CVTif	X	X	X	0	V	V
CVTif	X	X	X	1	X	0
CVTff	X	0	X	X	V	V
CVTff	X	1	X	X	X	0
MOV/N	X	0	X	X	V	V
MOV/N	X	1	X	X	X	0
CVTfi	X	X	X	X	X	X
MULL	X	X	X	X	X	X
CMP	X	X	X	X	X	0

```
E_A/E_S - Eff add/Eff substarct
MOV/N   - MOV/MNEG instruction
X - Don't care
V - Valid
```

The "=0" and ">0" under the EDIFF value column for E_A or E_S refers to the exponent difference value being equal to zero or greater than zero respectively.

11.24 Control

Figure 11–30: Control Block Diagram



11.24.1 Control Block Description

The control block supplies all the control signals for various operations in stage 4 and also sends the control information to interface delayed by a cycle. The control block gets it's input from stage 3.

11.24.2 Control Block Implementation

The main control is implemented with a PLA. The inputs to the PLA are the opcode and bypass signals. All the instruction information is encoded in FOP_FLOWR_H. The following control information is decoded in the PLA: EFF_SUB, SHIFT_EN, MULL, CVTFI, RND, ENA_DET and PCMPR. SHIFT_EN is asserted for CVTif, CVTDF, ADD/SUB, DIVf, MULf. RND is asserted for CVTif, CVTff, ADD/SUB, DIVf, MULf. ENA_DET is asserted for CVTff, ADD/SUB, DIVf, MULf, CVTff, CVTif.

The destination data type is decoded to get six signals for each datatype. They are: FTYPE,DTYPE, GTYPE, BYTE, WORD and LONG.

The logic used to generate other control signals in stage 4 are as follows:

```
                                          ____________________________
rnd_en_h       = fop_flowr_h<2> * rnd_h * (eff_sub AND e_diff_eql_0r)
fop_flowr_h<2> : If this signal is low for instruction in pipelined mode requiring rnd,
```

then rnd is disabled, i.e. it is a truncate mode.

```
rndf_h        = rnd_en_h * ftype
rnddi_h       = rnd_en_h * dtype
rndg_h        = rnd_en_h * gtype
cin_b58_h     = e_diff_eql_0r_h * eff_sub * f_nr_h
cinb55_one_h = (cvtfi_h * slr_h) + (signs_not_eqlr_h)

  fb_neg_h is generated by stage 3 and sent as fb_neg4r_h to stage 4. The equation
implemented in stage 3 is :
                                                              ___________________
                             fb_neg_h     = (cin_b58_h + cinb55_one) * f_i%fbox_bypass_h

sel_other_h  = pcmpr OR pslz_f_h
pslz_f_h     : This signal will be high if the result for a floating destination result
                  is 0 and if for a CMP instruction if both the operands are same.
```

11.25 MISCELLANEOUS AND SIGN LOGIC

Figure 11-31: Miscellaneous Pla Block Diagram

```
     |F_3_C%FOP_FLOWR_H<5:0>
     |    | E_NR  |F_ZR|E_DIFF_EQL_0     |SIGNS_NOT_EQL
     |    | | F_NR|    |  |E22|E12|S1|S2|
     |    | |     |    |  |   |   |  |  |
     V    V V     V    V  V   V   V  V  V
+-----------------------------------------------------+
|                                                      |--> PSLZ_F_H
|           MISC PLA                                   |
|                                                      |--> PSLN2_F_H
|                                                      |
|                                                      |--> PSLN1_F_H
|                                                      |
|                                                      |--> RESERVED_OPD
|                                                      |
|                                                      |--> F_DIV_BY_0
|                                                      |
+-----------------------------------------------------+
```

11.25.1 Miscellaneous Sign Logic Implementation

Stage 4 is used to find the sign of the final result, condition codes and exceptions. Specifically it does sign computation, integer overflow detection, zero result detection, negative result detection, reserved operands and floating divide by zero detection by utilizing the information provided from the previous stages of the pipe. If the result is zero, stage 4 will force its output to zero. In the case of floating underflow, the sign, PSLN_F_H, fraction, and exponent of the result are forced to zero.

11.25.2 Sign and Negative Result Logic

The sign of the final result and the PSL.N status bits are the same except for CMP and TST instructions. For CMP and TST instructions, the sign bit is a don't care and the PSL.N bit is high if the first operand is strictly less than the second operand. If the final result is a zero then the sign bit and the PSL.N bit should be forced to zero. The PSL.Z (see below) bit is set if the result is zero, which is used to force the sign and the PSL.N bit to zero. For the integer instruction the sign is already in the result, and hence sign is computed only for floating results. Hence the PSL.N bit for floating result PSLN_F_H is same as the sign bit. The signals, PSL.N and PSL.Z, are driven to the output interface on the active-low precharged bus which is shared with stage 3. During PHI_3, if stage 4's data_valid bit is set and the underflow condition is not detected, the inverted value of PSL.N is driven onto the bus. The inverted value of the PSL.Z signal is also driven onto the bus during PHI_3, if the data_valid bit is set, regardless of the underflow condition. The interface uses these signals to determine if the CMP condition is met or not.

The PSL.N bit is obtained as below.

```
                If PSL.Z then PSL.N = 0
```

For EFFADD/EFFSUB the PSL.N bit of the result is given as follows.

```
PSL.N = eff_add * s2r +

        eff_sub * [$\overline{e\_diff\_eql\_0}$ * (e_n*s2r + $\overline{e\_n}$ * $\overline{s2r}$)

              + e_diff_eql_0  * ( f_n * s2r + $\overline{f\_n}$ * $\overline{s2r}$)]
```

For MULf and DIVf the PSL.N bit of the result is the XOR of the sign of the input operands.

```
      PSL.N = signs_not_eql * (MULf + DIVf)
```

For MOV, CVTff and CVTif the PSL.N bit of the result is the sign of the input operand. For MNEG instruction the PSL.N bit of the result is the inverse of the sign of the input operand.

```
PSL.N = slr * (MOV + CVTff + CVTif) + slr * MNEG

For CMP and TST instruction the PSL.N bit is

PSL.N = [signs_not_eql*slr +

        signs_eql*{ e_diff_eql_0  * (f_n XOR slr) * $\overline{f\_z}$ +

        $\overline{e\_diff\_eql\_0}$ * (slr XOR e_n) }] * (CMP + TST)
```

All the above computations are done in the miscellaneous PLA. As the number of minterms for psln logic was large, two signals are generated in the PLA, which are OR'ed outside and AND'ed with PSLZ_F_H, to give the final PSLN_F_H. Sign has to be computed only for instructions considered above. For all the above instruction the final sign is either the PSL.N bit or it is a don't care, hence

```
F_4%S1R_H = PSLN_F_H
```

For CVTfi and MULL, the PSL.N bit is the MSB of the final result. For MULL and CVTfi (destination long), the MSB is SUM<B24>. For CVTfi with destination of word the MSB is SUM<B40> and with destination of byte the MSB is SUM<B48>. Also when the destination is byte and word, the only instruction possible is CVTfi. Hence the PSL.N bit is

```
PSL.N = SUM<B24> * (LONG * CVTfi + MULL ) +

        SUM<B40> * WORD + SUM<B48> * BYTE
```

11.25.3 Integer Overflow

Integer overflow is possible for MULL and CVTfi instructions. The overflow condition for the convert floating to integer instruction is determined in stages 1, 3 and 4 of the pipe. For MULL instruction, the overflow is determined in stage 4. All these conditions are combined to give the integer overflow signal to the interface stage.

OVERFLOW DETECTION FOR CVTfi

The CVTfi instruction overflow detection operation performed in all the stages is given below. All constants are in decimal.

Let the exponent of input operand be E1, then the actual exponent of the floating number is E1-bias. Let that number be ACTUAL_EXP.

hence,

```
actual_exp   = E1 - 128  ; for F and D type
actual_exp   = E1 - 1024 ; for G type
```

Let DEST_LEN equal the length, in bits, of the destination result

hence,

```
dest_len = 8  ; for convert floating to byte instruction
dest_len = 16 ; for convert floating to word instruction
dest_len = 32 ; for convert floating to longword instruction
```

For convert from floating to integers of length 8(B), 16(W), 32(LW) integer overflow occurs under the following condition.

```
1. if actual_exp > dest_len
2. if actual_exp = dest_len and slr=0
3. if actual_exp = dest_len and slr=1 and the integer portion
 is not equal to the most negative number

4. for CVT rounded to long only, in addition to the above conditions the
following conditions have to be checked:

a) if actual_exp = 31 and slr = 0 and the 32 bits of the integer part
are of the form 01111...111 and the remaining fraction is greater than
or equal to 0.5.

b) if actual_exp = 32 and slr = 1 and the 32 bits of the integer part
is of the form 10000...000 and the remaining fraction is greater than
or equal to 0.5.
```

The actual detection of the above conditions are done in stages 1, 3 and 4.

In stage 1 the following signals are generated.

```
lnegir          =  Least negative integer; high if <B0:B31> of F_I%FD1R_H are 1
mnegbr          =  Most negative byte; high if <B1:B7> of F_I%FD1R_H are 0
mnegwr          =  Most negative word; high if <B1:B15> of F_I%FD1R_H are 0
mnegbr          =  Most negative longword; high if <B1:B31> of F_I%FD1R_H are 0
crfl_rndr       =  Convert floating to longword round bit; <B32> of F_I%FD1R_H
e_diff_eql_24r  =  exponent difference equals 24
e_diff_eql_25r  =  exponent difference equals 25
```

In stage 3 an exponent difference (see below) is done to determine the first three conditions for CVTfi overflow. The fourth condition for CVTRfL is also determined in stage 3.

Let E1 be the exponent of the incoming operand in stage 3 and ER be the result of the subtraction in stage 3. ER is send to stage 4.

```
ER  = (bias + dest_len) - E1
    = constant - E1
```

Above Constant Values

F,D --> B	C=136
F,D --> W	C=144
F,D --> L	C=160
G --> B	C=1032
G --> W	C=1040
G --> L	C=1056

Stage 3 sends out two signals, IV3 and IV4, to stage 4 for CVTfi overflow detection. They are generated as follows.

```
iv3 =  slr * $\overline{mnegbr}$ for dest_len = 8(f-->BYTE)

     + slr * $\overline{mnegwr}$ for dest_len = 16(f-->WORD)

     - slr * $\overline{mneglr}$ for dest_len = 32(f-->LONGWORD)

iv4 = (lneglr * $\overline{slr}$ * e_diff_eql_25
      + mneglr * slr * crfl_rndr * e_diff_eql_24) * (CVTRfL)
```

In stage 4 the following operations are performed. Let,

```
e1<12:0>  = exponent result from stage 3 substraction
e_n       = the sign bit of e1 i.e. e1<12>
elz       = 1, e1<12:0> is zero
```

The first two conditions is determined as

```
cvtfi_ovfl12 = e_n + elz * slr
```

The third condition for CVTfi overflow is determined as:

```
cvtfi_ovfl3  = elz * iv3
```

The fourth condition for overflow is given by iv4. Finally the CVTfi overflow is determined as

```
cvtfi_ovfl = (cvtfi_ovfl12 + cvtfi_ovfl3 + iv4) * ( CVTfi)
```

OVERFLOW DETECTION FOR MULL

For MULL integer overflow occurs if the high half of the double length result is not equal to the sign extension of the low half. The following condition is determined on MULL result to detect integer overflow. The register F_3%FD1R_H<B0....B32> contains the high 33 bits of the MULL 64-bit result.

```
mull_zero  = NOR OF BITS fd1r(B0) THROUGH fd1r(B32) ;33 BITS

mull_one   = AND OF BITS fd1r(B0) THROUGH fd1r(B32) ;33 BITS

mull_ovf   = [$\overline{mull\_zero}$ * $\overline{mull\_one}$ ] * MULL
```

The integer overflow is defined as:

```
I_VR_H = cvtfi_ovfl + ( mull_ovf * f_iounfr_h)
```

11.25.4 Zero Result

When the final result is zero then the zero flag (PSL.Z) bit has to be set. Different instructions are analyzed.

For EFFADD/EFFSUB a zero result is possible when a) Both the input operands are equal and it is an effective SUB operation or b) Both the input operands are zero.

```
PSL.Z  = (eff_sub * f_z * e_diff_eql_0) + [(e1_z * e2_z)](ADD+SUB)
```

For a floating multiply instruction, zero result is possible only when one or both the input operands is zero.

```
PSL.Z   = (e1_z + e2_z) * MULf
```

For a floating divide instruction, a zero result is possible only when the dividend is zero. i.e. the second operand is zero. When the first operand, the divisor, is zero then it is floating divide by zero. When a floating divide by zero occurs then the PSL.Z bit is a don't care.

```
PSL.Z   = e2_z * DIVf
```

For MOV/MNEG/CVTff instructions zero result is possible only when the input operand is zero.

```
PSL.Z   = e1_z * (MOV + MNEG + CVTff)
```

For CMP/TST instruction zero flag has to be set when operand 1 is equal to operand 2. For the TST instruction operand 2 is zero.

```
PSL.Z   = (signs_eql *  e_diff_eql_0  * f_z)*(CMP + TST)
```

For convert integer to floating instructions the result is zero if the input integer is zero.

```
PSL.Z   = f_z * CVTif
```

All the above computation is done in the miscellaneous PLA. The output of the miscellaneous PLA is PSLZ_F_H, as only the PSL.Z bit for floating instruction was considered.

For integer multiply instructions and all convert floating to integer instructions, zero result is possible for many different input operands. Hence the final result will be checked for zero result. For the CVTfi instruction, stage 4 is used to do a 2's complement. The 2's complement of zero is again zero, and the 2's compliment of any non-zero number will not be zero. Hence the zero condition can be detected at the input of stage 4 rather than at its output. For MULL the low order 32 bits of the result need to be checked for zero result. The register MILSBR has the 6 low bits of 32-bit lsbs and register FD1R<B32:B57> has the other 26 bits of the 32-bit lsb result. The conditions which are generated are as follows:

```
f_4_d%zero_mil_h    = NOR of f_3%fd1r_h<B56:B57> * NOR of f_3_cmilsbr_h<5:0>
f_4_d%zero_byt_h    = NOR of f_3%fd1r_h<B48:B55>
f_4_d%zero_wor_h    = NOR of f_3%fd1r_h<B40:B47>
f_4_d%zero_mul_h    = NOR of f_3%fd1r_h<B32:B39>
f_4_d%zero_lon_h    = NOR of f_3%fd1r_h<B24:B31>
```

The zero detection is done as follows,

```
zero_b     = NOR OF FD1R(B48) THROUGH FD1R(B55)  ;8 bits
           = f_4_d%zero_byt_h

zero_w     = NOR OF FD1R(B40) THROUGH FD1R(B55)  ;16 bits
           = f_4_d%zero_byt_h * f_4_d%zero_wor_h

zero_l     = NOR OF FD1R(B24) THROUGH FD1R(B55)  ;32 bits
           = zero_w * f_4_d%zero_mul_h * f_4_d%zero_lon_h

zero_mull = (NOR OF FD1R(B32) THROUGH FD1R(B57)) *
            (NOR OF MILSBR(0) THROUGH MILSBR(5))
           = zero_w * f_4_d%zero_mul_h * f_4_d%zero_mil_h

PSL.Z      = zero_l * (long * CVTfi) + zero_w * word +
             zero_b * byte + zero_l * zero_mull * MULL
```

During PHI_3, if stage 4's data_valid bit is set, the inverted value of the PSL.Z bit is driven onto the active-low shared bus.

11.25.5 Reserved Operand

The reserved operand fault is checked in stage 4 of the pipe. A reserved operand fault is possible only when the input operand is floating type. When a reserved operand fault occurs the other condition codes are overridden. The reserved operand detection is done in the miscellaneous pla.

For one operand instruction:

```
RES.OPD =  (f_3_c%e1zr_h * f_3%s1r_h)*(MOV + MNEG + CVTff + CVTfi + TST)
```

For two operand instruction:

```
RES.OPD =  (f_3_c%e1zr_h * f_3%s1r_h + f_3_c%e2zr_h * f_3%s2r_h) *
  (ADD+ SUB + DIVf + MULf + CMP)
```

11.25.6 Floating Divide by Zero

When a floating divide by zero occurs, the f_div_by_zero bit has to be set. The floating divide by zero fault occurs if operand 1 is zero. The logic is done in miscellaneous PLA.

f_div_by_zero = [f_3_c%e1zr_h * $\overline{\texttt{f\_3\%s1r\_h}}$] * (DIVf)

11.26 FBOX TESTABILITY

This section describes FBOX_Test mode of operation. FBOX_Test mode would primarily be used during chip debug and possibly during manufacturing tests.

11.26.1 FBOX_Test Control Signals

Two FBOX input signals are associated with FBOX_Test mode. E%FBOX_TEST_ENB_H is received from the EBOX, latched during PHI1, and driven down the FBOX pipe as F_I%FBOX_BYPASS_H. Assertion of E%FBOX_TEST_ENB_H puts the FBOX into FBOX_Test mode. A second signal, E%FBOX_S4_BYPASS_ENB_H, has the function of selecting two slightly different modes of FBOX_Test mode. E%FBOX_S4_BYPASS_ENB_H is received from the EBOX by a PHI1 latch and driven into the Fbox core as F_I%S4_BYPASS_ENB_H by a following PHI3 latch .

11.26.2 FBOX_Test Mode Description

FBOX_Test mode allows simple testing of the FBOX fraction and exponent datapaths. When in FBOX_Test mode, the basic operation of each stage is to pass fraction and exponent data, unchanged, from its input to its output. Thus, the test mode features allow FD1R or FD2R to be passed through the fraction datapath and ED1R to be passed through the exponent datapath. Selection of whether to pass FD1R or FD2R to the Fbox output is done, in Stage3, by looking at the value of F_I%S4_BYPASS_ENB_H. SIGN bit processing is not affected by FBOX_Test mode.

11.26.2.1 FBOX Section Operation During FBOX_Test Mode

Input and Output — The Input and Output sections of the FBOX operate as normal.

Divider — In the Divider, F_I%FBOX_BYPASS_H assertion forces F_D_C%DIVDONE_DAT_H to be asserted to Stage1 effectively bypassing the Divider. This enables Stage1 to use data supplied by the Input interface as the result of the Divider stage.

Stage1 — In Stage1, F_I%FBOX_BYPASS_H assertion forces Stage1 output register select signals to a state that writes the Stage1 FD1R, FD2R, and ED1R output registers with the contents of the Input interface FD1R, FD2R, and ED1R respectively.

Stage2 — In Stage2, F_I%FBOX_BYPASS_H assertion forces right-shifter control to a "shift_of_zero" in order to pass FD1R through Stage2. Output register select signals are forced to a state which writes the Stage2 FD1R and ED1R output registers with the contents of the Stage1 FD1R and ED1R. Stage2 FD2R is always written with the contents of Stage1 FD2R irrespective of FBOX_Test mode.

Stage3 — In Stage3, F_I%FBOX_BYPASS_H assertion forces left-shifter control to a "shift_of_zero" in order to pass FD1R or FD2R through Stage3. Selection of whether to pass FD1R or FD2R is done by the value on F_I%S4_BYPASS_ENB_H and output is on Stage3's FD1R. Stage3 ED1R output is written with Stage2 ED1R input while in FBOX_test mode. Stage3 fraction output selectors are forced to output the contents of the left_shifter during FBOX_Test mode. The following table describes Stage3 operation modes and data driven on various busses for different modes of operation.

The main features of this implementation are:

- o Either FD1R or FD2R can be selected to pass directly through the FBOX
- o The two shared busses between Stages 3/4 and the output interface can be selectively driven by Stage 3 or Stage 4.
- o Provides visibility of the Stage3 miniround incrementer results.

```
F_I%FBOX_BYPASS_H
|
|F_I%S4_BYPASS_ENB_H
||
VV
```

	Stage 3 Operation Mode	Miniround Incrementer Input	Value Appearing On Busses F_B%F_OUT_L<B1:B55>	F_3%FD1R_H<A0:B58>	F_B%E_OUT_L<10:0>	F_3%ED
00	Normal Operation w/ S4_Bypass = OFF	Opcode Dependent	Stage 4 fraction result	Stage 3 fraction result	Stage 4 exponent result	Stage re
01	Normal Operation w/ S4_Bypass = ON	Opcode Dependent	Stage 3 fraction result if Stage 4 bypassed, else Stage 4 fraction	Stage 3 fraction result	Stage 3 exponent result if Stage 4 bypassed, else Stage 4 exponent	Stage re
10	FBOX_BYPASS, S4_Bypass = OFF	Opcode Dependent	Stage 4 Driven FD1R_H<B1:B55>	FD1R_H<A0:B58> *	Stage 4 Driven ED1R_H<10:0>	ED1
11	FBOX_BYPASS, S4_Bypass = ON, bypassable opcode	Opcode Dependent ** see footnote	Stage 3 Driven FD2R_H<B1:b55> **	FD2R_H<A0:B58> *	Stage 3 Driven ED1R_H<10:0>	ED1
11	FBOX_BYPASS, S4_Bypass = ON, non-bypassable opcode	Opcode Dependent	Stage 4 Driven FD2R_H<B1:B55> *	FD2R_H<A0:B58> *	Stage 4 Driven ED1R_H<10:0>	ED1

* = All fraction data bits are passed through Stage 3, as received, by way of the left shifter.

** = In FBOX_Test mode,with S4_Bypass on and a bypassable opcode in Stage3 the majority but not all of frac bits are passed through Stage 3,as received, by way of the left shifter and the output selector choosi shifter output.
For F-type data two fraction bits (B22:B23) are passed through Stage3 by way of the miniround incremen Similarly, for D-type data six fraction bits (B50:B55) and for G-type data three fraction bits (B50:B5 are passed through Stage3 miniround incrementers.
It is important to note that the control logic for the miniround input selectors makes it's selection on opcode information and the signal F_3_A%SHFT_DONE_H. FBOX_Test mode is not factored into the minir incrementer's input selector control. Depending on the opcode and exponent difference, miniround inpu could choose left shifter output or fraction adder output to be fed to the miniround incrementers.

The simplest way to pass FD2R through Stage3 (unchanged) is to select the proper opcode and data such that an effective subtract with exponent difference of zero will enter Stage3. This will select Stage 3's left shifter output as the source for the miniround incrementer input and the round bit position will be zero.

Stage4 — In Stage4, F_I%FBOX_BYPASS_H assertion forces fraction adder carry-in and round signals to zero to allow FD1R to pass through Stage3 unchanged. Stage4 FD1R and ED1R are written with the contents of Stage3 FD1R and ED1R respectively.

11.26.3 Revision History

Table 11–22: Revision History

Who	When	Description of change
Anil Jain	17-Mar-1989	Initial Release
Anil Jain	18-Dec-1989	Updated to reflect the Fbox implementation
Dave Deverell	25-Jan-1991	Updated to reflect PASS1 implementation and FOX_Test section added

Chapter 12

The Mbox

12.1 INTRODUCTION

The Mbox performs three primary functions:

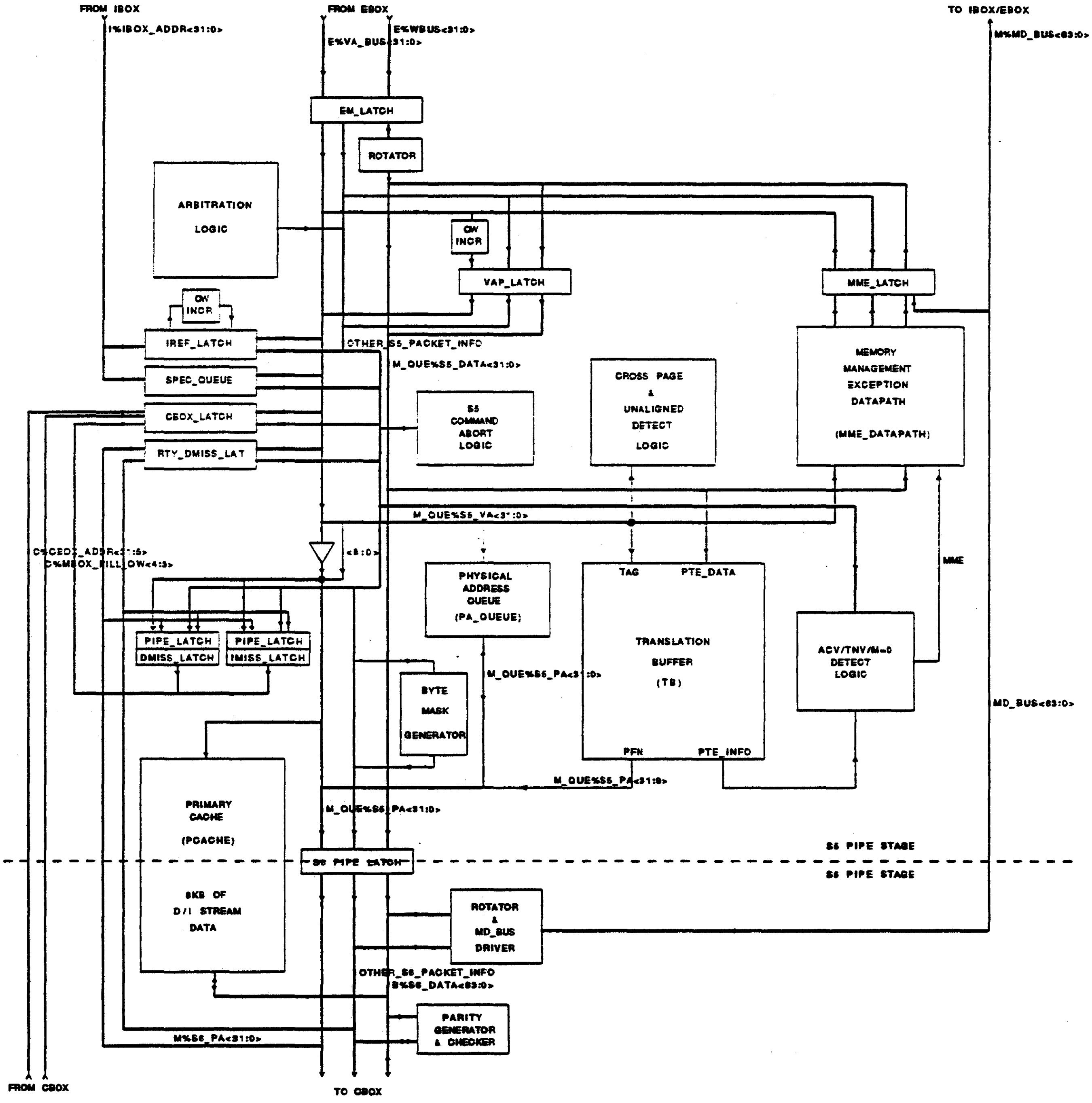
- VAX memory management: The Mbox, in conjunction with the operating system memory management software, is responsible for the allocation and use of physical memory. The Mbox performs the hardware functions necessary to implement VAX memory management. It performs translations of virtual addresses to physical addresses, access violation checks on all memory references, and initiates the invocation of software memory management code when necessary.
- Reference processing: Due to the macropipeline structure of NVAX, and the coupling between NVAX and its memory subsystem, the Mbox can receive memory references from the Ibox, Ebox and Cbox simultaneously. Thus, the Mbox is responsible for prioritizing, sequencing, and processing all references in an efficient and logically correct fashion and for transferring references and their corresponding data to/from the Ibox, Ebox, Pcache, and Cbox.
- Primary Cache Control: The Mbox maintains an 8KB physical address cache of I-stream and D-stream data. This cache, called the Pcache (Primary Cache), exists in order to provide a two cycle pipeline latency for most I-stream and D-stream data requests. It is the fastest D-stream storage medium for NVAX and represents the first level of D-stream memory hierarchy and the second level of I-stream memory hierarchy for the NVAX computer system. The Mbox is responsible for controlling Pcache operation.

12.2 MBOX STRUCTURE

This section presents a block diagram of the Mbox and defines the function of the basic Mbox components. This section neither explains why the functions of each component exist nor does it discuss the interactions among the components. The intent of this section is only to define the function and interconnection of the components for future discussion. Subsequent sections will deal component interaction.

The following block diagram illustrates the basic components of the Mbox.

Figure 12–1: Mbox Block Diagram



The Mbox is implemented as a two-stage pipeline located in the fifth and sixth segments of the NVAX macropipeline (S5 and S6). References processed by the Mbox are first executed in S5. Upon successful completion in S5, the reference is transferred into S6. At this point, the reference has either completed or is transferred to the Ibox, Ebox, or Cbox.

During any cycle, the fundamental state of the S5 and S6 stages can be defined by the particular references which currently reside in these two stages. For the purposes of describing the Mbox, all references can be viewed as a packet of information which is transferred on the S5 and S6 buses. The S5 reference packet, and the corresponding S5 buses are defined as:

- ADDRESS: The M_QUE%S5_VA_H<31:0> bus transfers all virtual addresses and some physical addresses into the S5 pipe. The M_QUE%S5_PA_H<31:0> bus transfers some physical addresses into the S5 pipe and transfers all addresses out of the S5 pipe.
- DATA: M_QUE%S5_DATA_H<31:0> transfers data originating from the Ebox, through the S5 pipe.
- COMMAND: M_QUE%S5_CMD_H<4:0> transfers the type of reference through the S5 pipe. This command field is defined in Section 12.3.1.
- TAG: The M_QUE%S5_TAG_H<4:0> transfers the Ebox register file destination address corresponding to the reference through the S5 pipe.
- DEST_BOX: M_QUE%S5_DEST_H<1:0> transfers the reference destination information through the S5 pipe. This field is defined as follows:

M_QUE%S5_DEST_ H	Definition
00:	the reference requests data destined for the Mbox.
01:	the reference requests data destined for the Ibox.
10:	the reference requests data destined for the Ebox.
11:	the reference requests data destined for the Ebox and Ibox.

- AT: The M_QUE%S5_AT_H<1:0> transfers the access type of the reference. This field is defined as follows:

M_QUE%S5_AT_ H	Definition
00:	tb passive query access (See PROBE command)
01:	read access
10:	write access
11:	modify access (read with write check for future write to same addr)

- DL: The M_QUE%S5_DL_H<1:0> transfers the data length of the reference. This field is defined as follows:

M_QUE%S5_DL_ H	Definition
00:	byte
01:	word
10:	longword
11:	quadword

- REF_QUAL: The **M_QUE%S5_QUAL_H**<6:0> transfers information which further qualifies the reference for the purpose of Mbox processing. This field is defined as follows:

M_QUE%S5_QUAL_H bit	Definition
M_QUE%S5_QUAL_H<6>	address of reference is currently a virtual address.
M_QUE%S5_QUAL_H<5>	reference has been tested for cross-page condition.
M_QUE%S5_QUAL_H<4>	reference is first part of an unaligned reference.
M_QUE%S5_QUAL_H<3>	reference is second part of an unaligned reference.
M_QUE%S5_QUAL_H<2>	enable ACV and M=0 checks.
M_QUE%S5_QUAL_H<1>	reference has or is forced to have a hard error.
M_QUE%S5_QUAL_H<0>	reference has or is forced to have a memory management fault (ACV/TNV/M=0).

The S6 reference packet, and the corresponding S6 buses are defined as:

- ADDRESS: The **M%S6_PA_H**<31:0> bus transfers a physical address through the S6 pipe.
- DATA: **B%S6_DATA_H**<63:0> transfers data through the S6 pipe.
- COMMAND: **M%S6_CMD_H**<4:0> transfers the type of reference through the S6 pipe. This command field is defined in Section 12.3.1.
- DEST_BOX: **M_QUE_MS2%S6_DEST_H**<1:0> transfers the reference destination information through the S6 pipe. This field is defined as follows:

M_QUE_MS2%S6_ DEST_H	Definition
00:	the reference requests data destined for the Mbox.
01:	the reference requests data destined for the Ibox.
10:	the reference requests data destined for the Ebox.
11:	the reference requests data destined for the Ebox and Ibox.

- S6_BYTE_MASK: **M%S6_BYTE_MASK_H**<7:0> transfers the byte mask information through the S6 pipe. The byte mask field is used to indicate which bytes of a longword or quadword write should actually be written to a cache or memory.
- REF_QUAL: **M_QUE_MS2%S6_QUAL_H**<3:0> transfers information which further qualifies the reference for the purpose of Mbox processing. This field is defined as follows:

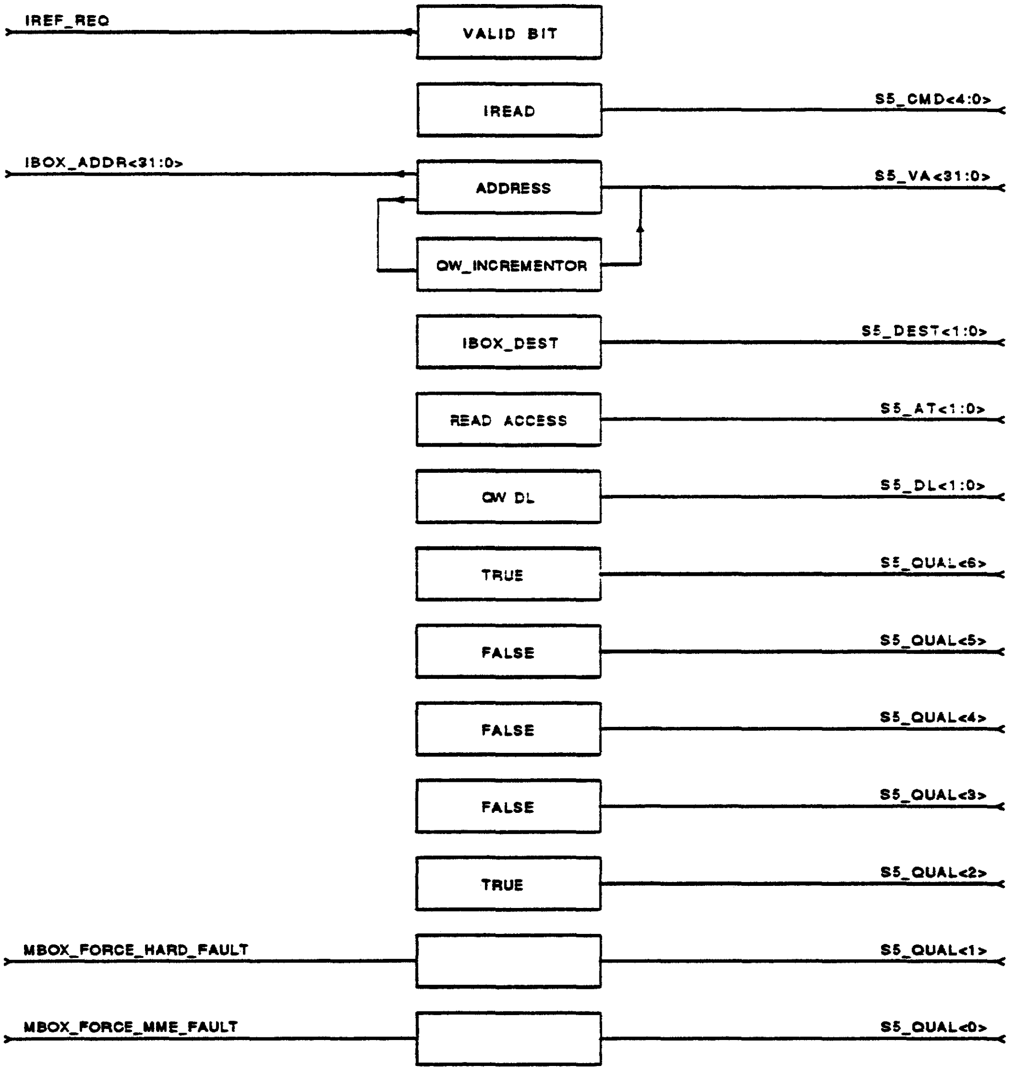
M_QUE_MS2%S6_QUAL_H bit	Definition
M_QUE_MS2%S6_QUAL_H<3>	reference is first part of an unaligned reference.
M_QUE_MS2%S6_QUAL_H<2>	reference is second part of an unaligned reference.
M_QUE_MS2%S6_QUAL_H<1>	reference has or is forced to have a hard error.
M_QUE_MS2%S6_QUAL_H<0>	reference has or is forced to have a memory management fault (ACV/TNV/M=(

12.2.1 IREF_LATCH

The IREF_LATCH is a latch which stores all I-stream read references (IREADs) requested by the Ibox. Each IREAD is stored in the IREF_LATCH until the reference successfully completes in S5.

The following figure illustrates the structure of the IREF_LATCH:

Figure 12-2: Iref Latch



The output of the address field of the IREF_LATCH has an incrementer associated with it in order to increment the quadword address. The output of this structure can be tristated.

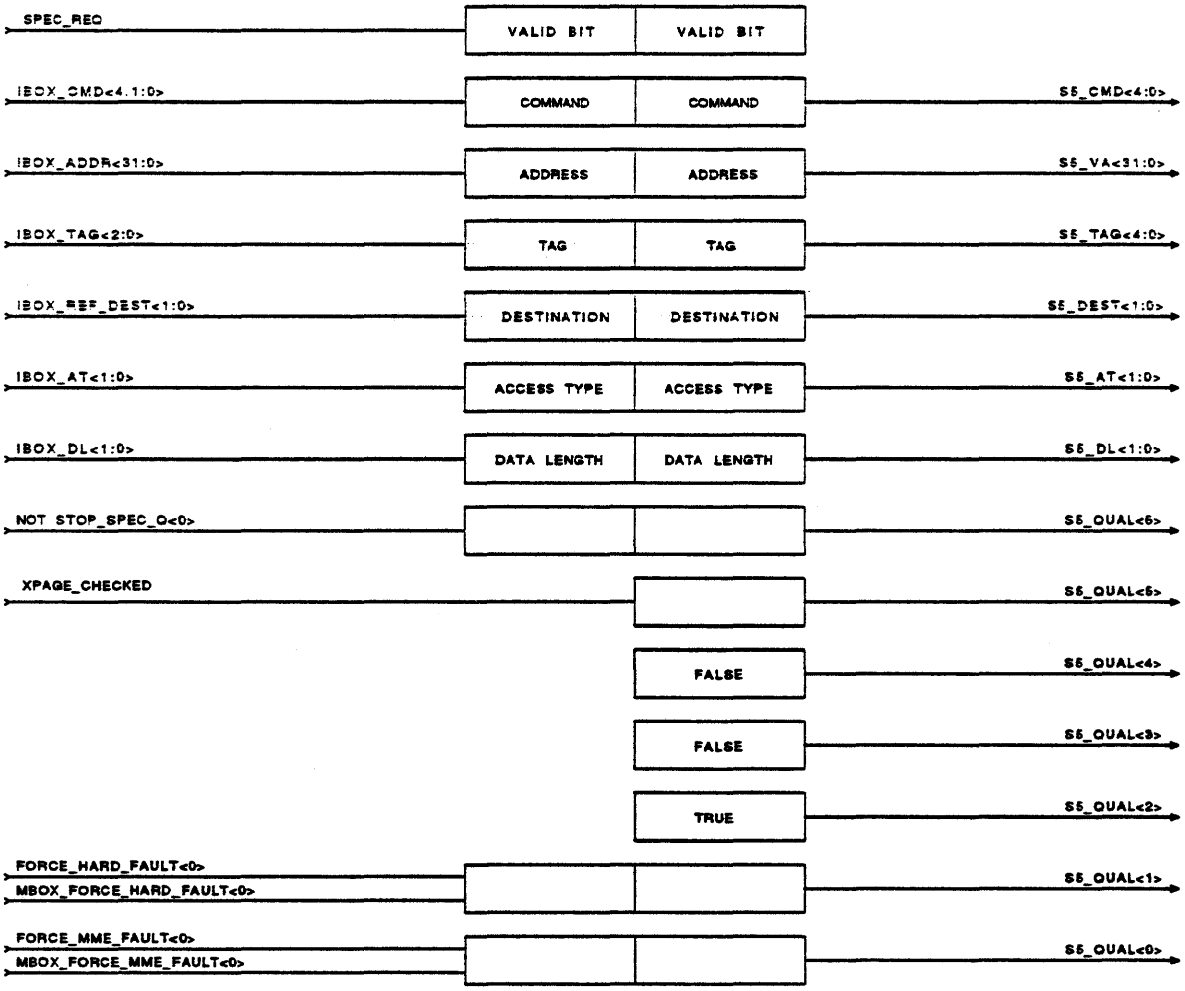
See Section 12.3.5.2 for a more complete understanding of IREF_LATCH function in the context of overall Mbox operation.

12.2.2 SPEC_QUEUE

The SPEC_QUEUE is a 2-entry FIFO structure which stores D-stream read and write references associated with specifier source and destination operands decoded by the Ibox. Each reference latched in the SPEC_QUEUE is stored until the reference successfully completes in S5. If the reference is unaligned, the entire reference must complete in S5 before the corresponding entry is invalidated.

The following figure illustrates the structure of the SPEC_QUEUE:

Figure 12–3: Spec Queue



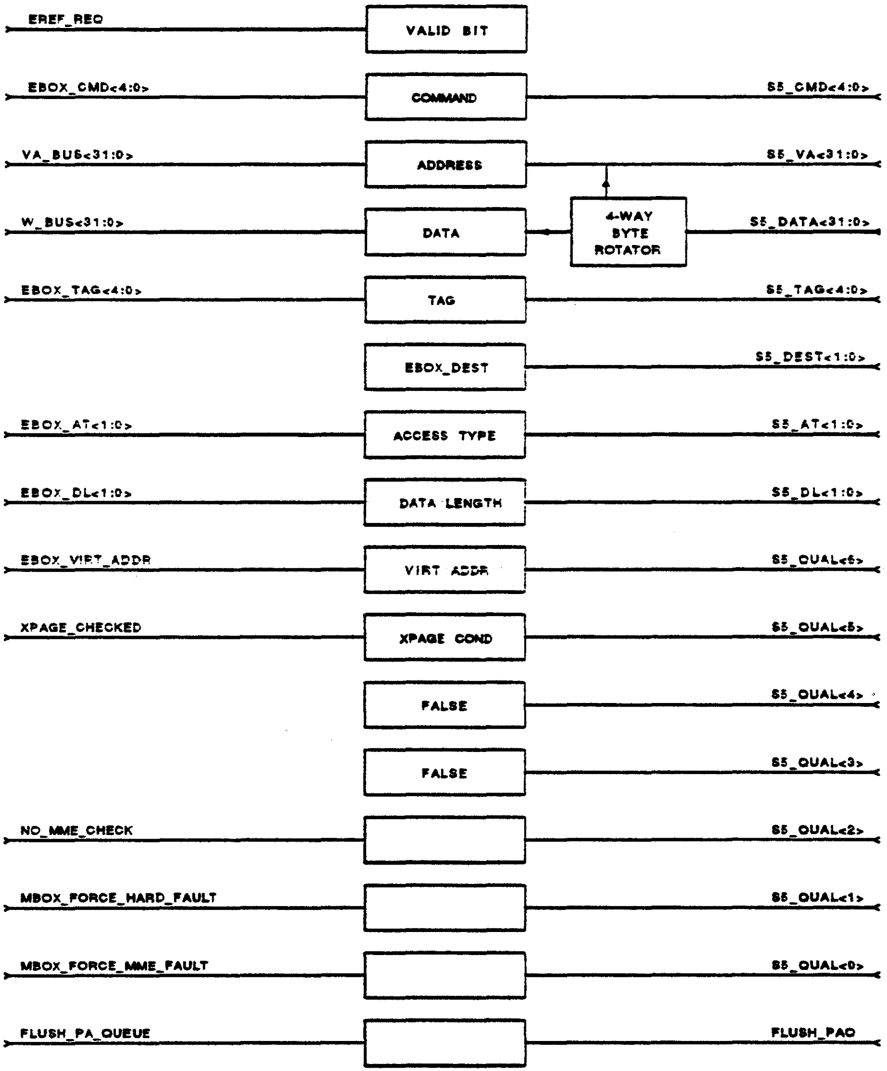
The output of this structure can be tristated.

12.2.3 EM_LATCH

The EM_LATCH latches and stores all commands originating from the Ebox. Each reference is stored until the following two conditions are satisfied: 1) the "complete logical reference" (i.e. the pair of aligned references required if the EM_LATCH reference is unaligned) clear memory management access checks, and 2) the EM_LATCH reference successfully completes in S5.

The following figure illustrates the structure of the EM_LATCH:

Figure 12–4: EM_LATCH



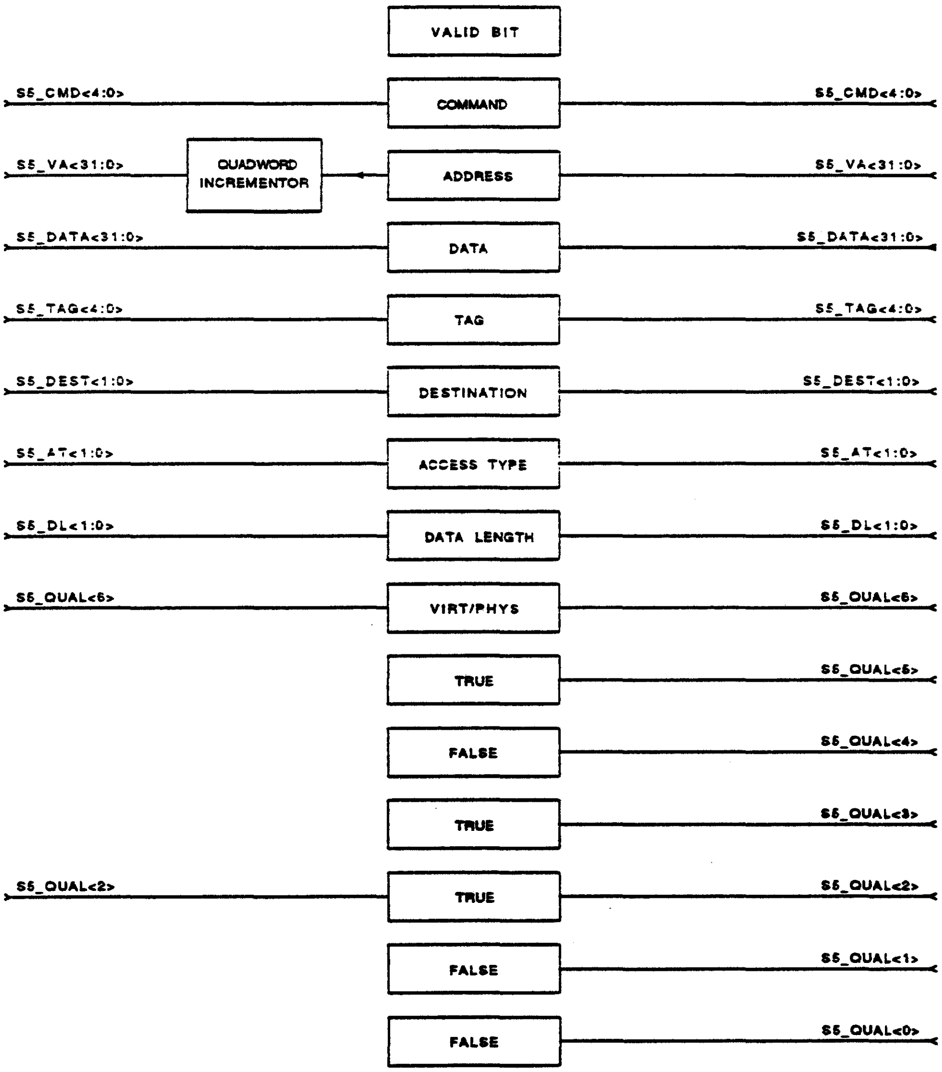
A 4-way byte barrel shifter is connected to the data portion of the EM_LATCH. This enables the write data to be byte-rotated into longword alignment.

The EM_LATCH output can be tristated.

12.2.4 VAP_LATCH

The function of the VAP_LATCH is to create and store the second reference of an unaligned reference pair. Each reference is stored until the reference successfully completes in S5. The following figure illustrates the structure of the VAP_LATCH:

Figure 12–5: VAP_LATCH



The VAP_LATCH transforms the current S5 reference into a new reference. Thus, input for the VAP_LATCH is taken off of the S5 buses. An incrementor exists on the input side of the address field which adds eight to **M_QUE%S5_VA_H<31:0>** in order to create the second reference in an unaligned pair of references.

The VAP_LATCH output can be tristated.

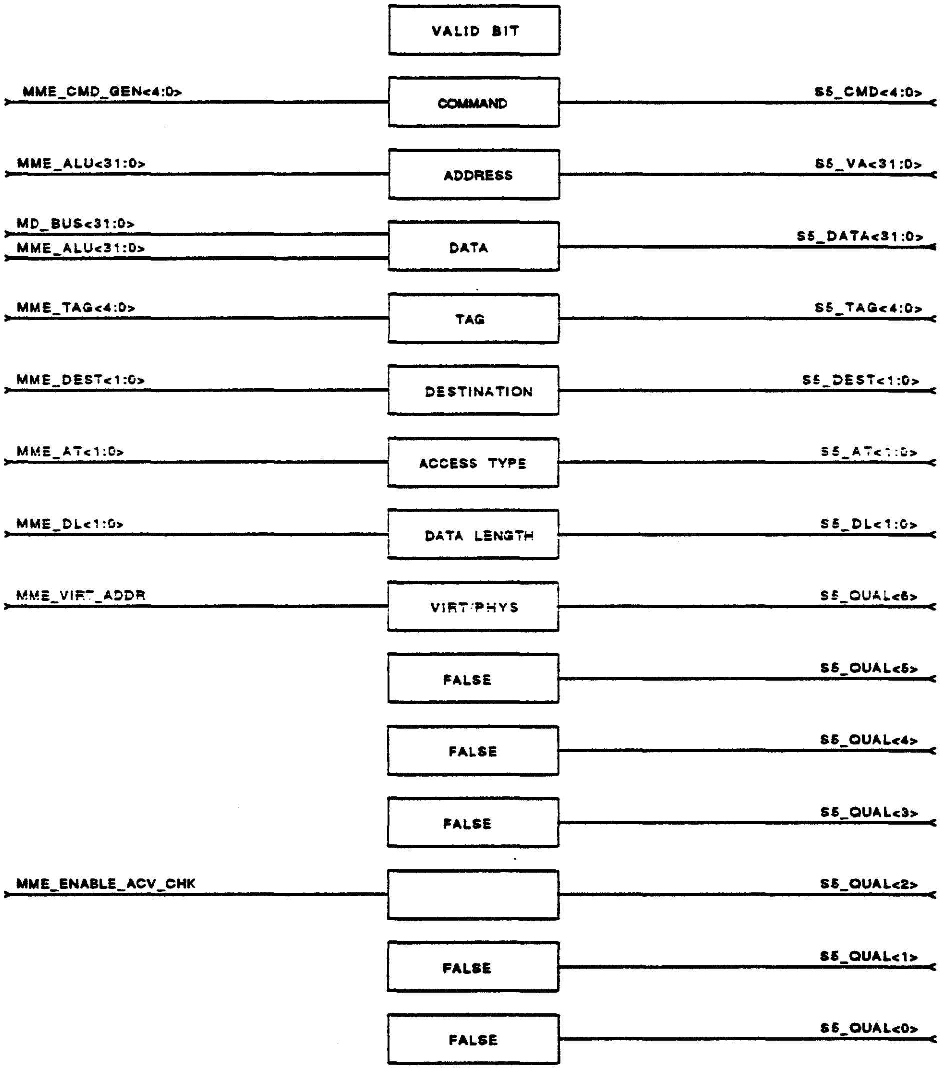
See Section 12.3.17 for a more complete understanding of VAP_LATCH function in the context of overall Mbox operation.

12.2.5 MME_LATCH

The MME_LATCH (Memory Management Exception Latch) stores references associated with memory management processing. It acts as a buffer between the S5 processing pipe and the MME_DATAPATH. The MME_LATCH is the S5 source for PTE references (page table entry reads), PTE data, and Mbox internal processor registers and TB fill operations.

The following figure illustrates the structure of the MME_LATCH:

Figure 12–6: MME_LATCH



Each reference is stored until the reference successfully completes in S5.

The MME_LATCH output can be tristated.

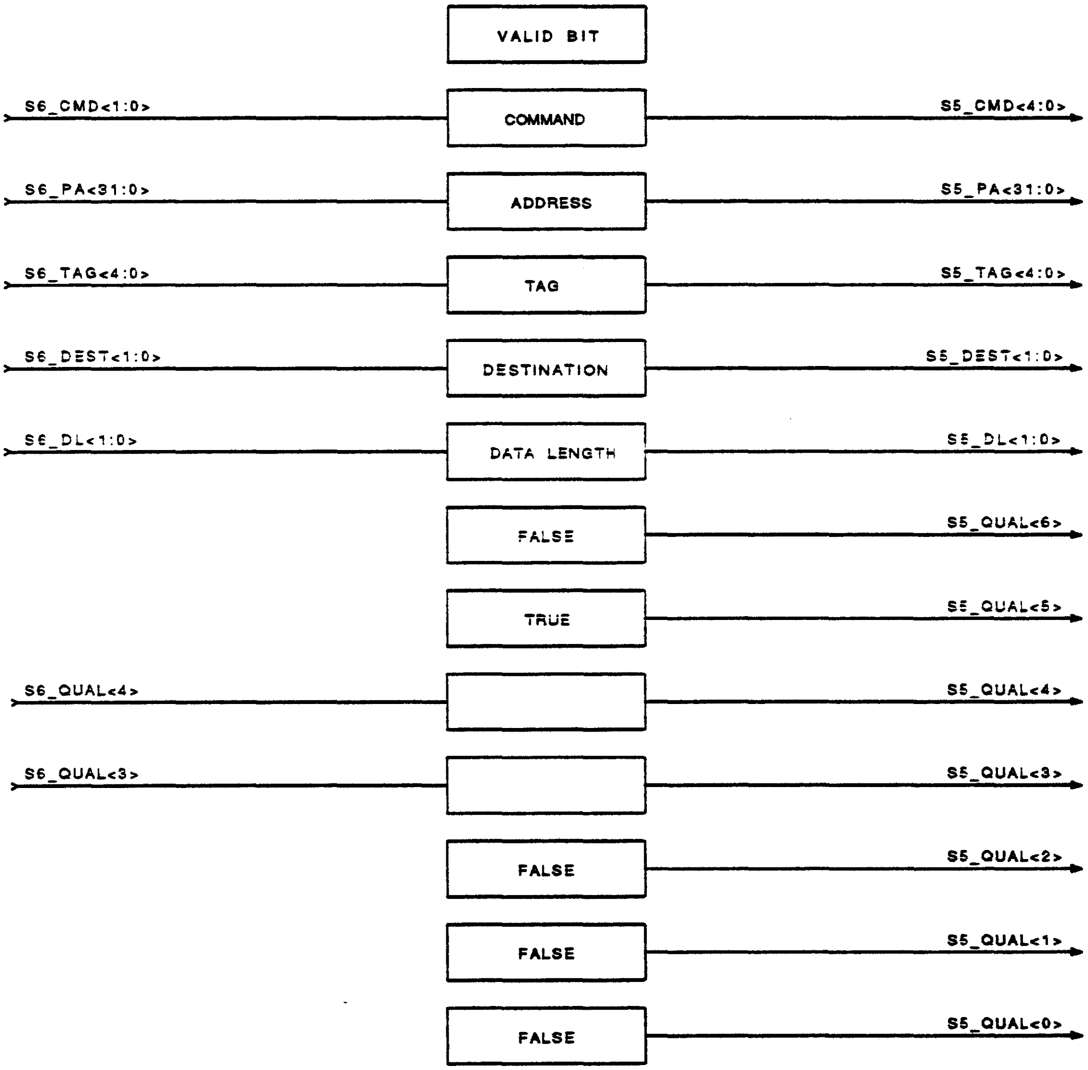
12.2.6 RTY_DMISS_LATCH

The RTY_DMISS_LATCH stores D-stream reads which missed in the Pcache when a previous D-stream fill sequence has not yet completed. This latch is the mechanism by which a D-stream read, which missed in the S6 pipe during another D-stream fill sequence, can be retried in the S5 pipe at some later point.

An S6 D-stream read is loaded into the RTY_DMISS_LATCH when it misses in the Pcache while a previous D-stream fill sequence is in progress. A RTY_DMISS_LATCH is driven into the S5 pipe during or after the point when the final D_CF reference is executing in S6 to complete the previous fill sequence. A RTY_DMISS_LATCH reference is invalidated when its read is retired from S5.

The following figure illustrates the structure of the RTY_DMISS_LATCH:

Figure 12-7: RTY_DMISS_LATCH



The RTY_DMISS_LATCH output can be tristated.

See Section 12.3.5.3.1 for a more complete understanding of RTY_DMISS_LATCH function in the context of overall Mbox operation.

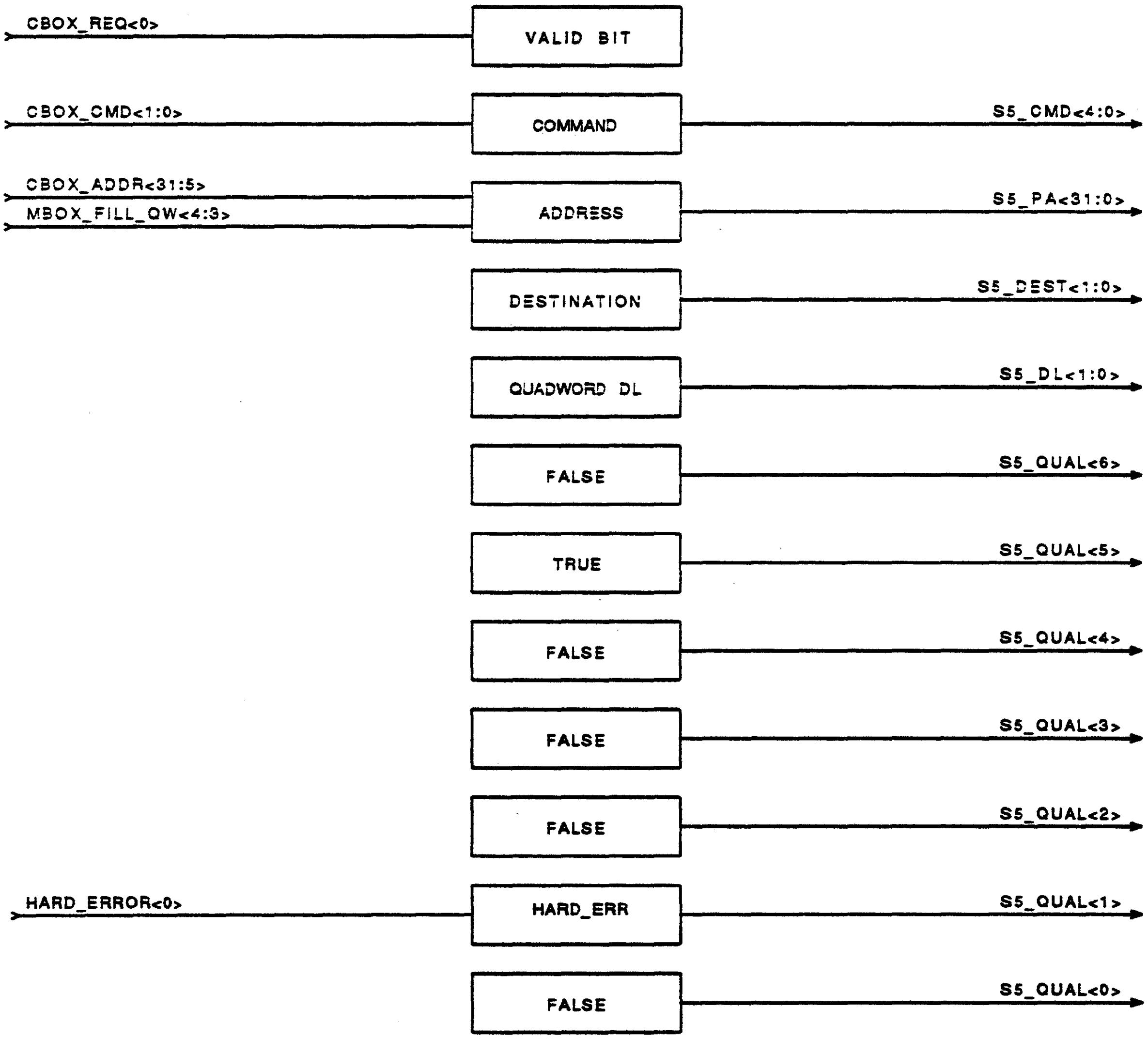
12.2.7 CBOX_LATCH

The CBOX_LATCH stores references originating from the Cbox. These references are I-stream Pcache fills, D-stream Pcache fills, or Pcache hexaword invalidates.

Each reference is stored until the reference successfully completes in S5.

The following figure illustrates the structure of the CBOX_LATCH:

Figure 12–8: CBOX_LATCH



Note that no data field is present in this latch even though this latch services cache fill commands.

Cache fill data will be supplied to the Pcache on the **B%S6_DATA_H** Bus by the Cbox during the appropriate S6 cache fill cycle. The **C%CBOX_ADDR_H** bus is driven by the Cbox during invalidate commands. During cache fill commands, all but two bits of the **C%CBOX_ADDR_H** bus are driven by the DMISS_LATCH or IMISS_LATCH. The Cbox will drive **C%MBOX_FILL_QW_H<4:3>** during cache fill commands in order to supply the quadword alignment of the fill data within the hexaword block.

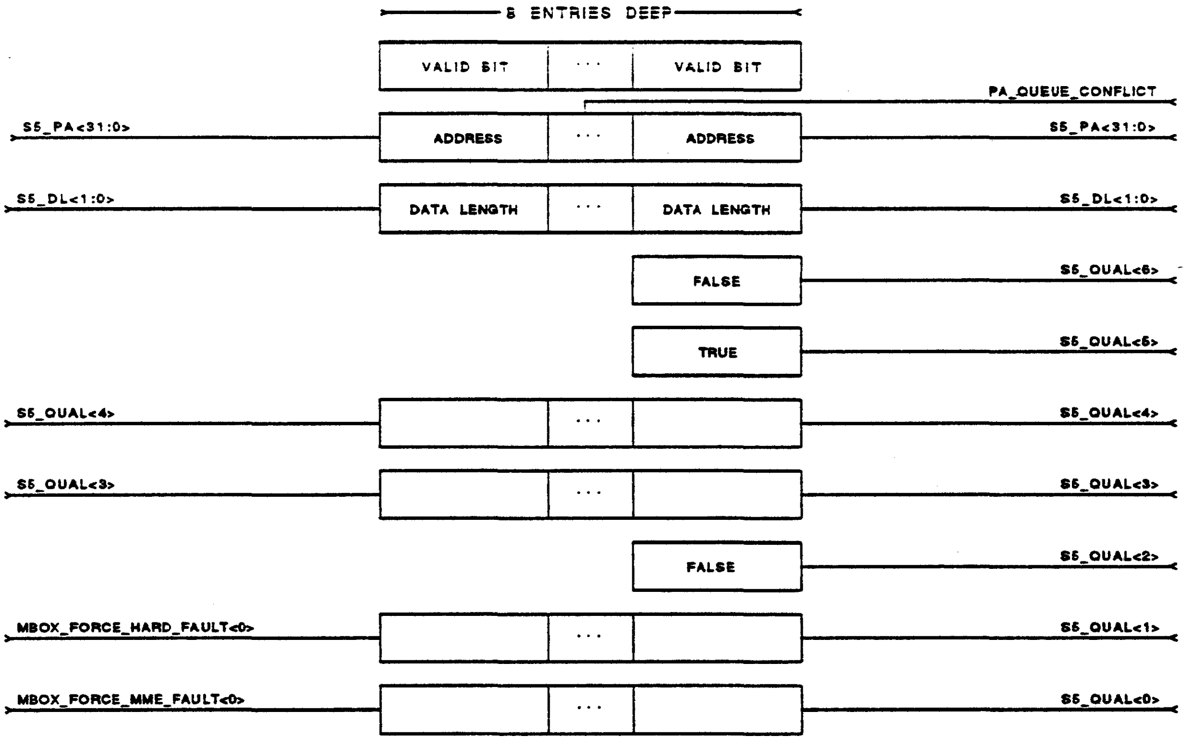
The CBOX_LATCH output can be tristated.

12.2.8 PA_QUEUE

The PA_QUEUE (Physical Address Queue) stores the physical addresses associated with destination specifier references made by the Ibox via a DEST_ADDR or READ_MODIFY command. The Ebox will supply the corresponding data at some later time via a STORE command. When the STORE data is supplied, the PA_QUEUE address is matched with the STORE data and the reference is turned into a physical WRITE operation.

The following figure illustrates the structure of the PA_QUEUE:

Figure 12–9: PA_QUEUE



The PA_QUEUE is organized as a 8-entry FIFO. Addresses from the Ibox are expected in the same order as the corresponding data from the Ebox.

The PA_QUEUE has address comparators built into all FIFO entries. These comparators detect when the physical address bits <8:3> of a valid PA_QUEUE entry matches the corresponding physical address of an Ibox D-stream read.

See Section 12.3.6.1 and Section 12.3.18.1.1 for a more complete understanding of PA_QUEUE function in the context of overall Mbox operation.

12.2.9 TB

The TB (translation buffer) is the mechanism by which the Mbox performs quick virtual-to-physical address translations. It is a 96-entry fully associative cache of PTEs (Page Table Entries). Bits 31 through 9 of all S5 virtual addresses act as the TB tag. The replacement algorithm implemented is Not-Last-Used.

See Section 12.5.1.3 for more information.

12.2.10 MME_DATAPATH

The MME_DATAPATH (Memory Management Datapath) is used to process most memory management functions performed by the Mbox. Specifically, it performs the following functions:

- Creates read references of PTEs in order to obtain virtual address translations not currently cached in the TB.
- Creates TB fill references in order to write PTE data into the TB.
- Stores memory management internal processor registers.
- Stores virtual addresses associated with memory management faults or TB parity errors.

The MME_DATAPATH implements these functions with a register file and an ALU. See Section 12.5.1 for a more complete description of the MME_DATAPATH.

12.2.11 ARBITRATION LOGIC

The ARBITRATION LOGIC is responsible for determining which reference source drives its reference packet into the S5 pipe. (See Section 12.3.4 for more information about reference arbitration.)

12.2.12 S6_PIPELATCH

The S6_PIPELATCH is the buffer between the S5 and S6 stages of the Mbox pipeline. It latches the S5 reference packet, modifies it appropriately, and drives it as an S6 reference packet into the S6 pipe. **M_QUE%S5_DATA_H**<31:0> is driven onto both the upper and lower halves of **B%S6_DATA_H**<63:0>. **M%S6_CMD_H**<4:0> is either:

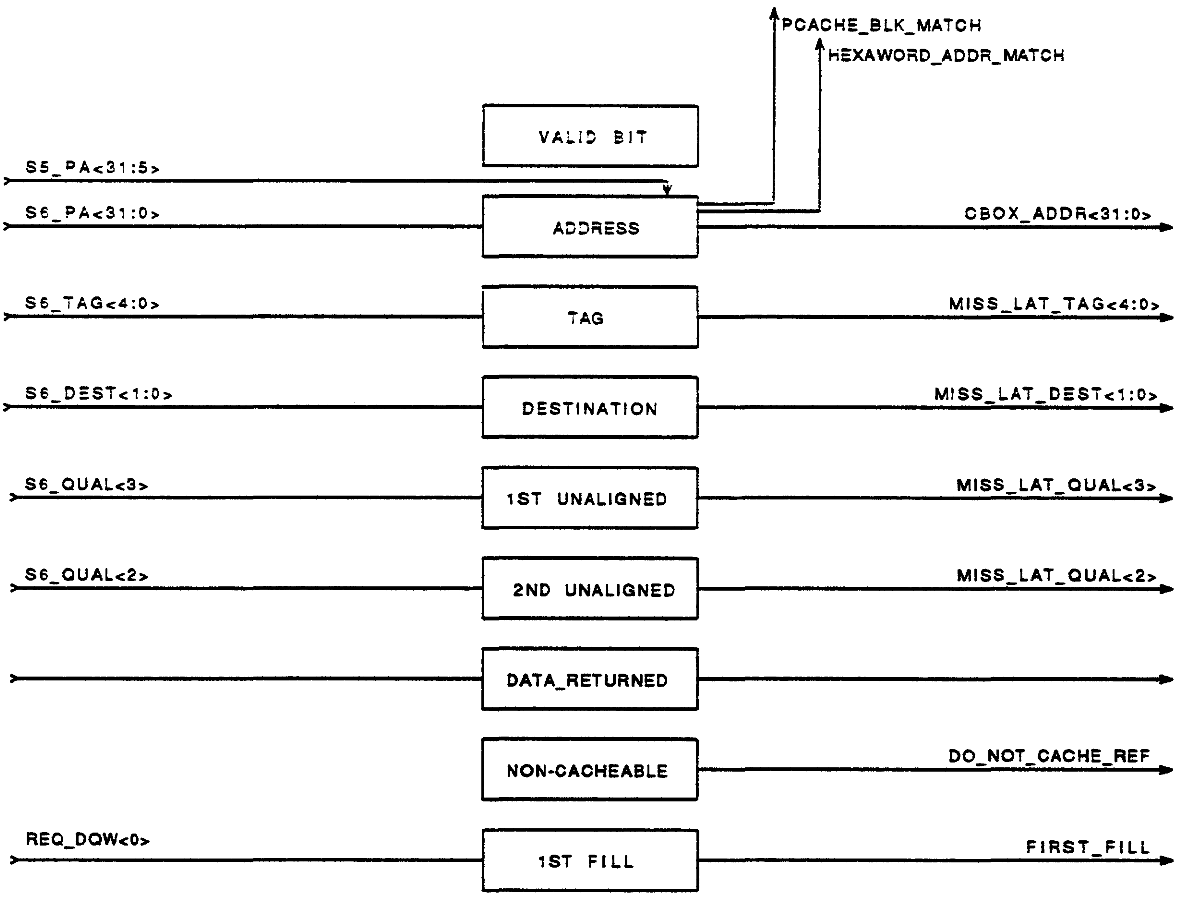
1. driven by the **M_QUE%S5_CMD_H**<4:0>
2. is changed into a NOP

12.2.13 DMISS_LATCH and IMISS_LATCH

The DMISS_LATCH stores the currently outstanding D-stream read. That is, a D-stream read, which missed in the Pcache, is stored in the DMISS_LATCH until the corrsponding Pcache block fill operation completes. The DMISS_LATCH also stores IPR_RDs to be processed by the Cbox until the Cbox supplies the data. I-stream reads are handled analogously by the IMISS_LATCH except that IPR_RDs are never handled by the IMISS_LATCH.

The following figure illustrates the structure of the DMISS_LATCH and the IMISS_LATCH:

Figure 12–10: DMISS_LATCH and IMISS_LATCH



These two latches have comparators built in in order to detect the following conditions:

- If the hexaword address of an invalidate matches the hexaword address stored in either MISS_LATCH, the corresponding MISS_LATCH sets a bit to indicate that the corresponding fill operation is no longer cacheable in the Pcache.

- Address<11:5> addresses a particular Pcache index (corresponding to two Pcache blocks). If address<8:5> of the DMISS_LATCH matches the corresponding bits of the physical address of an S5 I-stream read, the S5 I-stream read is stalled until the entire D-stream fill operation completes. This prevents the possibility of causing a D-stream fill sequence to a given Pcache block from simultaneously happening with an I-stream fill sequence to the same Pcache block.
- By the same argument, address<8:5> of the IMISS_LATCH is compared against S5 D-stream reads to prevent another simultaneous I-stream/D-stream fill sequence to the same Pcache block.
- Address<8:5> of both miss_latches is compared against any S5 memory write operation. This is necessary to prevent the write from interfering with the cache fill sequence.

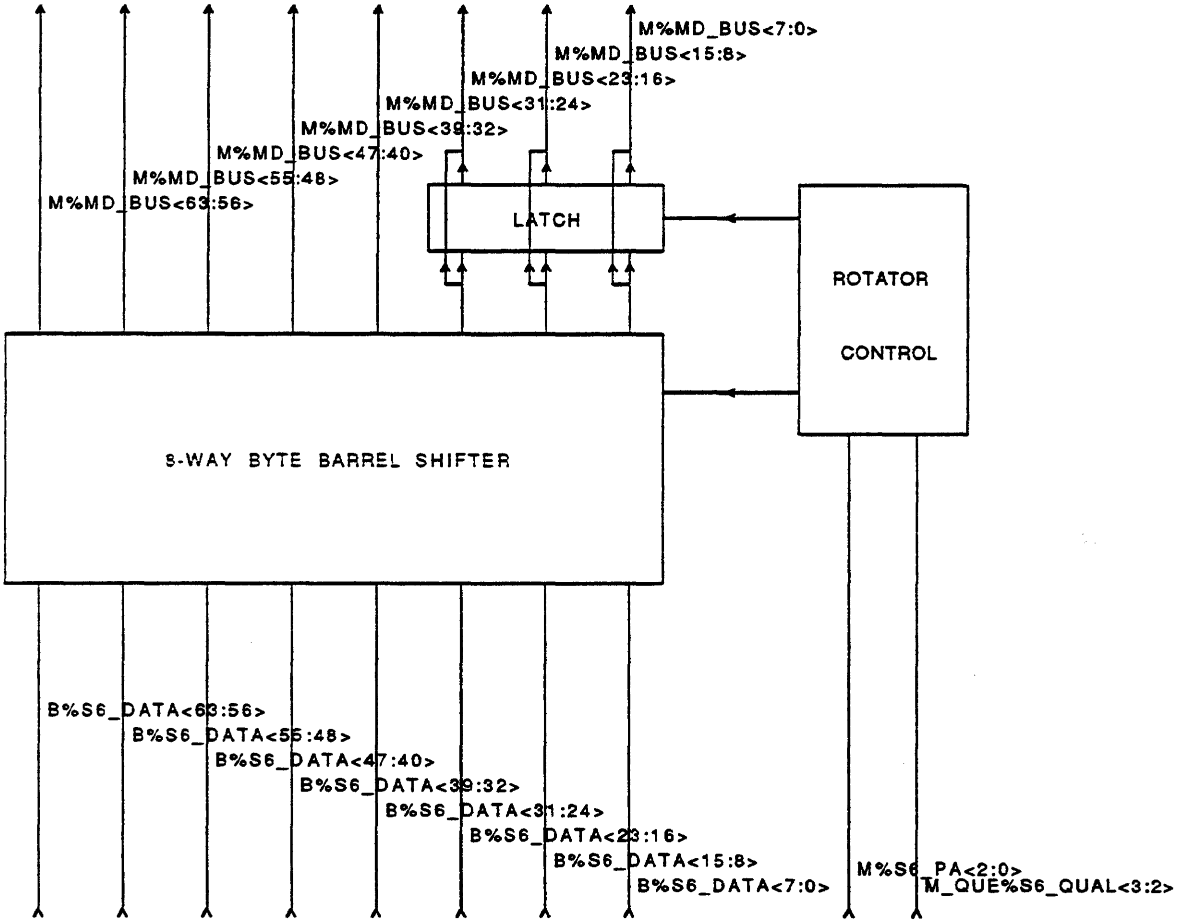
See Section 12.3.5.1 for a more complete understanding of the DMISS_LATCH/IMISS_LATCH functions in the context of overall Mbox operation.

12.2.14 MD_BUS_ROTATOR

The function of the MD_BUS_ROTATOR is to right-justify read data and drive it on the M%MD_BUS_H. For unaligned reads (see Section 12.3.17.1) the MD_BUS_ROTATOR is designed to assemble read data from two read references and drive it on the M%MD_BUS_H in right-justified form. This rotator coupled with the Mbox decomposition of unaligned references into two aligned references, allows the Ibox and Ebox to issue unaligned D-stream reads and receive the requested data aligned to the Ebox datapath.

The MD_BUS_ROTATOR is illustrated below:

Figure 12-11: MD_BUS_ROTATOR



Although the diagram above describes the MD_BUS_ROTATOR as an 8-way byte barrel shifter, its actual design is a functional subset of a full barrel shifter. The lower four bytes of the output of the rotator are designed as a full 8-way byte barrel shifter in order to right-justify D-stream longword data. However, the upper four bytes always directly pass **M%MD_BUS_H<63:32>** since these bytes are only used when aligned I-stream quadword data is sent to the VIC.

12.2.15 Pcache

The Pcache is a two-way set associative, read allocate, no-write allocate, write through, physical address cache of I-stream and D-stream data. It stores 8192 bytes (8K) of data and 256 tags corresponding to 256 hexaword blocks (1 hexaword = 32 bytes). Each tag is 20 bits wide corresponding to bits <31:12> of the physical address. There are four quadword subblocks per block with a valid bit associated with each subblock. The access size for both Pcache reads and writes is one quadword. Byte parity is maintained for each byte of data (32 bits per block). One bit of parity is maintained for every tag. The Pcache has a one cycle access and a one cycle repetition rate for both reads and writes (note however, that the entire Mbox latency is two cycles due to the two stage Mbox pipeline).

The Pcache represents the first level of D-stream memory hierarchy and the second level of I-stream memory hierarchy in all NVAX computer systems. Pcache entries must be invalidated in order to maintain cache coherency with higher levels of the memory hierarchy. See Section 12.4 for more information on the Pcache.

12.3 REFERENCE PROCESSING

This section discusses how references are processed by the Mbox, and how the Mbox functional components interact to carry out reference processing.

12.3.1 REFERENCE DEFINITIONS

The following table describes all types of references processed by the Mbox:

Table 12–1: Reference Definitions

Name	Value (hex)	Reference Source	Description
IREAD	0E	Ibox	Aligned quadword I-stream read
DREAD	1C	Ibox, Ebox, Mbox	Variable length D-stream read
DREAD_MODIFY	1D	Ibox	Variable length D-stream read with modify intent as a result of Ibox-decoded modify specifiers
DREAD_LOCK	1F	Ebox	Variable length D-stream read with atomic memory lock
WRITE_UNLOCK	1A	Ebox	Variable length write with atomic memory unlock
WRITE	1B	Ebox	Variable length write
DEST_ADDR	0D	Ibox	Supplies address of a write-only destination specifier
STORE	19	Ebox	Supplies write data corresponding to a previously translated destination specifier address.
IPR_WR	06	Ebox	Internal Processor Register Write
IPR_RD	07	Ebox	Internal Processor Register Read
IPR_DATA	04	Mbox	Transfers Mbox IPR data to Ebox
LOAD_PC	05	Ebox	Transfers a PC value to Ibox via M%MD_BUS_H<31:0>
PROBE	09	Ebox	Mbox returns ACV/TNV/M=0 status of specified address to Ebox.
MME_CHK	08	Ebox, Mbox	Performs ACV/TNV/M=0 check on specified address and invokes the appropriate memory management exception
TB_TAG_FILL	0C	Ebox, Mbox	Writes a TB tag into a TB entry.
TB_PTE_FILL	14	Ebox, Mbox	Writes PTE data into a TB entry.

Table 12–1 (Cont.): Reference Definitions

Name	Value (hex)	Reference Source	Description
TBIS	10	Ebox	Invalidates a specific PTE entry in the TB.
TBIA	18	Ebox,Mbox	Invalidates all entries in TB.
TBIP	11	Ebox	Invalidates all PTE entries in TB corresponding to process-space trans lations.
D_CF	03	Cbox	D-stream quadword Pcache fill
I_CF	02	Cbox	I-stream quadword Pcache fill
INVAL	01	Cbox	Hexaword invalidate of a Pcache entry
STOP_SPEC_Q	0F	Ibox	Stops processing of specifier references.
NOP	00	Ibox, Ebox, Mbox	No operation

12.3.2 SIMPLE MBOX PIPELINE FLOW

A major Mbox design consideration was to return requested read data to the Ibox and Ebox as quickly as possible in order to minimize macropipeline stalls. If the Ebox pipeline is stalled because it is waiting for a memory operand to be loaded into its register file (md_stall condition), then the amount of time the Ebox remains stalled is related to how quickly the Mbox can return the data. In order to minimize Mbox read latency, a two-cycle pipeline organization is used. This organization allows requested read data to be returned in a minimum of two cycles after the read reference is shipped to the Mbox.

The timing diagram below illustrates the basic sequential processing within the two-cycle Mbox pipeline.

Figure 12–12: Basic Mbox Timing

```
                    S5 PIPE                                       S6 PIPE
|                                      |                                      |
|-----------|------------|-----------|------------|-----------|------------|------------|------------|
|                                      |                                      |

 <----- TB LOOKUP ------>                 <------- ROTATE & RETURN DATA -------->
                                                    TO IBOX & EBOX
          <---------------- Pcache ACCESS ---------------->
                 (read, write, fill, invalidate)
```

At the start of the S5 cycle, the Mbox drives the highest priority reference into the S5 pipe. The Mbox arbitration logic determines which reference should be driven into S5 at the end of the previous cycle. The first half of the S5 cycle is used to translate the virtual address to a physical address via the TB.

The Pcache access is started during phase two of S5 and continues into the first quarter of S6.

If the reference should cause data to be returned to the Ibox or Ebox, the first three phases of the S6 cycle is used to rotate the read data (if the data is not right-justified) and to transfer the data back to the Ibox and/or Ebox.

Thus, assuming an aligned read reference is issued in cycle x by the Ibox or Ebox, the Mbox can return the requested data in cycle x+2 provided that 1) the translated read address was cached in the TB, 2) no memory management exceptions occurred, 3) the read data was cached in the Pcache, and 4) no other higher priority or pending reference inhibited the immediate processing of this read.

12.3.3 REFERENCE ORDER RESTRICTIONS

Due to the macropipeline structure of NVAX, the Mbox can receive "out-of-order" references from the Ibox and Ebox. That is, the Ibox can send a reference corresponding to an opcode decode before the Ebox has sent all references corresponding to the previous opcode. Issuing references "out-of-order" in a macropipeline introduces complexities in the Mbox to guarantee that all references will be processed correctly within the context of the VAX architecture, the NVAX macropipeline, and the Mbox hardware. Many of these complexities take the form of restrictions on how and when references can be processed by the Mbox.

The following synchronization example is useful to illustrate several of the reference order restrictions.

Figure 12–13: 2 Processor Synchronization Example

```
 PROCESSOR 1                 PROCESSOR 2
 -----------                 -----------

  MOVL #1,C               10$ BLBC T,10$
  MOVL #1,T                   MOVL C,R0
```

This example illustrates two processors operating in a multiprocessor environment. Initially, processor 1 owns the critical section corresponding to memory location T. Processor 1 will modify memory location C since it currently has ownership. Subsequently, processor 1 will release ownership by writing a 1 into T. Meanwhile, processor 2 is "spinning" on location T waiting for T to become non-zero. Once T is non-zero, processor 2 will read the value of C.

Note that this example is not the preferred way to implement synchronization. A better way would be to use VAX interlocked instructions which guarantee atomicity. This is, however, a valid example under current SRM rules because it does not disallow an NVAX multiprocessor system from supporting this synchronization structure.

The following discussion explains the Mbox reference order restrictions.

12.3.3.1 No D-stream hits under D-stream misses

"No D-stream hits under D-stream misses" refers to the fact that the Mbox will not allow a D-stream read reference, which hits in the Pcache, to execute as long as requested data for a previous D-stream read has not yet been supplied.

Consider the code that processor 2 executes in the example above. If the Mbox allowed D-stream hits under D-stream misses, then it is possible for the Ibox read of C to hit in the Pcache during a pending read miss sequence to T. In doing so, the Mbox could supply the value of C before processor 1 modified C. Thus, processor 2 would get the old C with the new T causing the synchronization code to operate improperly.

Note that, while D-stream hits under D-stream misses is prohibited, the Mbox will execute a D-stream hit under a D-stream fill operation. In other words, the Mbox will supply data for a read which hit in the Pcache while a Pcache fill operation to a previous missed read is in progress, provided that the missed read data has already been supplied.

I-stream and D-stream references are handled independently of each other. That is, I-stream processing can proceed regardless of whether a D-stream miss sequence is currently executing, assuming there is not Pcache index conflict.

12.3.3.2 No I-stream hits under I-stream misses

This is the analogous case for I-stream read references. This restriction is necessary to guarantee that the Ibox will always receive its requested I-stream reference first, before any other I-stream data is received.

12.3.3.3 Maintain the order of writes

Consider the example shown above. If the Mbox of processor 1 were to reorder the write to C with the write to T, then processor 2 could read the old value of C before processor 1 updated C. Thus, the Mbox must never re-order the sequence of writes generated by the Ebox microcode.

12.3.3.4 Maintain the order of Cbox references

Again consider the example above. Processor 2 will receive an invalidate for C as a result of the write done by processor 1 in the MOVL #1,C instruction. If this invalidate were not to be processed until after processor 2 did the read of C then, the wrong value of C has been placed in R0.

Strictly speaking we must guarantee that the invalidate to C happens before the read of C. However, since C may be in the Pcache of processor 2, there is nothing to stop the read of C from occurring before the invalidate is received. Thus from the point of view of processor 2, the real restriction here is that the invalidate to C must happen before the invalidate to T which must happen before the READ of T which causes processor 2 to fall throught the loop. As long as the Mbox does not re-order Cbox references, the invalidate to C will occur before a non-zero value of T is read.

12.3.3.5 Preserve the order of Ibox reads relative to any pending Ebox writes to the same quadword address

Consider the following example:

Figure 12–14: Memory Scoreboard Example

```
MOVL #1,C
MOVL C,R0
```

In the NVAX macropipeline, the Ibox prefetches specifier operands. Thus, the Mbox receives a read of C corresponding to the "MOVL C,R0" instruction. This read, however, cannot be done until the write to C from the previous instruction completes. Otherwise, the wrong value of C will be read.

In general, the Mbox must ensure that Ibox reads will only be executed once all previous writes to the same location have completed.

12.3.3.6 I/O Space Reads from the Ibox must only be executed when the Ebox is executing the corresponding instruction

Unlike memory reads, reads to certain I/O space addresses can cause state to be modified. As a result, these I/O space reads must only be done in the context of the instruction execution to which the read corresponds. Due to the macropipeline structure of NVAX, the Ibox can issue an I/O space read to prefetch an operand of an instruction which the Ebox is not currently executing. Due to branches in instruction execution, the Ebox may in fact never execute the instruction corresponding to the I/O space read. Therefore, in order to prevent improper state modification, the Mbox must inhibit the processing of I/O space reads issued by the Ibox until the Ebox is actually executing the instruction corresponding to the I/O space read.

12.3.3.7 Reads to the same Pcache block as a pending read/fill operation must be inhibited

The organization of the Pcache is such that one address tag corresponds to four subblock valid bits. Therefore, the validated contents of all four subblocks must always correspond to the tag address. If two distinct Pcache fill operations are simultaneously filling the same Pcache block, it is possible for the fill data to be intermixed between the two fill operations. As a result, an IREAD to the same Pcache block as a pending D-stream read/fill is inhibited until the pending read/fill operation completes. Similarly, a D-stream read to the same Pcache block as a pending I-stream read/fill is also inhibited until the fill completes.

12.3.3.8 Writes to the same Pcache block as a pending read/fill operation must be inhibited until the read/fill operation completes

As in the above, this restriction is necessary in order to guarantee that all valid subblocks contain valid up-to-date data. Consider the following situation. The Mbox executes a write to an invalid subblock of a Pcache block which is currently being filled. One cycle later, the cache fill to that same subblock arrives at the Pcache. Thus, the latest subblock data, which came from the write, is overwritten by older cache fill data. This subblock is now marked valid with "old" data. To avoid this situation, writes to the same Pcache block as a pending read/fill operation are inhibited until the cache fill sequence completes.

12.3.4 REFERENCE ARBITRATION

The Mbox maintains seven different reference storage devices in S5. The purpose of these devices is to buffer pending references, which originate from different sections of the chip, until they can be processed by the Mbox. In order to optimize performance of the NVAX pipeline, and to maintain functional correctness of reference processing in light of the Mbox hardware configuration and reference order restrictions, the Mbox services references from these queues in a prioritized fashion.

12.3.4.1 Arbitration Priority

During every Mbox cycle, the reference arbitration logic is responsible for determining which unserviced references should be processed next cycle. The reference sources are listed below from highest to lowest priority:

1. CBOX_LATCH
2. RTY_DMISS_LATCH
3. MME_LATCH
4. VAP_LATCH
5. EM_LATCH
6. SPEC_QUEUE
7. IREF_LATCH
8. nothing can be driven ==> Mbox drives a NOP command into S5

This prioritized scheme does not directly indicate which pending reference will be driven next, but instead indicates in what order the pending references should be tested to determine which one will be processed. Conceptually, the highest pending reference which satisfies all conditions for driving the reference is the one which is allowed to execute during the subsequent cycle.

The rationale behind this priority scheme can be explained as follows. All references coming from the Cbox are always serviced as soon as they are available. Since Cbox references are guaranteed to complete in S5 in one cycle, we eliminate the need to queue up Cbox references and to provide a back-pressure mechanism to notify the Cbox to stop sending references.

A D-stream read reference in the RTY_DMISS_LATCH is guaranteed to have cleared all potential memory management problems. Therefore, any reference stored in this latch is the second consideration for processing.

If a reference related to memory management processing is pending in the MME_LATCH, it is given priority over the remaining four sources because the Mbox is designed to clear all memory management exceptions through the use of the MME_LATCH before normal processing can resume.

The VAP_LATCH stores the second reference of an unaligned reference pair. Since we desire to complete the entire unaligned reference before starting another reference, the VAP_LATCH has next highest priority in order to complete the unaligned sequence that was initiated from a reference of lesser priority.

The EM_LATCH stores references from the Ebox. It is given priority over the SPEC_QUEUE and IREF_LATCH sources because Ebox references are physically further along in the pipe than Ibox references. The presumed implication of this fact is that the Ebox has a more immediate need to satisfy its reference requests than the Ibox, since the Ebox is always performing real work and the Ibox is prefetching operands that may, in fact, never be used.

The SPEC_QUEUE stores Ibox operand references. It is next in line for consideration. The SPEC_QUEUE has priority over the IREF_LATCH because specifier references are again considered further along in the pipeline than I-stream prefetching.

If no other reference can currently be driven, the IREF_LATCH can drive an I-stream read reference in order to supply data to the Ibox.

If no reference can currently be driven into S5, the Mbox automatically drives a NOP command.

12.3.4.2 Arbitration Algorithm

Based on the priority scheme discussed above, the arbitration logic tests each reference to see whether it can be processed next cycle by evaluating the current state of the Mbox. The test associated with each latch is described below:

- CBOX_LATCH: Since Cbox references always want to be processed immediately, a validated CBOX_LATCH always causes the Cbox reference to be driven before all other pending references.
- RTY_DMISS_LATCH: A pending D-stream read reference will be driven from this latch once the final D_CF command has been retired from the S5 pipe.
- MME_LATCH: A pending MME reference will be driven when the contents of the MME_LATCH is validated.
- VAP_LATCH: A reference from the VAP_LATCH will be driven provided that the VAP_LATCH is validated.

- EM_LATCH: A reference from the EM_LATCH will be driven provided that the EM_LATCH is validated.
- SPEC_QUEUE: A validated reference in the SPEC_QUEUE will be driven provided that the SPEC_QUEUE has not been stopped due to explicit Ebox writes in progress (see Section 12.3.20).
- IREF_LATCH: A reference from the IREF_LATCH will be driven provided that the IREF_LATCH has not been stopped due to a pending READ_LOCK/WRITE_UNLOCK sequence (See Section 12.3.19.2).

If none of the conditions above are satisfied, the Mbox will drive a NOP command onto M_QUE%S5_CMD_H<4:0> causing the S5 pipe to become idle.

12.3.5 READS

12.3.5.1 Generic Read-hit and Read-miss/Cache_fill Sequences

In order to orient the reader as to how memory reads are processed by the Mbox, this section will describe the "vanilla" read sequence. It does not discuss reads which TB_MISS, or otherwise are stalled for a variety of different reasons.

The byte mask generator generates the corresponding byte mask by looking at M_QUE%S5_VA_H<2:0> and M_QUE%S5_DL_H<1:0> and then drives the byte mask data onto M%S6_BYTE_MASK_H<7:0> during the subsequent cycle. Byte mask data is generated on a read operation in order to supply the byte alignment information to the Cbox on an I/O space read.

When a read reference is initiated in the S5 pipe, the address is translated by the TB (assuming the address was virtual) to a physical address during the first half of the S5 cycle. The Pcache initiates a cache lookup sequence using this physical address during the second half of the S5 cycle. This cache access sequence overlaps into the following S6 cycle. During phase four of the S6 cycle, the Pcache determines whether the read reference is present in its array.

If the Pcache determined that the requested data is present, a "cache hit" or "read hit" condition occurs. In this event, the Pcache drives the requested data onto B%S6_DATA_H<63:0>. The signal, M%CBOX_REF_ENABLE_L, is de-asserted to inform the Cbox that it should not process the S6 read since the Mbox will supply the data from the Pcache.

If the Pcache determined that the requested data is not present, a "cache miss" or "read miss" condition occurs. In this event, the read reference is loaded into the IMISS_LATCH or DMISS_LATCH (depending on whether the read was I-stream or D-stream) and the Cbox is instructed to continue processing the read by the Mbox assertion of M%CBOX_REF_ENABLE_L. At some point later, the Cbox obtains the requested data. The Cbox will then send four quadwords of data using the I_CF (I-stream cache fill) or D_CF (D-stream cache fill) commands. The four cache fill commands together are used to fill the entire Pcache block corresponding to the hexaword read address. In the case of D-stream fills, one of the four cache fill command will be qualified with C%REQ_DQW_H indicating that this quadword fill contains the requested D-stream data corresponding to the quadword address of the read. When this fill is encountered, it will be used to supply the requested read data to the Mbox, Ibox and/or Ebox.

If, however, the physical address corresponding to the I_CF or D_CF command falls into I/O space, only one quadword fill is returned and the data is not cached in the Pcache. Only memory data is cached in the Pcache.

Each cache fill command sent to the Mbox is latched in the CBOX_LATCH. Note that neither the entire cache fill address nor the fill data are loaded into the CBOX_LATCH. The address in the IMISS_LATCH or DMISS_LATCH, together with two quadword alignment bits latched in the CBOX_LATCH are used to create the quadword cache fill address when the cache fill command is executed in S5. When the fill operation propagates into S6, the Cbox drives the corresponding cache fill data onto B%S6_DATA_H<63:0> in order for the Pcache to perform the fill.

12.3.5.1.1 Returning Read Data

Data resulting from a read operation is driven on B%S6_DATA_H by the Pcache (in the cache hit case) or by the Cbox (in the cache miss case). This data is then driven on M%MD_BUS_H<63:0> by the MD_BUS_ROTATOR in right-justified form. The signals M%VIC_DATA_L, M%IBOX_DATA_L, M%IBOX_IPR_WR_H, M%EBOX_DATA_H, M%MBOX_DATA, are conditionally asserted with the data to indicate the destination(s) of the data.

12.3.5.1.1.1 Pcache Data Bypass

In order to return the requested read data to the Ibox and/or Ebox as soon as possible, the Cbox implements a Pcache Data Bypass mechanism. When this mechanism is invoked, the requested read data can be returned one cycle earlier than when the data is driven for the S6 cache fill operation. The bypass mechanism works by having the Mbox inform the Cbox that the next S6 cycle will be idle, and thus the B%S6_DATA_H bus will be available to the Cbox. When the Cbox is informed of the S6 idle cycle, it drives the B%S6_DATA_H bus with the requested read data if read data is currently available (if no read data is available during a bypass cycle, the Cbox drives some indeterminent data and no valid data is bypassed). The read data is then formatted by the MD_BUS_ROTATOR and transferred onto the M%MD_BUS_H to be returned to the Ibox and/or Ebox, qualified by M%VIC_DATA_L, M%IBOX_DATA_L, and/or M%EBOX_DATA_H.

12.3.5.2 I-stream Read Processing

Memory access to all I-stream code is implemented by the Mbox on behalf of the Ibox. The Ibox uses the I-stream data to load its prefetch queue and to fill the VIC (Virtual Instruction Cache).

When the Ibox requires I-stream data which is not stored in the prefetch queue or the VIC, the Ibox issues an I-stream read request which is latched by the IREF_LATCH. The Ibox address is always interpreted by the Mbox as being an aligned quadword address. Depending on whether the read hits or misses in the Pcache, the amount of data returned varies. The Ibox continually accepts I-stream data from the Mbox until the Mbox qualifies I-stream MD_BUS data with the M%LAST_FILL_H signal. M%LAST_FILL_H informs the Ibox that the current fill terminates the initial IREAD transaction.

12.3.5.2.1 I-stream Read Hits

When the requested data hits in the Pcache, the Mbox turns the IREF_LATCH reference into a series of I-stream reads to implement a VIC "fill forward" algorithm. The fill forward algorithm generates increasing quadword read addresses from the original address to the highest quadword address of the original hexaword address. In other words, the Mbox generates read references so that the hexaword VIC block corresponding to the original address is filled from the point of the request to the end of the block. The theory behind this fill forward scheme is that it only makes sense to supply I-stream data following the requested reference since I-stream execution causes monotonically increasing I-stream addresses (neglecting branches).

The fill forward scheme is implemented by the IREF_LATCH. Once the IREF_LATCH read completes in S5, the IREF_LATCH quadword address incrementor modifies the stored address of the IREF_LATCH so that its contents becomes the next quadword IREAD. Once this "new" reference completes in S5, the next IREAD reference is generated. When the IREF_LATCH finally issues the IREAD corresponding to the highest quadword address of the hexaword address, the forward fill process is terminated by invalidating the IREF_LATCH.

12.3.5.2.2 I-stream Read Misses

The fill forward algorithm described above is always invoked upon receipt of an IREAD. However, when one of the IREADs is found to have missed in the Pcache, the subsequent IREAD references are flushed out of the S5 pipe and the IREF_LATCH. The missed IREAD causes the IMISS_LATCH to be loaded and the Cbox to continue processing the read. When the Cbox returns the resulting four quadwords of Pcache data, all four quadwords are transferred back to the Ibox qualified by M%VIC_DATA_L. This in effect, results in a VIC "fill full" algorithm since the entire VIC block will be filled. Fill full is done instead of fill forward because it costs little to implement. The Mbox must allocate a block of cycles to process the four cache fills; therefore, all the Pcache fill data can be shipped to the VIC with no extra cost in Mbox cycles since the M%MD_BUS_H would otherwise be idle during these fill cycles.

Note that the Ibox is unaware of what fill mode the Mbox is currently operating in. The VIC continues to fill I-stream data from the M%MD_BUS_H whenever M%VIC_DATA_L is asserted regardless of the Mbox fill mode. The Mbox asserts the M%LAST_FILL_H signal to the Ibox during the cycle which the Mbox is driving the last I-stream fill to the Ibox. M%LAST_FILL_H informs the Ibox that is is receiving the final VIC fill this cycle and that it should not expect any more. In fill forward mode, the Mbox asserts M%LAST_FILL_H when the quadword alignment equals 11 (i.e. the upper-most quadword of the hexaword). In fill full mode, the Mbox receives the last fill information from the Cbox and transfers it to the Ibox through the M%LAST_FILL_H signal.

It is possible to start processing I-stream reads in fill forward mode, but then switch to fill full. This could occur because one of the references in the chain of fill forward IREADs misses due to a recent invalidate or due to displacement of Pcache I-stream data by a D-stream cache fill. In this case, the Ibox will receive more than four fills but will remain in synchronization with the Mbox because it continually expects to see fills until M%LAST_FILL_H is asserted.

12.3.5.2.3 I/O Space I-stream Reads

See Section 12.3.5.4.

12.3.5.3 D-stream Read Processing

Memory access to all D-stream references is implemented by the Mbox on behalf of the Ibox (for specifier processing), the Mbox (for PTE references), and the Ebox (for all other D-stream references).

In general D-stream read processing behaves the same way as I-stream read processing except that there is no fill forward or fill full scheme. In other words, only the requested data is shipped to the initiator of the read. From the Pcache point of view, however, a D-stream fill full scheme is implemented since four D_CF commands are still issued to the Pcache.

D-stream reads can have a data length of byte, word, longword or quadword. With the exception of the cross-page check function, a quadword read is treated as if its data length were a longword. Thus a D-stream quadword read returns the lower half of the referenced quadword. The source of most D-stream quadword reads is the Ibox. The Ibox will issue a D-stream longword read to the upper half of the referenced quadword immediately after issuing the quadword read. Thus, the entire quadword of data is accessed by two back-to-back D-stream read operations.

A DREAD_LOCK command always forces a Pcache read miss sequence regardless of whether the referenced data was actually stored in the Pcache. This is necessary in order that the read propagate out to the Cbox so that the memory lock/unlock protocols can be properly processed.

12.3.5.3.1 Reads under Fills

The Mbox will attempt to process a DREAD after the requested fill of a previous D-stream fill sequence has completed. This mechanism, called "reads under fills" is done to try to return read data to the Ibox and/or Ebox as quickly as possible, without having to wait for the previous fill sequence to complete.

If the attempted read hits in the Pcache, the data is returned and the read completes. If the read misses in the S6 pipe, the corresponding fill sequence is not immediately initiated for two reasons:

- A D-stream cache fill sequence for this read cannot be started because the DMISS_LATCH is full corresponding to the currently outstanding cache fill sequence.
- The D-stream read may hit in the Pcache once the current fill sequence completes because the current fill sequence may supply the data necessary to satisfy the new D-stream read.

Because this DREAD has already propagated through the S5 pipe, the read must be stored somewhere in order that it can be restarted in S5. The RTY_DMISS_LATCH is the mechanism by which the S6 read is saved and restarted in the S5 pipe.

Once the read is stored in the RTY_DMISS_LATCH, it will be retried in S5 after the final D_CF reference is retired from S5 (the final D_CF completes the previous D-stream fill sequence). The RTY_DMISS_LATCH is invalidated when the retried reference is retired from S5.

12.3.5.4 I/O Space Reads

I/O space reads are defined as reads which address I/O space. Therefore, a read is an I/O read when the physical address bits, addr<31:29>, are set. I/O space reads are treated by the Mbox in exactly the same way as any other read, except for the following differences:

- I/O space data is never cached in the Pcache. Therefore, an I/O space read always generates a read-miss sequence and causes the Cbox to process the reference.
- Unlike, a memory space miss sequence, which returns a hexaword of data via four I_CF or D_CF commands, an I/O space read returns only one piece of data via one I_CF or D_CF command. Thus the Cbox always asserts C%LAST_FILL_H on the first and only I_CF or D_CF I/O space operation. If the I/O space read is D-stream, the returned D_CF data is always less than or equal to a longword in length.

- I/O space D-stream reads are never prefetched ahead of Ebox execution. An I/O space D-stream read issued from the Ibox is only processed when the Ebox is known to be stalling on that particular I/O space read (see Section 12.3.18.1.1).

NVAX RESTRICTION

I-stream I/O space reads must return a quadword of data. Execution of an I-stream I/O space read which does not return a quadword of data is unpredicatable.

12.3.6 WRITES

All writes are initiated by the Mbox on behalf of the Ebox. The Ebox microcode is capable of generating write references with data lengths of byte, word, longword, or quadword. With the exception of cross-page checks (see Section 12.5.1.5.4), the Mbox treats quadword write references as longword write references because the Ebox datapath only supplies a longword of data per cycle. Ebox writes can be unaligned.

The Mbox performs the following functions during a write reference:

- Memory Management checks: The Mbox checks to be sure the page or pages referenced have the appropriate write access and that the valid virtual address translations are available. (See Section 12.5)
- The supplied data is properly rotated to the memory aligned longword boundary.
- Byte Mask Generation: The Mbox generates the byte mask of the write reference by examining the write address and the data length of the reference.
- Pcache writes: The Pcache is a write-through cache. Therefore, writes are only written into the Pcache if the write address matches a validated Pcache tag entry.

  The one exception to this rule is when the Pcache is configured in force D-stream hit mode. In this mode, the data is always written to the Pcache regardless of whether the tag matches or mismatches.
- All write references which pass memory management checks are transferred to the Cbox via B%S6_DATA_H<63:0>. The Cbox is responsible for processing writes in the Bcache and for controlling the protocols related to the write-back memory subsystem.

When write data is latched in the EM_LATCH, the 4-way byte barrel shifter associated with the EM_LATCH rotates the EM_LATCH data into proper alignment based on the lower two bits of the corresponding address. The diagram below illustrates the barrel shifter function:

Figure 12–15: Barrel Shifter Function

```
original                     +-----+-----+-----+-----+
4 bytes of                   |  4  |  3  |  2  |  1  |
Ebox data                    +-----+-----+-----+-----+

barrel shifter               +-----+-----+-----+-----+
output when                  |  3  |  2  |  1  |  4  |
M_QUE%S5_VA_H<1:0> = 01      +-----+-----------+-----+

barrel shifter               +-----+-----+-----+-----+
output when                  |  2  |  1  |  4  |  3  |
M_QUE%S5_VA_H<1:0> = 10      +-----+-----+-----+-----+

barrel shifter               +-----+-----+-----+-----+
output when                  |  1  |  4  |  3  |  2  |
M_QUE%S5_VA_H<1:0> = 11      ------------------------
```

NVAX Pcache –
I + D stream
Istream processing
proceed independently
of Dstream processing –
But D blocks on D
and I blocks on I

The result of this data rotation is that all bytes (
relative to memory longword boundaries.

When write data is driven from the EM_LATCH, M_
of the barrel shifter so that data will always be pr(

Note that, while the **M_QUE%S5_DATA_H** bus is a
quadword wide. **B%S6_DATA_H** is a quadword wi
The quadword access size facilitates Pcache and
half of **B%S6_DATA_H**<63:0> is ever used to write the Pcache since all write
a longword or less of data. When a write reference propagates from S5 to S6, the longword aligned data on **M_QUE%S5_DATA_H**<31:0> is transferred onto both the upper and lower halves of **B%S6_DATA_H**<63:0> to guarantee that the data is also quadword aligned to the Pcache and Cbox. The byte mask corresponding to the reference will control which bytes of **B%S6_DATA_H**<63:0> actually get written into the Pcache or Bcache.

Write references are formed through two distinct mechanisms described below.

12.3.6.1 Destination Specifier Writes

Destination specifier writes are those writes which are initiated by the Ibox upon decoding a destination specifier of an instruction. When a destination specifier to memory is decoded, the Ibox issues a reference packet corresponding to the destination address. Note that no data is present in this packet because the data is generated when the Ebox subsequently executes the instruction. The command field of this packet is either a DEST_ADDR command (when the specifier had access type of write) or a DREAD_MODIFY command (when the specifier had access type of modify).

The address of this command packet is translated by the TB, memory management access checks are performed, and the corresponding byte mask is generated. The physical address, DL and other qualifer bits are loaded into the PA_QUEUE. When the DEST_ADDR command completes in S5, it is turned into a NOP command in S6 because no further processing can take place without the actual write data.

When the Ebox executes the opcode corresponding to the Ibox destination specifier, the corresponding memory data to be written is generated. This data is sent to the Mbox by a STORE command. The STORE packet contains only data. When the Mbox executes the STORE command in S5, the corresponding PA_QUEUE packet is driven into the S5 pipe. The data in the EM_LATCH is rotated into proper longword alignment using the byte rotator and the lower two bits of the corresponding PA_QUEUE address and are then driven into S5. In effect, the DEST_ADDR and STORE commands are merged together to form a complete physical address WRITE operation. This WRITE operation propagates through the S5/S6 pipeline to perform the write in the Pcache (if the address hits in the Pcache) and in the memory subsystem.

12.3.6.2 Explicit Writes

The term explicit writes defines writes generated solely by the Ebox. That is, writes which do not result from the Ibox decoding a destination specifier but rather writes which are explicitly initiated and fully generated by the Ebox. An example of an explicit write is a write performed during a MOVC instruction. In this example, the Ebox generates the virtual write address of every write as well as supplying the corresponding data. The PA_QUEUE is never involved in processing an explicit write.

Explicit writes are transferred to the Mbox in the form of a WRITE command issued by the Ebox. These writes directly execute in S5 and S6 in the same manner as when a write packet is formed from the PA_QUEUE contents and the STORE data.

12.3.6.3 Writes to I/O Space

I/O space writes are defined as a write command which addresses I/O space. Therefore, a write is an I/O space write when the physical address bits, addr<31:29>, are set. I/O space writes are treated by the Mbox in exactly the same way as any other write, except for the following differences:

- I/O space data is never cached in the Pcache; therefore, an I/O space write always misses in the Pcache.

12.3.6.4 Byte Mask Generation

Since memory is byte-addressable, all memory storage devices must be able to selectively write specified bytes of data without writing the entire set of bytes made available to the storage device.

The byte mask field of a write reference packet specifies which bytes within the quadword Pcache access size get written. The byte mask is generated in the Mbox by the byte mask generation logic based on M_QUE%S5_VA_H<2:0> and the data length of the reference.

Byte mask data is generated on a read as well as a wriate in order to supply the byte alignment information to the Cbox on an I/O space read. The following table illustrates the behavior of the byte mask generator for all aligned reads and writes:

Table 12–2: Byte Mask Logic for Aligned References

addr<2:0>	BM (DL=byte)	BM (DL=word)	BM (DL=long)	BM (DL=quad)
000	00000001	00000011	00001111	00001111
001	00000010	00000110	00011110	00011110
010	00000100	00001100	00111100	00111100
011	00001000	00011000	01111000	01111000
100	00010000	00110000	11110000	11110000
101	00100000	01100000	unaligned	unaligned
110	01000000	11000000	unaligned	unaligned
111	10000000	unaligned	unaligned	unaligned

See Section 12.3.17.3 for a description of byte mask generator for unaligned references.

12.3.7 IPR PROCESSING

12.3.7.1 MBOX IPRs

The Mbox maintains the following internal processor registers:

Table 12–3: Mbox IPRs

Register Name	IPR Address (in hex)
MP0BR (Mbox P0 Base Register)[1]	E0
MP0LR (Mbox P0 Length Register)[1]	E1
MP1BR (Mbox P1 Base Register)[1]	E2
MP1LR (Mbox P1 Length Register)[1]	E3
MSBR (Mbox System Base Register)[1]	E4
MSLR (Mbox System Length Register)[1]	E5
MMAPEN (Map Enable Bit)[1]	E6
PAMODE (Address Mode)	E7
MMEADR (MME Faulting Address Register)[1]	E8
MMEPTE (PTE Address Register)[1]	E9
MMESTS (status of memory management exception)[1]	EA
TBADR (address of reference causing TB parity error)	EC
TBSTS (status of TB parity error)	ED
PCADR (address of reference causing Pcache parity error)	F2
PCSTS (status of Pcache parity error and PTE hard errors)	F4
PCCTL (control state of Pcache operation)	F8

[1] Testability and diagnostic use only; not for software use in normal operation.

Table 12–3 (Cont.): Mbox IPRs

Register Name	IPR Address (in hex)
PCTAG	01800000..01801FE(
PCDAP	01C00000..01C01FF

The first thirteen IPRs listed above (memory management IPRs) are stored in the S5 pipe in the register file of the MME_DATAPATH. All other IPRs are stored in the S6 pipe. Note that when an Mbox IPR, other than a Pcache tag, is addressed, the actual IPR address is received on **M_QUE%S5_VA_H<9:2>** (the table above is written such that all addresses start at bit<0>).

The following is the format description of each Mbox IPR. Each format illustrates the format visible at the programmer level. The formats do not necessarily illustrate the internal hardware storage format.

Figure 12–16: IPR E0 (hex), MP0BR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1| 0|          system virtual page address of P0 page table          | 0| 0| 0| 0| 0| 0| 0| 0| 0|:MP0BR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 12–17: IPR E1 (hex), MP0LR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0|              length of P0 page table in longwords              |:MP0LR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 12–15: Barrel Shifter Function

```
original                   +-----+-----+-----+-----+
4 bytes of                 |  4  |  3  |  2  |  1  |
Ebox data                  +-----+-----+-----+-----+

barrel shifter             +-----+-----+-----+-----+
output when                |  3  |  2  |  1  |  4  |
M_QUE%S5_VA_H<1:0> = 01    +-----+-----+-----+-----+

barrel shifter             +-----+-----+-----+-----+
output when                |  2  |  1  |  4  |  3  |
M_QUE%S5_VA_H<1:0> = 10    +-----+-----+-----+-----+

barrel shifter             +-----+-----+-----+-----+
output when                |  1  |  4  |  3  |  2  |
M_QUE%S5_VA_H<1:0> = 11    +-----+-----+-----+-----+
```

The result of this data rotation is that all bytes of data are now in the correct byte positions relative to memory longword boundaries.

When write data is driven from the EM_LATCH, M_QUE%S5_DATA_H<31:0> is driven by the output of the barrel shifter so that data will always be properly aligned to memory longword addresses.

Note that, while the M_QUE%S5_DATA_H bus is a longword wide, the B%S6_DATA_H bus is a quadword wide. B%S6_DATA_H is a quadword wide due to the quadword Pcache access size. The quadword access size facilitates Pcache and VIC fills. However for all writes, at most half of B%S6_DATA_H<63:0> is ever used to write the Pcache since all write commands modify a longword or less of data. When a write reference propagates from S5 to S6, the longword aligned data on M_QUE%S5_DATA_H<31:0> is transferred onto both the upper and lower halves of B%S6_DATA_H<63:0> to guarantee that the data is also quadword aligned to the Pcache and Cbox. The byte mask corresponding to the reference will control which bytes of B%S6_DATA_H<63:0> actually get written into the Pcache or Bcache.

Write references are formed through two distinct mechanisms described below.

12.3.6.1 Destination Specifier Writes

Destination specifier writes are those writes which are initiated by the Ibox upon decoding a destination specifier of an instruction. When a destination specifier to memory is decoded, the Ibox issues a reference packet corresponding to the destination address. Note that no data is present in this packet because the data is generated when the Ebox subsequently executes the instruction. The command field of this packet is either a DEST_ADDR command (when the specifier had access type of write) or a DREAD_MODIFY command (when the specifier had access type of modify).

The address of this command packet is translated by the TB, memory management access checks are performed, and the corresponding byte mask is generated. The physical address, DL and other qualifer bits are loaded into the PA_QUEUE. When the DEST_ADDR command completes in S5, it is turned into a NOP command in S6 because no further processing can take place without the actual write data.

When the Ebox executes the opcode corresponding to the Ibox destination specifier, the corresponding memory data to be written is generated. This data is sent to the Mbox by a STORE command. The STORE packet contains only data. When the Mbox executes the STORE command in S5, the corresponding PA_QUEUE packet is driven into the S5 pipe. The data in the EM_LATCH is rotated into proper longword alignment using the byte rotator and the lower two bits of the corresponding PA_QUEUE address and are then driven into S5. In effect, the DEST_ADDR and STORE commands are merged together to form a complete physical address WRITE operation. This WRITE operation propagates through the S5/S6 pipeline to perform the write in the Pcache (if the address hits in the Pcache) and in the memory subsystem.

12.3.6.2 Explicit Writes

The term explicit writes defines writes generated solely by the Ebox. That is, writes which do not result from the Ibox decoding a destination specifier but rather writes which are explicitly initiated and fully generated by the Ebox. An example of an explicit write is a write performed during a MOVC instruction. In this example, the Ebox generates the virtual write address of every write as well as supplying the corresponding data. The PA_QUEUE is never involved in processing an explicit write.

Explicit writes are transferred to the Mbox in the form of a WRITE command issued by the Ebox. These writes directly execute in S5 and S6 in the same manner as when a write packet is formed from the PA_QUEUE contents and the STORE data.

12.3.6.3 Writes to I/O Space

I/O space writes are defined as a write command which addresses I/O space. Therefore, a write is an I/O space write when the physical address bits, addr<31:29>, are set. I/O space writes are treated by the Mbox in exactly the same way as any other write, except for the following differences:

- I/O space data is never cached in the Pcache; therefore, an I/O space write always misses in the Pcache.

12.3.6.4 Byte Mask Generation

Since memory is byte-addressable, all memory storage devices must be able to selectively write specified bytes of data without writing the entire set of bytes made available to the storage device.

The byte mask field of a write reference packet specifies which bytes within the quadword Pcache access size get written. The byte mask is generated in the Mbox by the byte mask generation logic based on M_QUE%S5_VA_H<2:0> and the data length of the reference.

Byte mask data is generated on a read as well as a wriate in order to supply the byte alignment information to the Cbox on an I/O space read. The following table illustrates the behavior of the byte mask generator for all aligned reads and writes:

Figure 12–18: IPR E2 (hex), MP1BR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1| 0|        system virtual page address of P1 page table        | 0| 0| 0| 0| 0| 0| 0| 0| 0|:MP1BR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 12–19: IPR E3 (hex), MP1LR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0|          length of (2**21) - P1 page table in longwords       |:MP1LR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 12–20: IPR E4 (hex), MSBR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|              physical page address of system page table              | 0| 0| 0| 0| 0| 0| 0| 0| 0|:MSBR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 12–21: IPR E5 (hex), MSLR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0|          length of system page table in longwords             |:MSLR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 12–22: IPR E6 (hex), MMAPEN

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| M|:MMAPEN
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Table 12–4: MMAPEN Field Descriptions

Name	Extent	Type	Description
M	0	RW	When 0, disables Mbox memory management. When 1, enables Mbox memory management.

Figure 12–23: IPR E7 (hex), PAMODE

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0|  |:PAMODE
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                                                              |
                                                                                  MODE--------+
```

Table 12–5: PAMODE Field Descriptions

Name	Extent	Type	Description
MODE	0	RW	When 0, maps addresses from a 30-bit physical address space. When 1, maps addresses from a 32-bit physical address space.

Figure 12–24: IPR E8 (hex), MMEADR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                            address associated with recorded MME fault                         |:MMEADR
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Figure 12–25: IPR E9 (hex), MMEPTE

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                PTE address associated with an address corresponding to a modify fault         |:MMEPTE
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Figure 12–26: IPR EA (hex), MMESTS

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|           |    SRC    | 0| 0| 0| 0| 0| 0| 0| 0| 0| 0|FAULT| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| M|   |LV|:MMESTS
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
<---+---->                                                                                  |
    |                                                                                       |
    +---- LOCK                                                                    PTE_REF--+
```

Table 12–6: MMESTS Field Descriptions

Name	Extent	Type	Description
LV	0	RO	Indicates ACV fault occurred due to length violation.
PTE_REF	1	RO	Indicates ACV/TNV fault occurred on PTE reference corresponding to MMEADR.
M	2	RO	Indicates corresponding reference had write or modify intent.
FAULT	15:14	RO	Indicates nature of memory management fault. See Fault bit encodings below
SRC	28:26	RO	Complemented shadow copy of LOCK bits. However, the SRC bits do not get reset when the LOCK bits are cleared.
LOCK	31:29	RO,0	Indicates the lock status of MMESTS. See LOCK encodings below. This field is cleared on **E%FLUSH_MBOX_H**.

See Section 12.5.1.5.3.5 for information on how these fields are encoded.

Figure 12–27: IPR EC (hex), TBADR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                  virtual address associated with the recorded TB parity error                 |:TBADR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 12–28: IPR ED (hex), TBSTS

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  SRC   | 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0|      CMD     |  |  |  |  |:TBSTS
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                                                       |  |  |  |
                                                                      EM_VAL-----------+  |  |  |
                                                                      TPERR---------------+  |  |
                                                                      DPERR------------------+  |
                                                                      LOCK----------------------+
```

Table 12–7: TBSTS Field Descriptions

Name	Extent	Type	Description
LOCK	0	WC	Lock Bit. When set, validates TBSTS contents and prevents any other field from further modification. When clear, indicates that no TB parity error has been recorded and allows TBSTS and TBADR to be updated.
DPERR	1	RO	Data Error Bit. When set, indicates a TB data parity error.
TPERR	2	RO	Tag Error Bit. When set, indicates a TB tag parity error.
EM_VAL	3	RO	EM_LATCH valid bit. Indicates if EM_LATCH was valid at the time of the error TB parity error detection. This helps the software error handler determine if a write operation may have been lost due to the TB parity error.
CMD	8:4	RO	S5 command corresponding to TB parity error.
SRC	31:29	RO	Indicates the original source of the reference causing TB parity error.

See Section 12.6.4.1 for information on how these fields are encoded.

Figure 12–29: IPR F2 (hex), PCADR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|      quadword physical address associated with the recorded Pcache parity error      | 0| 0| 0|:PCADR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 12–30: IPR F4 (hex), PCSTS

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1|  |  |     CMD      |  |  |  |  |:PCSTS
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                                |  |                 |  |  |  |
                                                 PTE_ER---------+  |                 |  |  |  |
                                                 PTE_ER_WR---------+                 |  |  |  |
                                                 LEFT_BANK---------------------------+  |  |  |
                                                 RIGHT_BANK-----------------------------+  |  |
                                                 DPERR-------------------------------------+  |
                                                 LOCK-----------------------------------------+
```

Table 12–8: PCSTS Field Descriptions

Name	Extent	Type	Description
LOCK	0	WC	Lock Bit. When set, validates PCSTS<8:1> contents and prevents modification of these fields. When clear, invalidates PCSTS<8:1> and allows these fields and PCADR to be updated.
DPERR	1	RO	Data Error Bit. When set, indicates a Pcache data parity error.
RIGHT_BANK	2	RO	Right Bank Tag Error Bit. When set, indicates a Pcache tag parity error on the right bank.
LEFT_BANK	3	RO	Left Bank Tag Error Bit. When set, indicates a Pcache tag parity error on the left bank.
CMD	8:4	RO	S6 command corresponding to Pcache parity error.
PTE_ER_WR	9	WC	Indicates a hard error on a PTE DREAD which resulted from a TB miss on a WRITE or WRITE_UNLOCK.
PTE_ER	10	WC	Indicates a hard error on a PTE DREAD.

Note that the state of PCSTS<31:11> are "don't cares" during an IPR write operation.

Figure 12–31: IPR F8 (hex), PCCTL

```
 31  30  29  28 |27  26  25  24 |23  22  21  20 |19  18  17  16 |15  14  13  12 |11  10  09  08 |07  06  05  04 |03  02  01  00
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1|   |   |    PMM    |   |   |   |   |   |:PCCTL
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                                                                                          |   |               |   |   |   |   |
                                                                             RED_ENABLE---+   |               |   |   |   |   |
                                                                             ELEC_DISABLE-----+               |   |   |   |   |
                                                                             P_ENABLE-------------------------+   |   |   |   |
                                                                             BANK_SEL-----------------------------+   |   |   |
                                                                             FORCE_HIT--------------------------------+   |   |
                                                                             I_ENABLE ------------------------------------+   |
                                                                             D_ENABLE ----------------------------------------+
```

Table 12–9: PCCTL Field Descriptions

Name	Extent	Type	Description
D_ENABLE	0	RW,0	When set, enables Pcache for all INVAL operations and for all D-stream read/write/fill operations, qualified by other control bits. When clear, forces a Pcache miss on all Pcache D-stream read/write/fill operations. Note, however, that an ACV/TNV/M=0 condition overrides a desasserted D_ENABLE in that it will force a Pcache hit condition with D_ENABLE=0.
I_ENABLE	1	RW,0	When set, enables Pcache processing of INVAL, IREAD and I_CF commands. When clear, forces a Pcache miss on IREAD operations and prevents state modification due to an I_CF operation. Note, however, that an ACV/TNV/M=0 condition overrides a desasserted I_ENABLE in that it will force a Pcache hit condition with I_ENABLE=0.
FORCE_HIT	2	RW,0	When set, forces a Pcache hit on all reads and writes when Pcache is enabled for I or D-stream operation.
BANK_SEL	3	RW,0	When set with FORCE_HIT=1, selects the "right bank" of the addressed Pcache index. When clear with FORCE_HIT=1, selects the "left bank" of the addressed Pcache index. BANK_SEL is a don't care when FORCE_HIT=0. NOTE: BANK_SEL never affects bank selection during IPR reads and IPR writes to the Pcache tags or Pcache data parity bits; bank selection for these commands is always determined by the specified IPR address.
P_ENABLE	4	RW,0	When set, enables detection of Pcache tag and data parity errors. When deasserted, disables Pcache parity error detection.
PMM	7:5	RW,0	Specifies Mbox performance monitor mode (see Section 12.10). Note that this field does not control or affect the operation of the Pcache in any way. PMM is placed in PCCTL for the convenience of the hardware implementation.

Table 12–9 (Cont.): PCCTL Field Descriptions

Name	Extent	Type	Description
ELEC_DISABLE	8	RW,0	When set, the Pcache is disabled electrically to reduce power dissipation. NOTE: This bit should only be set when the Pcache is functionally turned off by the deassertion of both I_ENABLE and D_ENABLE. UNPREDICTABLE operation will result when this bit is set when either I_ENABLE or D_ENABLE is also set. Also note that Pcache tag or parity IPRs will not function properly when this bit is unconditionally set.
RED_ENABLE	9	RO	When set, indicates that one or more Pcache redundancy elements are enabled (see Section 12.4.11 for more information).

Note that the state of PCCTL<31:10> are "don't cares" during an IPR write operation.

Figure 12–32: IPRs 01800000 thru 01801FE0 (hex), PCTAG

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                              tag                          | 1| 1| 1| 1| 1| 1| P| valid bits| A|:PCTAG
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Table 12–10: PCTAG Field Descriptions

Name	Extent	Type	Description
A	0	RW	Allocation Bit corresponding to index of this tag.
valid bits	4:1	RW	Valid Bits corresponding to the four data subblocks. PCTAG<4> corresponds to uppermost quadword in block. PCTAG<1> corresponds to lowermost quadword in block.
P	5	RW	Even Tag Parity
tag	31:12	RW	Tag Data

Note that the state of PCTAG<11:6> are "don't cares" during an IPR write operation.

Figure 12–33: IPRs 01C00000 thru 01C01FF8 (hex), PCDAP

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1|       DATA_PARITY     |:PCDAP
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Table 12–11: PCDAP Field Descriptions

Name	Extent	Type	Description
DATA_PARITY	7:0	RW	Even byte parity corresponding to addressed quadword of data. Bit n represents parity for byte n of addressed quadword.

Note that the state of PCDAP<31:8> are "don't cares" during an IPR write operation.

12.3.7.2 Hardware MBOX IPR Format

The IPR formats listed above reflect the formats used by the programmer to execute IPR read and write operations. However, due to the specific structure of the Mbox memory management datapath, four memory management registers are internally stored in a different format in order to facilitate all length violation checks and P1 space PTE calculations. The following describes the hardware formats of these registers:

Figure 12–34: MP0LR Register

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0|             length of P0 page table in longwords                | 0| 0| 0| 0| 0| 0| 0| 0| 0|:MP0LR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 12–35: MP1LR Register

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|      (length of (2**21) - P1 page table in longwords) + lr_bias     | 0| 0| 0| 0| 0| 0| 0| 0| 0|:MP1LR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 12–36: MSLR Register

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0|                 length of system page table in longwords           | 0| 0| 0| 0| 0| 0| 0| 0| 0|:MSLR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

The re-formating operation necessary to convert the program-level format to the hardware-level format is handled by microcode. When IPR writes are done to these registers, the microcode shifts the length register data 9 bits to the left before delivering the IPR_WRITE reference to the Mbox. In the MP1LR case, the microcode adds a bias value to the data following the shift operation. This is done in order to compensate for the "1" which will occur in virtual_addr<30> position during length check subtraction operations for all P1 space virtual references.

The microcode reverses the format operation to convert the Mbox IPR data back into the program-level format during MxLR IPR_READ operations.

The hardware format for MP1BR is shown below:

Figure 12–37: MP1BR Register

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|        system virtual page address of P1 page table - br_bias       | 0| 0| 0| 0| 0| 0| 0| 0| 0|:MP1BR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Before sending the IPR_WRITE data to the Mbox, the microcode substracts a different bias value from the P1 space base register. This is done in order to compensate for the "1" which will occur in virtual_addr<30> position during P1 space PTE address calculations.

The microcode reverses this format operation to convert the Mbox IPR data back into the program-level format during MP1BR IPR_READ operations.

12.3.7.3 IPR Reads

IPR reads (internal processor register reads) are issued to the Mbox by the Ebox using the IPR_RD command. The Ebox issues an IPR_RD in order to obtain the contents of an NVAX internal processor register existing somewhere in the system other than the Ebox.

12.3.7.3.1 Mbox IPR Reads

When the Ebox issues an IPR_RD to an Mbox S5 IPR, the MME_DATAPATH will respond by accessing the appropriate register and loading it into the data field of the MME_LATCH. The MME_LATCH is then validated with an IPR_DATA command. Subsequently, the IPR_DATA command will execute in the Mbox pipe by passing the requested IPR data back to the Ebox on M%MD_BUS_H<31:0>, qualified by M%EBOX_DATA_H.

All Mbox S6 IPRs return their data directly on M%MD_BUS_H<31:0>, qualified by M%EBOX_DATA_H, during the S6 execution of the IPR_RD command.

Any IPR address in the range E0-FF which is not specified above is called a reserved Mbox IPR (reserved for any future Mbox IPR functional requirements). An IPR_RD to a reserved Mbox IPR will cause the assertion of M%EBOX_DATA_H in order to unstall the Ebox which is waiting for IPR data to be returned. Note however, that the returned data is UNPREDICATABLE.

12.3.7.3.2 Non-Mbox IPR Reads

The Ebox will issue an IPR_RD command to the Mbox to access IPRs existing in other sections of the NVAX computer system. Specifically, IPR_RD commands are issued to address IPRs in the Ibox, Cbox, NDAL and memory subsystem.

IPR_RDs to the Ibox (IPR addresses D0-DF) are treated as NOPs. That is, execution of an Ibox IPR_RD command performs no Mbox function and does not modify any Mbox state. This behavior facilitates the Ebox microcode decode of IPR commands by allowing Ibox IPR_RDs to be issued to the Mbox even though the Mbox does not play a role in returning Ibox IPR data.

IPR_RDs which do not address the Ibox or the Mbox are transferred to the Cbox for further processing by asserting M%CBOX_REF_ENABLE_L when the IPR_RD is in S6. These IPR_RDs are handled by the Mbox in a manner similar to a DREAD which misses in the Pcache. The IPR_RD command is loaded into the DMISS_LATCH as the command is transferred to the Cbox. DMISS_LATCH state is set to indicate that the reference is not cacheable. Subsequently, the Cbox responds to the IPR_RD by sending back the requested data via one D_CF command. The IPR_RD sequence is similar to an I/O space READ miss sequence in that only one D_CF command is sent rather than four, and the returned data is not loaded in the Pcache even though a D_CF command was used to return the data.

12.3.7.4 IPR WRITES

IPR writes (internal processor register writes) are issued to the Mbox by the Ebox using the IPR_WR command. The IPR_WR command modifies the contents of an internal processor register which is located in the Ibox, Mbox, Cbox, NDAL or memory subsystem. The addressed register is modified using the longword of data associated with the IPR_WR command.

12.3.7.4.1 Mbox IPR Writes

All Mbox IPRs located in S5 reside in the MME_DATAPATH. These IPRs are written by the IPR_WR command during the cycle after the IPR_WR executes in S5. All other Mbox IPRs reside in S6 and are written during the cycle when the IPR_WR executes in S6. See Table 12–3 for a description of the Mbox IPR registers.

An IPR_WR to an Mbox reserved IPR causes no action to be taken.

12.3.7.4.2 Non-Mbox IPR Writes

Unlike Ibox IPR reads, the Mbox plays a role in processing Ibox IPR writes. The Mbox recognizes all Ibox IPR writes (addresses D0-DF) and passes the data through the Mbox pipeline onto M%MD_BUS_H<31:0> qualified by M%IBOX_IPR_WR. The Ibox receives the IPR write data and stores it in the Ibox IPR specified by information received directly from the Ebox. Processing Ibox IPR writes via the Mbox allows the M%MD_BUS_H to be used to transfer Ibox IPR write data without the need for a special Ebox-Ibox data bus.

The Mbox asserts M%CBOX_REF_ENABLE_L to the Cbox when the addressed IPR falls outside of the Ibox and Mbox IPR address space. This causes the Cbox to continue processing the IPR_WR.

12.3.8 LOAD_PC

The LOAD_PC command is used to transfer a new Program Counter value from the Ebox to the Ibox via the Mbox. This PC value propagates through the Mbox in order to transfer the Ibox data across M%MD_BUS_H<31:0>. Using the M%MD_BUS_H for this purpose eliminates the need for a special Ebox-Ibox data bus.

The LOAD_PC command operates in a manner identical to an Ibox IPR_WR command. The only difference between a LOAD_PC and an Ibox IPR_WR command is that no IPR address need be decoded. The LOAD_PC command directly specifies the destination of the data as being the Ibox PC.

12.3.9 INVALIDATES

The Pcache must always be a coherent cache with respect to the Bcache. In other words, the Pcache must always contain a strict subset of the data cached in the Bcache. If cache coherency were not maintained, incorrect computational sequences could result from reading "stale" data out of the Pcache in multi-processor system configurations.

An invalidate is the mechanism by which the Pcache is kept coherent with the Bcache. A Pcache invalidate operation occurs when data is displaced from the Bcache or when Bcache data is invalidated. The Cbox initiates an invalidate by specifying a hexaword physical address qualified by the INVAL command. The INVAL command is latched by the Mbox in the CBOX_LATCH.

Execution of an INVAL command guarantees that data corresponding to the specified hexaword address will not be valid in the Pcache. If the hexaword address of the INVAL command does not match to either Pcache tag in the addressed index, no operation takes place. If the hexaword address matches one of the tags, the four corresponding subblock valid bits are cleared to guarantee that any subsequent Pcache accesses of this hexaword will miss until this hexaword is re-validated by a subsequent Pcache fill sequence. If a cache fill sequence to the same hexaword address is in progress when the INVAL is executed, a bit in the corresponding MISS_LATCH is set to inhibit any further cache fills from loading data or validating data for this cache block.

Also note that an assertion of C%CBOX_HARD_ERR_H during a cache fill command causes the cache fill operation to be processed as if it were an INVAL operation.

12.3.10 CACHE FILL COMMANDS

See Section 12.3.5.1 for a discussion of cache fill operations.

12.3.11 MME CHECK COMMANDS

Two commands exist for the purpose of checking references for possible memory management exceptions.

12.3.11.1 MME_CHK

The function of the MME_CHK command is to obtain the allowed access rights for a specified page, and to compare it against an intended access mode specified by **M_QUE%S5_AT_H<1:0>**. The MME_CHK command causes a TB access of the PTE corresponding to the MME_CHK address. If the PTE is not cached in the TB, the Mbox first fetches the PTE from memory. Once the PTE information is accessed, ACV/TNV/M=0 checks are performed. If an ACV, TNV or M=0 fault is detected, the appropriate memory management fault response is invoked (See Section 12.5.1.5.3 for a description of ACV/TNV/M=0 faults).

12.3.11.2 PROBE

The PROBE command is used when the microcode must determine the accessibility of a page before changing any state (e.g. PROBER, PROBEW, CHMx macro instructions). It functions exactly as an MME_CHK command except for three differences:

- If an ACV, TNV, or M=0 condition is detected, no ACV, TNV, or M=0 response is invoked. That is, a PROBE merely detects the condition without actually causing a memory management exception. The PROBE command will update MMESTS based on the probe information if MMESTS is unlocked. However, a PROBE command will never lock MMESTS.
- The PROBE command returns status to the Ebox which indicates the nature of any memory management condition the PROBE may have detected.
- If **M_QUE%S5_AT_H<1:0>**=00 corresponding to the PROBE reference, then the MME_DATAPATH tb_miss sequence is not invoked when the TB detects a miss.

Status is returned to the Ebox on the **M%MD_BUS_H** in the following format:

- **M%MD_BUS_H**<3> is set when the PROBE reference hits in the TB.
- **M%MD_BUS_H**<2> is set when the PROBE reference corresponds to an ACV fault.
- **M%MD_BUS_H**<1> is set when the PROBE reference corresponds to an TNV fault.
- **M%MD_BUS_H**<0> is set when the PROBE reference corresponds to an M=0 fault.
- All other **M%MD_BUS_H** bits are undefined.

NOTE

> One exception to this PROBE status format exists. When **M%MD_BUS_H**<2:0> = 011, the meaning of this code indicates that a TNV has occurred on the PPTE (Process Page Table Entry) corresponding to the PROBE address. It does NOT mean that a TNV and M=0 fault have simultaneously occurred on the PROBE address (this would not make sense).

The following tables summarizes all possible PROBE status encodings.

Table 12–12: Probe Status Encodings

M%MD_BUS_H<3:0>	M_QUE%S5_AT_H<1:0>	Probe Status
X000	at^=00	No fault.
X001	at^=00	Modify fault.
X010	at^=00	TNV fault.
X011	at^=00	TNV fault on PPTE reference.
X100	at^=00	ACV fault.
X101	at^=00	illegal status (will never be generated)
X110	at^=00	illegal status (will never be generated)
X111	at^=00	illegal status (will never be generated)
0XXX	at=00	PROBE missed in TB. Lower three bits are a don't care.
1XXX	at=00	PROBE hit in TB. Lower three bits are a don't care.

If memory management is turned off (i.e. MAPEN=0) execution of the PROBE command returns a status of M%MD_BUS_H<2:0>=0 indicating that no fault was detected (M%MD_BUS_H<3> will vary based on hit/miss TB status).

12.3.12 TB Fills

12.3.12.1 TB Tag Fills

The TB_TAG_FILL command is used in conjunction with the TB_PTE_FILL command to cache a PTE in the TB. The data associated with the TB_TAG_FILL command corresponds to a virtual byte address in some virtual page. The TB_TAG_FILL command causes the page address on M_QUE%S5_VA_H<31:9> of the TB_TAG_FILL data to be written into the tag field of the TB entry pointed to by the NLU TB allocation pointer (see Section 12.5.1.3 for information about the NLU TB allocation pointer). The TB valid bit (TBV) of the entry is cleared.

When TB_TAG_FILLs occur from the MME_LATCH, the tag data is driven onto M_QUE%S5_VA_H in the following format:

Figure 12–38: TB_TAG_FILL Format (from MME_LATCH)

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                VPN                                 | 0| 0| 0| 0| 0| 0| 0| 0| 0|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Table 12–13: TB_TAG_FILL Definition

Name	Extent	Type	Description
VPN	31:9	W	Virtual page address used to fill a TB tag field.

During the TB_TAG_FILL, the TB logic will automatically generate even tag parity corresponding to PTE<31:9>. This parity will be written into the TB during the TB_TAG_FILL operation.

When TB_TAG_FILLs occur from the Ebox, the tag data is supplied from the address field of the EM_LATCH and is driven onto **M_QUE%S5_VA_H** in the following format:

Figure 12–39: TB_TAG_FILL Format (from EM_LATCH): IPR 7E (hex), MTBTAG

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                VPN                                 | 0| 0| 0| 0| 0| 0| 0| 0|TP| :MTBTA
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Table 12–14: MTBTAG Field Descriptions

Name	Extent	Type	Description
TP	0	W	Even tag parity bit.
VPN	31:9	W	Virtual page address used to fill a TB tag field.

In this case, the even tag parity corresponding to the VPN is specified in bit<0> of the data field for the TB_TAG_FILL. This mechanism allows correct or incorrect parity to be deliberately written into the TB tag array for testability purposes by invoking the TB_TAG_FILL operation through the appropriate MTPR instruction.

12.3.12.2 TB PTE Fills

The TB_PTE_FILL operation drives the PTE data onto **M_QUE%S5_VA_H<31:0>** in order that this data can be written into the data array of the TB. The data is written into the entry pointed to by the NLU TB allocation pointer. The TB valid bit (TBV bit) of the entry is set (Note that a TB_TAG_FILL command will not be issued by the Mbox if PTE<31> is clear in order to guarantee that only validated PTEs are ever cached in the TB). The NLU TB allocation pointer is incremented after the fill is done.

When TB_PTE_FILLs occur from the MME_LATCH, the PTE data is driven onto **M_QUE%S5_VA_H** during a TB_PTE_FILL in the following format:

Figure 12–40: TB_PTE_FILL Data Format (from MME_LATCH)

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1|    PROT   | M| 0| 0| 0|                                PFN                                 |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Table 12–15: TB_PTE_FILL Definition

Name	Extent	Type	Description
PFN	22:0	W	Page frame address
0	25:23		Forced to 0 by MME_LATCH
M	26	W	PTE modify bit.
PROT	30:27	W	PTE protection field.
1	31		Valid bit of PTE (must be a "1". See below)

Only bits <30:26>, <22:0> and the corresponding PTE parity bit are actually written into the TB array during a TB_PTE_FILL. TB_PTE_FILLs from the MME_LATCH will only be issued for validated PTEs. Therefore, PTE<31> will always be set. The TB logic will automatically generate even parity to be written during the fill corresponding to PTE<31:0>. Note that the parity generator includes PTE<31> in this calculation even though this bit is not written into the TB. Since PTE<31> is always a "1" during a TB_PTE_FILL, the stored parity can be thought of as odd parity on bits <30:0>.

When TB_PTE_FILLs occur from the EM_LATCH, the PTE data is driven onto **M_QUE%S5_VA_H** during a TB_PTE_FILL in the following format:

Figure 12–41: TB_PTE_FILL Data Format (from EM_LATCH): IPR 7F (hex), MTBPTE

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1|   PROT    | M| 0| P| 0|                                PFN                                 | :MTBPT
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Table 12–16: MTBPTE Field Descriptions

Name	Extent	Type	Description
PFN	22:0	W	Page frame address
0	23		Assumed to be a "0" for parity calculation.
P	24	W	User-settable even parity corresponding to PTE<31:26> and PTE<22:0>.
0	25		Assumed to be a "0" for parity calculation.
M	26	W	PTE modify bit.
PROT	30:27	W	PTE protection field.
1	31		Assumed to be a "1" for parity calculation. (See below)

Bits <30:26>, <22:0> are written into the TB array during a TB_PTE_FILL. Bit<24> is interpreted as the corresponding PTE parity and is directly written into the TB as such. This gives the user the flexibility of writing correct or incorrect PTE parity for testability purposes. Note however that while PTE<31> is not written into the TB, it must be assumed that this bit is set when the user calculates even parity on PTE<31:0>. Similarly, PTE<25> and PTE<23> must be cleared for proper parity calculation.

See Section 12.5.1.5.2 for a description of TB fill sequences.

12.3.13 TBIS

The TBIS (TB Invalidate Single) command invalidates the PTE entry corresponding to the specified virtual address, providing that the PTE is cached in the TB. If the PTE is not cached in the TB, no action is taken.

12.3.14 TBIP

The TBIP (TB Invalidate Process) command invalidates all the PTE entries corresponding to P0 or P1 space translations which are currently cached in the TB. This command is used when the CPU changes process context. It allows a new process translation state to be set up for the new process context without being polluted by old translations corresponding to the old process context. TBIP does not invalidate PTEs corresponding to system space translations because these translations are valid across all processes.

12.3.15 TBIA

The TBIA (TB Invalidate All) command invalidates all PTE entries in the TB and resets the NLU TB allocation pointer to a known state. This is done for CPU initialization purposes, when the operating system reconfigures its system space translations, and when the Mbox clears the TB after encountering a TB parity error.

12.3.16 STOP_SPEC_Q

The STOP_SPEC_Q command is sent by the Ibox to inform the Mbox that no subsequent Ibox specifier references should be processed until the Ebox sends the proper synchronization. This command decrements the SPEC_Q_SYNC_CTR. In all other respects, it is treated as a NOP by the Mbox. See Section 12.3.20 to understand the context of the use of STOP_SPEC_Q.

12.3.17 UNALIGNED REFERENCES

An unaligned reference is a D-stream memory read or memory write reference that refers to data which crosses a quadword-aligned boundary (note that unaligned I/O space references are defined to cause UNPREDICTABLE behavior). A quadword boundary is the appropriate address resolution because the Pcache and Cbox read and write aligned quadwords of data. If a reference crosses a quadword-aligned boundary, the unaligned reference must be translated into two references–one for each distinct quadword memory access.

Detection of an unaligned reference is done in S5 by the unaligned detection logic and is a function of **M_QUE%S5_VA_H**<2:0> and **M_QUE%S5_DL_H**<1:0> of the S5 reference packet. The following table summarizes all possible unaligned configurations:

DL	ADDR<2:0>
word	111
longword	101, 110, 111
quadword	101, 110, 111

When an unaligned D-stream read, STORE or WRITE is detected, the Mbox does the following:

- The address of the unaligned reference is used to reference the aligned quadword corresponding to the lower portion of the data.
- The Mbox generates a second reference corresponding to the aligned quadword corresponding to the upper portion of the reference.
- In the case of reads, once both references have been executed, the requested data is extracted from the two quadwords and aligned to **M%MD_BUS_H**<31:0>.

The implication of unaligned processing by the Mbox is that unaligned references are functionally invisible to the Ibox and Ebox. That is, the Ibox and Ebox can perform reads and writes without regard to alignment. Note that Mbox-generated references and I-stream reads are always aligned references.

12.3.17.1 Unaligned Reads

When an S5 read is determined to be unaligned, the S5 command packet is loaded into the VAP_LATCH. However, **M_QUE%S5_VA_H**<31:0> is not directly loaded. Instead the quadword incrementor associated with the VAP_LATCH increments the **M_QUE%S5_VA_H** quadword address. This new address is loaded and is used to reference the upper half of the unaligned data.

Meanwhile, the current S5 read command is allowed to execute. When this read successfully completes in S5, the VAP_LATCH is validated indicating that it contains the upper half of the unaligned reference and that it can now be executed. Subsequently, the VAP_LATCH reference will be processed in the S5 pipe. Once it successfully completes in S5, the VAP_LATCH is invalidated. Note that if the read originated from the EM_LATCH, the EM_LATCH was invalidated as the first reference of the unaligned pair successfully completed. However, if the read came from the SPEC_QUEUE, the SPEC_QUEUE is not invalidated until the VAP_LATCH reference successfully completes (See Section 12.3.19.1).

When data for the first read is available on **B%S6_DATA_H**<63:0> (either from the Pcache or the Cbox), the data is rotated by the MD_BUS_ROTATOR based on **M%S6_PA_H**<2:0> and latched in the MD_BUS_ROTATOR latches. Since the VAP_LATCH read was executed after the initial read, its data is guaranteed to be available during some cycle after the initial data is latched by the MD_BUS_ROTATOR. When the second data arrives in S6, the data is rotated by the same number of bytes as was done for the first reference. The lower one, two, or three bytes of the **M%MD_BUS_H** is then driven from the MD_BUS_ROTATOR latches which contain valid data from the first reference while the remaining bytes of **M%MD_BUS_H** are driven directly from the rotator. The effect of this sequence is to assemble the data from the two reads in a right-justified manner on the **M%MD_BUS_H**. When the assembled data is driven, **M%IBOX_DATA_L** and/or **M%EBOX_DATA_H** are asserted to indicate the destination of the data.

The RTY_DMISS_LATCH always contains a physical address because it stores retried reads from the S6 pipe. The implication of this fact on unaligned reads is that an unaligned sequence is never initiated from the RTY_DMISS_LATCH because the RTY_DMISS_LATCH address is physical. If an unaligned reference crosses a page boundary, the physical address of the second reference is not guaranteed to be a quadword incremented version of the first reference since the first and second references are associated with different address translations.

12.3.17.2 Unaligned Writes

Like unaligned reads, unaligned writes are processed by breaking the reference into two aligned quadword references such that the VAP_LATCH always generates and stores the upper portion. When this EM_LATCH command successfully completes in S5, the VAP_LATCH generates the upper portion of the unaligned write reference in the same manner as an unaligned read. The data driven on **M_QUE%S5_DATA_H**<31:0> from the EM_LATCH byte rotator during the first write is latched in the VAP_LATCH. Thus, when the VAP_LATCH write executes, the same data is again driven onto **M_QUE%S5_DATA_H**<31:0>. It is the different byte masks and addresses of the two aligned writes which cause the proper bytes to be written into the proper bytes of memory.

12.3.17.3 Byte Mask Generation for Unaligned Writes

The byte mask generator must understand whether a given reference is the first or second reference of an unaligned reference pair in order to generate the appropriate byte mask. M_QUE%S5_QUAL_H<3> is used to determine this.

The following table illustrates examples of the behavior of the byte mask generator for aligned and unaligned writes:

Table 12–17: Byte Mask Logic for Aligned and Unaligned References

ref	addr<2:0>	BM (DL=byte)	BM (DL=word)	BM (DL=long)	BM (DL=quad)
1st	000	00000001	00000011	00001111	00001111
2nd	000	—	—	—	—
1st	001	00000010	00000110	00011110	00011110
2nd	001	—	—	—	—
1st	010	00000100	00001100	00111100	00111100
2nd	010	—	—	—	—
1st	011	00001000	00011000	01111000	01111000
2nd	011	—	—	—	—
1st	100	00010000	00110000	11110000	11110000
2nd	100	—	—	—	—
1st	101	00100000	01100000	11100000	11100000
2nd	101	—	—	00000001	00000001
1st	110	01000000	11000000	11000000	11000000
2nd	110	—	—	00000011	00000011
1st	111	10000000	10000000	10000000	10000000
2nd	111	—	00000001	00000111	00000111

Since the VAP_LATCH always increments the virtual address by eight, the lower three bits of the VAP_LATCH address will always be the same as the original address. However, the lower three bits of the address sent to the Cbox (M%C_S6_PA<2:0>) are always zeroed on the second half of an unaligned reference in order that the address that is sent off chip is consistent with the corresponding byte mask value.

12.3.17.4 Unaligned Destination Specifier Writes

When an unaligned DEST_ADDR or unaligned DREAD_MODIFY command is latched in the SPEC_QUEUE, the unaligned detection logic flags the unaligned condition and thus, the reference is split into two aligned references by the mechanism described previously. As each one of the pair of commands executes, one entry will be added to the PA_QUEUE.

When the corresponding data arrives in the EM_LATCH via the STORE command, the data is rotated based on the lower two address bits output from the PA_QUEUE. The rotated data is then matched up with the reference driven from the PA_QUEUE to form a newly assembled WRITE command. Since the reference driven from the PA_QUEUE indicates that M_QUE%S5_QUAL_H<4>=1 (i.e. this reference is the first part of an unaligned pair), the VAP_LATCH latches and validates a copy of the STORE command with the rotated STORE data. When this newly assembled WRITE command successfully completes in S5, the bottom entry of the PA_QUEUE is retired. When the VAP_LATCH subsequently executes the second STORE reference, the second entry in the PA_QUEUE is matched with it and retired. In effect, the STORE data is split into two STORE commands so that each STORE is merged with each PA_QUEUE entry to form two WRITE commands.

12.3.17.5 Implication of Ebox unaligned references on M%EM_LAT_FULL_H

The EM_LATCH is invalidated whenever the EM_LATCH reference successfully completes in S5. However, if the EM_LATCH reference was unaligned, the second half of the reference still awaits processing in the VAP_LATCH even though the EM_LATCH has been invalidated. Clearing the EM_LATCH while the second half of an unaligned Ebox reference is still pending could release the EM_STALL condition causing the Ebox microcode to advance even though the Mbox has not completed processing of the second part of the previous unaligned reference.

This scenario is undesireable since the Ebox microcode makes synchronization assumptions based on references being retired from the EM_LATCH. To preserve these assumptions, the Mbox will assert M%EM_LAT_FULL_H until both halves of the unaligned reference have been retired even though the EM_LATCH will have been invalidated earlier. Note that this applies to both unaligned reads and unaligned writes.

12.3.18 ABORTING REFERENCES

The Mbox abort operation is used to cancel the current S5 operation. When an abort is executed, the S5 state, which would normally be updated due to execution of the current S5 reference, is not updated. The aborted S5 reference is not propagated into S6. Instead, a NOP is introduced into the S6 pipe. In effect, an aborted S5 reference is equivalent to a NOP command being executed in S5.

Note that the abort operation should be viewed as only cancelling the current execution of a reference. In most cases, aborting an operation does not invalidate the existence of the corresponding reference, which will still be stored in one of the reference sources and retried at a later point.

The abort operation is executed when M_S5C_ABT%ABORT_L is asserted. The following changes to Mbox state are inhibited during the cycle in which M_S5C_ABT%ABORT_L is asserted:

- The reference source which drove the aborted command into S5 does not invalidate the corresponding command. Thus, the reference still exists to be retried during a subsequent cycle.

> **NOTE**
>
> There are two exceptions to this rule. The CBOX_LATCH is always invalidated after it drives a command into S5. The EM_LATCH will be invalidated if the Ebox has explicitly requested it to be (via the **E%EM_ABORT_L** signal).

- Loading the PA_QUEUE with a DEST_ADDR or DREAD_MODIFY command is inhibited. Emptying the PA_QUEUE when a STORE command is driven in S5 is inhibited.
- If the unaligned detection logic detected an unaligned reference during the aborted cycle, the VAP_LATCH is not validated to contain the second portion of the unaligned sequence.

12.3.18.1 Conditions for Aborting References

In general, references are aborted for five reasons:

- The reference is aborted to prevent a reference order restriction from occurring (see Section 12.3.18.1.1).
- The reference is aborted because insufficient hardware resources are available to complete processing of the current command.
- The reference is aborted because a memory management operation must be performed prior to execution of the current reference.
- The reference is aborted in order to avoid a deadlock condition related to unaligned references.
- The reference is aborted due to an external flush condition.

The following describes the specific conditions which can invoke an abort operation for each of the five categories listed above.

12.3.18.1.1 Aborting to Maintain Reference Order Restrictions

- Aborting D-stream hits under D-stream misses: Consider the case where two D-stream reads are executed in back-to-back cycles. In this case, the second D-stream read will be aborted in S5 if the first D-stream read misses in the Pcache in S6. This prevents the possibility of propagating the second read into S6 and having it hit and return data before the first read returns data.

  Note that this condition applies to all D-stream "read-like" references (i.e. references which return data to the Ebox). Specifically, this condition applies to DREAD, DREAD_MODIFY, DREAD_LOCK, IPR_RD, and PROBE commands.
- Aborting I-stream hits under I-stream misses: The Mbox initiates an IREAD sequence by issuing consecutive IREAD commands via the I-stream "fill forward" mode (See Section 12.3.5.2.1). If the first IREAD in this sequence misses in the Pcache in S6 while the second IREAD is executing in S5, the second IREAD is aborted. This is done to handle I-stream reads in an analogous fashion to D-stream reads.
- Aborting to preserve order of Ibox reads relative to Ebox writes: As explained previously, the PA_QUEUE is the structure used to store pending destination specifier addresses until the Ebox can supply the corresponding data to complete the write reference. Once the Ebox supplies the data, the write executes and the corresponding entry in the PA_QUEUE is invalidated.

The comparator function built into the PA_QUEUE is used to detect address matches on bits<8:3> between Ibox D-stream read references and any of the valid PA_QUEUE entries.

Consider the example shown in Figure 12–14.

In this example, the Ibox would decode the destination specifier of the first MOVL instruction which causes a DEST_ADDR command to be sent to the PA_QUEUE. Subsequently, the Ibox would decode the first specifier of the second MOVL, causing a read to be issued to the Mbox. When this read is started in the S5 pipe, a PA_QUEUE comparator will detect an address conflict between the read and the pending destination address. As a result, the read is aborted and is not successfully executed until the write completes. Thus, all reads originating from the SPEC_QUEUE are aborted if the PA_QUEUE detects an address conflict.

Note that the PA_QUEUE must always detect physical address conflicts. Detecting virtual address conflicts is not sufficient since two or more different virtual pages could be mapped to the same physical page causing two or more different virtual addresses to conflict on the same physical longword.

Also note that the PA_QUEUE is capable of detecting false conflicts because only address bits <8:3> are compared rather than the entire address. Performance data indicates that the number of false conflicts using addr<8:3> is sufficiently low to have an insignificant performance degradation. Bits <8:3> are used since they are untranslated address bits and, therefore, are immediately available for use without waiting for the address to be translated. The lower three bits are not used because the PA_QUEUE must detect conflicts at quadword resolution. The following diagram illustrates why quadword resolution must be used:

Figure 12–42: PA_QUEUE conflict detection

```
<------------------------------------ memory aligned quadword ----------------------------------->

|                                                                                               |
|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
|                      <---PA_QUEUE entry addresses this longword-->                            |

                                   <----+----->                 PA_QUEUE addr<2:0>: 010
A DREAD is issued which                 |                       DREAD addr<2:0>:    101
addresses this byte --------------------+
```

The diagram above illustrates eight bytes of memory within a memory aligned quadword. In this example, the PA_QUEUE contains a destination address which references a longword. While this reference is not longword aligned, it is handled as an aligned reference because the reference does not cross an aligned quadword boundary. Consider the byte DREAD shown above which is issued by the SPEC_QUEUE and is executed in S5 in the presence of the PA_QUEUE entry. While a PA_QUEUE address conflict clearly exists on the fifth byte within this quadword, the lower three bits of the PA_QUEUE address do not match the lower three bits of the DREAD address. Thus, the the lower three bits cannot be used for the purposes of PA_QUEUE conflict detection.

DREAD_MODIFY references with DL=quadword pose a special problem for the PA_QUEUE conflict logic. Quadword memory operands are requested by the Ibox by issuing a D-stream reference with DL=quadword followed by another D-stream reference with DL=longword. The first reference causes the lower half of the quadword operand to be returned on M%MD_BUS_H<31:0> (i.e. all quadword DREADs only return a longword of data). The second reference addresses the upper half of the quadword causing the upper half of the operand to be returned on M%MD_BUS_H<31:0>. If the quadword operand is aligned, both the quadword and the longword references have the same quadword address. Thus, when the DREAD_MODIFY longword reference is executed in S5, a PA_QUEUE address conflict could be detected against the DREAD_MODIFY quadword reference previously loaded. If this were to happen, a deadlock state would exist within the NVAX chip because the corresponding STORE data for the quadword operand cannot be generated to clear the PA_QUEUE until the Ebox receives the entire requested quadword operand, which cannot happen as long as a PA_QUEUE address conflict is detected. A similar deadlock situation could result from an unaligned DREAD_MODIFY quadword operand.

To avoid this deadlock problem, the PA_QUEUE control logic stores a state bit for each entry to indicate whether the DL is quadword. If the last entry loaded contains a quadword, the PA_QUEUE address conflict logic associated with that PA_QUEUE entry is inhibited. This avoids deadlock by preventing the PA_QUEUE conflict logic from detecting a conflict between the first half and the second half of the same DREAD_MODIFY quadword specifier.

- I/O space reads prefetched by the Ibox which are destined for the Ebox must be inhibited until the Ebox is stalling on that particular I/O space read: Since certain I/O devices can cause their state to change based on a read reference to that device, the possibility exists for I/O device state to be improperly modified based on Ibox prefetching of operands. We must guarantee that any state change only occurs within the context of Ebox execution of the corresponding instruction.

  Thus, I/O space reads are aborted in S5 until we can guarantee that the Ebox is executing the instruction corresponding to the I/O space read. This function is implemented by aborting any I/O space read originating from the SPEC_QUEUE which returns data to the Ebox when either of the following two conditions is true:

  1. **E%START_IBOX_IO_RD_H** is deasserted. **E%START_IBOX_IO_RD_H** is an Ebox signal that informs the Mbox that the S3 Ebox pipe is currently in MD_STALL waiting for an operand to be returned. Thus, the deassertion of this signal indicates that the Ebox cannot currently be stalling on the I/O space operand.
  2. A NOP command does not currently exist in the S6 pipe. This condition is necessary to account for a timing boundary condition which can exist between the Mbox and Ebox. It is possible for the Ebox to be MD_STALLing on an S6 reference corresponding to a previous instruction when the I/O read is in S5. In this case, **E%START_IBOX_IO_RD_H** could be asserted in reference to the previous MD data which may exist in the S6 pipe while the I/O space reference exists in the S5 pipe. To avoid this potential problem, the I/O space reference is aborted until a NOP is detected in S6 which indicates that this boundary condition cannot exist.

  Note that it is necessary to stipulate that this abort condition only affect Ibox I/O space DREAD references which directly return data to the Ebox. This is because it is conceivable that a deferred mode destination specifier could cause the DREAD of the address of the operand to map to I/O space. In this situation, the Ebox will never MD_STALL on this reference since it corresponds to a destination specifier. Thus, the pipeline could hang if the Mbox unconditionally aborted all Ibox I/O space DREADSs. By conditioning **M_QUE%S5_DEST_H** into this abort equation, this deadlock condition is avoided by only applying this abort condition to DREADs which return data to the Ebox

- Aborting reads to the same Pcache index as a pending read/fill operation: As stated in Section 12.2.13, allowing two Pcache fill sequences to simulataneously operate on the same Pcache block creates the possibility of corrupting this Pcache block. To prevent this, address bits <8:5> of the DMISS_LATCH are compared against **M_QUE%S5_PA_H**<8:5> when S5 contains an IREAD and the DMISS_LATCH is validated. If there is a match, the S5 IREAD is aborted in order that a potential I-stream fill sequence does not pollute the Pcache block associated with the D-stream fill already in progress.

Note that address bits<8:5> are used to detect a Pcache index address conflict even though bits<11:5> represents the entire Pcache index. The upper three bits of the Pcache index are not used because these can be translated address bits which are not available in time for the address comparator circuit. By only using bits <8:5>, some false address conflicts may occur. A false address conflict will needlessly delay processing of a read or write reference, however, the NVAX performance model has shown that this has a negligible impact on overall performance.

Even if a true Pcache index conflict is detected, it is possible that there is no block conflict because the 2-way set associative Pcache contains two blocks per index. In order to reduce hardware complexity however, a block conflict is assumed to have occurred whenever an index conflict is detected even though the references may address different blocks within the index.

By the same rationale, the same address bits of a valid IMISS_LATCH are compared against M_QUE%S5_PA_H<8:5> when S5 contains a D-stream read. If a match is found, the S5 read is aborted in order to let the I-stream fill proceed without possible corruption.

- Aborting writes or STORESs to the same Pcache index as a pending read/fill operation: As stated in Section 12.3.18.1.1, writes should be inhibited from executing if they map to the same Pcache block as a Pcache fill already in progress. Otherwise, the memory write data could miss in the Pcache block during a fill sequence before the Cbox supplied the fill data. When this subblock is filled by the Cbox, this Pcache subblock would be validated with old data. Therefore, the write data which was processed by the Mbox would not be reflected in the Pcache.

  Avoiding this situation is accomplished by the comparators built into the DMISS_LATCH and IMISS_LATCH. If either of these latches are valid, and bits <8:5> of the fill address equals M_QUE%S5_PA_H<8:5> of an S5 write or S5 STORE, then the S5 write is aborted. Note that since the entire write address is not compared, we may abort writes when there was not a true address conflict. This is done however, for circuit speed reasons and does affect the overall CPU performance appreciably.

12.3.18.1.2 Aborting due to lack of hardware resources

- Aborting a "read-like" reference when the RTY_DMISS_LATCH is full: Consider the situation where a D-stream fill is executing and the RTY_DMISS_LATCH stores the next read to be executed. If a third read is started in S5, it is automatically aborted. If the third read were not aborted two incorrect scenarios would result. The third read could miss in S6 with no where to put it, since both the DMISS_LATCH and the RTY_DMISS_LATCH are full. If the third read hit, its data would be returned before the data of the second read, which is equivalent to an illegal "hit under miss" scenario.

  For the purposes of the above discussion, a "read-like" reference is defined as any reference which returns data to the Ebox. Thus, a read-like reference is a DREAD, DREAD_MODIFY, DREAD_LOCK, IPR_RD, or PROBE command.

- Aborting DEST_ADDR or DREAD_MODIFY due to insufficient room in PA_QUEUE: If a destination specifier reference is executing in S5, but there are insufficient PA_QUEUE entries to store the reference, the Mbox has no choice but to abort the S5 reference and retry it later when more PA_QUEUE entries free up. If the S5 reference is unaligned, the abort logic tests for two empty slots in the PA_QUEUE since two will be required for the unaligned reference. If the S5 reference is aligned, only one slot need be available.

- Aborting an S5 write, STORE or Cbox IPR_WR due to Cbox back-pressure: All S6 Cbox writes are automatically transferred to a write buffer in the Cbox. The Cbox uses this write buffer to store the writes until they can be written into the Cbox, Bcache or main memory. If this write buffer becomes sufficiently full so that we cannot guarantee that the S5 write or STORE can be loaded into the write buffer when it propagates to S6, the S5 command is aborted. The Cbox asserts write buffer back-pressure to the Mbox by asserting **C%WR_BUF_BACK_PRES_H**.

12.3.18.1.3 Aborting due to memory management operation

When a tb_miss or cross-page condition is detected, a memory management operation must be processed before the S5 reference can be allowed to complete. Thus, detection of a tb_miss or cross-page condition causes the S5 command to be aborted until the memory management operation finishes. This also prevents the possibility of having to handle a second memory management sequence before the first memory management sequence completes.

The two specific abort conditions are:

- Aborting an S5 reference due to TB_MISS condition: If the virtual address of the S5 reference is not found in the TB, the corresponding physical address cannot be immediately derived. Therefore, the reference is aborted until the translation can be cached in the TB (See Section 12.5.1.5.2 for information on memory management).
- Aborting an S5 reference due to CROSS_PAGE condition: If an unaligned S5 reference references two pages, a CROSS_PAGE condition has been detected. In this situation, access checks of both pages must be made before the reference is allowed to complete. Therefore, the reference is aborted and retried after the CROSS_PAGE check has tested the upper page (See Section 12.5.1.5.4).

In either situation described above, all but two reference types from the Ibox or Ebox references will be continually aborted until the memory management sequence completes. The two exceptions are the STOP_SPEC_Q and STORE commands. Since these references are guaranteed not to require any memory management function, these references are allowed to proceed. Note that while a STOP_SPEC_Q command is never aborted, it is transformed into a NOP command as it enters the S6 pipe. This is allowable since no S6 function is performed by this command and it offers an extra S6 data bypass opportunity.

12.3.18.1.4 Aborting due to an external flush condition

This abort condition will be explained in the discussion of flushes.

12.3.19 MBOX PIPELINE DEADLOCK AVOIDANCE SCENARIOS

Two special considerations have been designed into the Mbox in order to avoid two possible pipeline deadlock conditions.

12.3.19.1 Unaligned Reference Deadlock Condition

Consider the situation where the second part of an unaligned D-stream read is driven into S5 from the VAP_LATCH. If this read conflicts with the quadword address of a valid PA_QUEUE entry, this read will be aborted based on PA_QUEUE address conflict detection.

If the VAP_LATCH is not cleared, a pipeline deadlock situation has occurred because the VAP_LATCH command will always execute before an EM_LATCH command. However, a STORE command originating from the EM_LATCH is the only way the PA_QUEUE conflict can be eliminated. Therefore, in addition to aborting the VAP_LATCH reference during a PA_QUEUE conflict, the VAP_LATCH must be invalidated in order that the arbitration logic can select the EM_LATCH STORE command to clear the PA_QUEUE conflict condition.

Clearing the VAP_LATCH due to PA_QUEUE conflict detection has several implications. It means that the unaligned sequence must be restarted from the beginning in order to re-generate the VAP_LATCH reference. This is why the corresponding SPEC_QUEUE entry is not invalidated until the entire unaligned sequence successfully completes in S5. A side effect of this is that the first read of the unaligned sequence will be re-executed causing two read references to the same data. This, however, is harmless if the read is to memory. This may not be harmless if the read is to I/O space, however, unaligned I/O space reads are defined to yield UNPREDICTABLE results.

Another implication of avoiding this pipeline deadlock is that the bottom entry of the PA_QUEUE must be invalidated if the VAP_LATCH command was a DREAD_MODIFY command. If it was a DREAD_MODIFY, the first reference of the unaligned pair had already introduced an entry into the PA_QUEUE. Since the first reference will be re-executed, the corresponding PA_QUEUE entry is invalidated to avoid replicating the same PA_QUEUE entry twice.

12.3.19.2 READ_LOCK/WRITE_UNLOCK Deadlock Condition

Once a READ_LOCK command has been passed to the Cbox, the Cbox will not process any subsequent D-stream read references until the corresponding WRITE_UNLOCK command has been executed. This behavior introduces a deadlock consideration.

Consider the situation where a DREAD_LOCK has been sent to the Cbox. Before the EM_LATCH is loaded with the corresponding WRITE_UNLOCK, the Mbox starts processing an IREAD reference which misses in the TB. The resulting memory management sequence will issue a D-stream PTE read which the Cbox will not process until it has received the WRITE_UNLOCK command. However, the Mbox will never send the WRITE_UNLOCK (or any other Ebox or Ibox reference) until the memory management sequence completes, which can not occur until the PTE DREAD completes.

This deadlock condition is avoided by the arbitration logic by disabling IREF_LATCH selection once a DREAD_LOCK command has successfully been retired from the S5 pipe. Thus, no IREAD TB_MISS can occur between the READ_LOCK and WRITE_UNLOCK, thus avoiding the deadlock situation.

The arbitration logic will re-enable IREF_LATCH selection on either of the following two conditions:

1. A WRITE_UNLOCK reference has been retired from the S5 pipe. This will cause the Cbox to resume D-stream read processing, thus eliminating the deadlock condition.

2. E%FLUSH_MBOX_H is asserted by the Ebox due to a hard error. This condition should occur much more infrequently than the above condition because a WRITE_UNLOCK must normally be issued after a READ_LOCK. However, if an error occurred sometime between the READ_LOCK and WRITE_UNLOCK, a hard error microtrap will result preventing a WRITE_UNLOCK from being issued. The microtrap will generate E%FLUSH_MBOX_H which re-enables IREF_LATCH selection because no WRITE_UNLOCK will follow.

   Note that the Cbox state, which prevents subsequent D-stream reads from being processed before the WRITE_UNLOCK, will be cleared by an IPR_WRITE during the error handler.

Note that the analogous deadlock condition involving a SPEC_QUEUE reference cannot occur because Ibox processing will have been halted prior to the READ_LOCK/WRITE_UNLOCK sequence. The analogous deadlock condition involving an EM_LATCH reference will not occur because Ebox microcode will never issue a D-stream read in the middle of a READ_LOCK/WRITE_UNLOCK sequence.

12.3.20 THE SPEC_Q_SYNC_CTR

The PA_QUEUE address comparator function can maintain the relative order of specifier reads and destination specifier writes because both the reads and the writes originate from the same Ibox pipeline stage and are loaded into the same reference queue. However, when the Ebox issues reads or writes independently of the Ibox destination specifier decodes, the PA_QUEUE cannot be used since there is no implied ordering between the Ibox reads and Ebox reads or writes from two different pipeline stages. In this case, an 8-state counter, called the SPEC_Q_SYNC_CTR, is used to prevent Ibox memory operand prefetching when the Ebox can be writing to memory.

When the Ibox decodes an instruction that can cause explicit Ebox writes which are independent of the Ibox destination specifier decodes (e.g. MOVC), the Ibox loads the SPEC_QUEUE with a STOP_SPEC_Q command after all specifer references for the same instruction have been loaded. Execution of STOP_SPEC_Q in S5 causes the SPEC_Q_SYNC_CTR to be decremented. The nominal state of this counter is one. Whenever, the value of SPEC_Q_SYNC_CTR is zero, the arbitration logic will not select a SPEC_QUEUE reference as the source for the S5 pipe for the next cycle. The effect achieved is to stop all Ibox specifier references from occurring after the STOP_SPEC_Q command has executed. When the Ebox completes all explicit writes for the instruction which caused the Ibox to issue the STOP_SPEC_Q command, the Ebox asserts the E%RESTART_SPEC_QUEUE_H signal. Each assertion of E%RESTART_SPEC_QUEUE_H causes the SPEC_Q_SYNC_CTR to be incremented. Subsequent specifier reference processing resumes when the value of SPEC_Q_SYNC_CTR is positive. Thus, the SPEC_Q_SYNC_CTR acts as a synchronization device to stop processing of specifier references whenever the Ebox may be independently modifying memory state.

Note that a value of zero in the SPEC_Q_SYNC_CTR only prevents the arbitration logic from selecting the SPEC_QUEUE as the S5 reference source. It does not prevent the Ibox from loading additional references into empty SPEC_QUEUE entries.

The SPEC_Q_SYNC_CTR is an 8-state unsigned counter which can store values from 0 to 7. A counter function must be used for this synchronization function because pipeline behavior can cause the Ebox to assert E%RESTART_SPEC_QUEUE_H multiple times before the Mbox ever processes any STOP_SPEC_Q commands. For example, if the Mbox is executing a TB_MISS flow while the Ebox is retiring multiple instructions associated with this synchronization scheme, multiple assertions of E%RESTART_SPEC_QUEUE_H will result even though no STOP_SPEC_Q commands have been processed yet due to the on-going memory management sequence.

Thus, the SPEC_Q_SYNC_CTR buffers up the **E%RESTART_SPEC_QUEUE_H** assertions until the corresponding STOP_SPEC_Q commands are processed from the SPEC_QUEUE. Note that there is no need for the SPEC_Q_SYNC_CTR to buffer up multiple instances of STOP_SPEC_Q because the SPEC_QUEUE intrinsically buffers these instances.

The 8-state SPEC_Q_SYNC_CTR can buffer up to six **E%RESTART_SPEC_QUEUE_H** assertions (SPEC_Q_SYNC_CTR values 2 through 7). Six buffer states are sufficient to buffer all pending instructions which could result in the Ebox assertion of **E%RESTART_SPEC_QUEUE_H** because at most six of these instructions can be issued to the Ebox before the Ibox is back-pressured from decoding the next instruction of this type. Six buffered states are derived from the fact that the Ibox must fill its four-stage pipeline in addition to the 2-entry SPEC_QUEUE before it is back-pressured by the SPEC_QUEUE from issuing any further instructions which the Ebox could assert **E%RESTART_SPEC_QUEUE_H** in response to.

12.3.21 FLUSHING REFERENCES FROM THE MBOX PIPE

Flushing the Mbox pipeline refers to altering the state of the Mbox in a controlled way so that certain pending and currently executing references are eliminated from the Mbox. There are two distinct mechanisms that cause different types of references to be flushed. One type of flush originates from the Ibox and the other type from the Ebox.

12.3.21.1 Ibox Flushes

If the Ibox VIC is in the process of being filled by a previously requested IREAD, and the Ibox has determined, or has been forced, to start decoding instructions at a new point in the I-stream requiring another VIC fill, the Ibox asserts the signal, **I%FLUSH_IREF_LAT_H**, to the Mbox. From the Ibox point of view, assertion of **I%FLUSH_IREF_LAT_H** indicates that the current VIC fill operation will be immediately cancelled. This allows the Ibox to invoke a new VIC fill operation via a new IREAD, without having to wait for the current VIC fill operation to complete.

From the Mbox point of view, assertion of **I%FLUSH_IREF_LAT_H** aborts all pending and currently executing I-stream activity by performing the following actions:

1. The IREF_LATCH is invalidated. Any IREAD sent to the Mbox during the cycle **I%FLUSH_IREF_LAT_H** is asserted is not validated.
2. If the current S5 reference is an IREAD or an I_CF, it is aborted.
3. The IMISS_LATCH is invalidated and all state indicating an outstanding I-stream fill is cleared. If the IMISS_LATCH is being loaded during the cycle that **I%FLUSH_IREF_LAT_H** is asserted, the IMISS_LATCH is not validated.
4. The signal, **M%ABORT_CBOX_IRD_H**, is asserted to the CBOX to indicate that the Mbox does not want any more I_CF references which may have been pending in the Cbox.

If **I%FLUSH_IREF_LAT_H** is asserted during a cycle with an outstanding istream read or fill, the Mbox logic guarantees that the **M%VIC_DATA_L** signal will not be asserted in response to the IREAD during any subsequent cycles. However, **M%VIC_DATA_L** may be asserted during the same cycle that **I%FLUSH_IREF_LAT_H** is asserted. It is the responsibility of the Ibox to ignore the corresponding data in this case.

12.3.21.2 Ebox Flushes

12.3.21.2.1 Flushing due to E%EM_ABORT_L

Due to the construction of the microcode, it is possible for the Ebox to issue a reference to the Mbox only to discover during the following cycle that the reference should not have been issued. In this case, the Ebox asserts **E%EM_ABORT_L** during the cycle following when the reference was issued. **E%EM_ABORT_L** causes the Mbox to unconditionally clear the EM_LATCH and to abort the S5 reference if that reference was driven from the EM_LATCH. The net effect is to flush out all Mbox state associated with this Ebox reference.

12.3.21.2.2 Flushing due to E%FLUSH_MBOX_H

When the Ebox determines that a branch misprediction took place, or that process context is to be changed, or that an exception or interrrupt has occured, the macropipeline must be flushed in order that no processor state changes as a result of subsequent pipeline operations. As part of this flush operation, all pending or currently executing references in the Mbox which correspond to flushed instructions are immediately and permanently aborted. The Ebox informs the Mbox of this situation by asserting **E%FLUSH_MBOX_H**.

The assertion of **E%FLUSH_MBOX_H** invokes the following Mbox actions:

1. The SPEC_QUEUE is invalidated. Any reference sent to the Mbox SPEC_QUEUE during the cycle in which **E%FLUSH_MBOX_H** is asserted is not validated.
2. The SPEC_Q_SYNC_CTR is unconditionally reset to the value of 0. The effect of this is to inhibit further SPEC_QUEUE reference processing by never selecting the SPEC_QUEUE as the S5 reference source (See Section 12.3.20). It does not inhibit the Ibox from loading references into the SPEC_QUEUE during subsequent cycles, however. This function is associated with the scheme for flushing the PA_QUEUE. See Section 12.3.21.2.3.
3. If the current S5 reference was driven from the SPEC_QUEUE, it is aborted.
4. If the EM_LATCH contains any type of read, IPR_RD, probe or MME_CHK, it is invalidated. Any reference sent to the EM_LATCH during the cycle that **E%FLUSH_MBOX_H** is asserted is not validated.
5. If the current S5 reference was driven from the EM_LATCH, and this reference is any type of read, IPR_RD, probe or MME_CHK it is aborted.
6. If the VAP_LATCH contains any type of read or DEST_ADDR, it is invalidated. If a read or DEST_ADDR is being loaded into the VAP_LATCH during the cycle that **E%FLUSH_MBOX_H** is asserted, the VAP_LATCH is not validated.
7. If the current S5 reference was driven from the VAP_LATCH, and this reference is any type of read or DEST_ADDR, it is aborted.
8. If the RTY_DMISS_LATCH contains any type of an Ibox or Ebox read, it is invalidated. If an Ibox or Ebox read is being loaded into the RTY_DMISS_LATCH during the cycle that **E%FLUSH_MBOX_H** is asserted, the RTY_DMISS_LATCH is not validated.
9. If the current S5 reference was driven from the RTY_DMISS_LATCH, and this reference is an Ibox or Ebox read, it is aborted.
10. If the DMISS_LATCH contains a currently outstanding Ibox or Ebox read, the DMISS_LATCH state is modified to indicate that the data should not be sent to the Ibox or Ebox when the data becomes available.
11. MMESTS<31:29> are cleared. This unlocks the MMESTS reg.

The effect of items 1 through 10 above can be summarized as follows. All Ibox and Ebox D-stream reads, which have not yet propagated into S6, are blown away. Note that Mbox D-stream reads (PTE references) are not affected by **E%FLUSH_MBOX_H**. Any outstanding D-stream fill sequence corresponding to an Ibox or Ebox D-stream read is allowed to complete in order that the D-stream data is filled in the Pcache. However, the requested data will not be returned to the Ibox and/or Ebox. Any WRITE or STORE reference which existed in one of the Mbox reference sources PRIOR to the **E%FLUSH_MBOX_H** assertion is allowed to complete in the presence of the **E%FLUSH_MBOX_H** assertion. This is necessary because any write data existing in the Mbox prior to the **E%FLUSH_MBOX_H** assertion represents a memory modification corresponding to an action before the Ebox decided to flush.

If **E%FLUSH_MBOX_H** is asserted during a cycle with an outstanding D-stream read or D-stream fill, the Mbox logic guarantees that the **M%IBOX_DATA_L** and **M%EBOX_DATA_H** signals will not be asserted in response to the D-stream read/fill during any subsequent cycles. However, **M%IBOX_DATA_L** or **M%EBOX_DATA_H** may be asserted during the same cycle that **E%FLUSH_MBOX_H** is asserted. It is the responsibility of the Ibox and Ebox to ignore the corresponding data in this case.

Note that **I%FLUSH_IREF_LAT_H** causes an outstanding I-stream fill sequence to be completely stopped, but **E%FLUSH_MBOX_H** allows an outstanding D-stream fill sequence to continue without returning data to the Ibox and/or Ebox. These two cases are handled differently based on performance model data which indicates that it is beneficial to future references to complete the D-stream fill, but allowing the I-stream fill to complete only hinders the immediate need of accessing different I-stream data.

12.3.21.2.3 Ebox Flushing of the PA_QUEUE

The function of **E%FLUSH_MBOX_H** described above is to clear out reference state associated with instructions that had not yet been started by the Ebox. Note however, that **E%FLUSH_MBOX_H** does not flush the PA_QUEUE even though the PA_QUEUE may contain reference state that should be logically flushed by **E%FLUSH_MBOX_H**. This is because the PA_QUEUE may also contain reference state associated with the currently executing Ebox instruction. The PA_QUEUE entries associated with the currently executing Ebox instruction must be retired from the PA_QUEUE in the normal fashion before the remaining PA_QUEUE entries may be flushed.

Thus, flushing the PA_QUEUE is a two-step process described as follows:

1. As described in Section 12.3.21.2, **E%FLUSH_MBOX_H** inhibits the Mbox arbitration logic from selecting SPEC_QUEUE references for processing during subsequent cycles. This function guarantees that no more PA_QUEUE entries can be filled during subsequent cycles.
2. Once the Ebox has issued all STOREs corresponding to state modifications that must occur before the Mbox is completely flushed, the Ebox issues another reference which is qualified with the **E%FLUSH_PA_QUEUE_H** signal. Once this EM_LATCH reference executes in S5, the Mbox is guaranteed to have completed all subsequent STORE references. Thus, when this EM_LATCH reference executes, the remaining entries in the PA_QUEUE are flushed. Note that both halves of an unaligned STORE will complete before the "**E%FLUSH_PA_QUEUE**" reference is executed because the second half of the reference is stored in the VAP_LATCH, which has higher priority than the EM_LATCH.

The Ebox will assert **E%RESTART_SPEC_QUEUE_H** once the "**E%FLUSH_PA_QUEUE**" reference has been latched in the EM_LATCH. **E%RESTART_SPEC_QUEUE_H** re-enables Mbox processing of SPEC_QUEUE references during subsequent cycles.

MICROCODE RESTRICTION

E%FLUSH_MBOX_H
has been asserted, **E%FLUSH_PA_QUEUE_H** and **E%RESTART_SPEC_QUEUE_H** must be asserted before the Ibox or Ebox require further Mbox processing of Ibox or Ebox D-stream references. **E%FLUSH_PA_QUEUE_H** and **E%RESTART_SPEC_QUEUE_H** must be asserted during a cycle subsequent to the assertion of **E%FLUSH_MBOX_H**, and only when the microcode guarantees that all corresponding STORE commands have been retired by the EM_LATCH.

12.4 THE PCACHE

The Pcache is a two-way set associative, read allocate, no-write allocate, write through, physical address cache of I-stream and D-stream data. The Pcache has a one cycle access and a one cycle repetition rate for both reads and writes. It stores 8192 bytes (8K) of data and 256 tags corresponding to 256 hexaword blocks (1 hexaword = 32 bytes). Each tag is 20 bits wide corresponding to bits <31:12> of the physical address. There are four quadword subblocks per block with a valid bit associated with each subblock. The access size for both Pcache reads and writes is one quadword. Even byte parity is maintained for each byte of data (32 bits per block). One bit of even parity is maintained for every tag.

The logical orgainization of the Pcache is shown below:

Figure 12–43: Logical Pcache Organization

```
             <-------------- left bank ----------------> <-------------- right bank --------------->
         +---+----+-----+----+------+------+------+------+----+-----+----+------+------+------+------+
  0:     | A | TP | TAG | VB | D/DP | D/DP | D/DP | D/DP | TP | TAG | VB | D/DP | D/DP | D/DP | D/DP |
         +---+----+-----+----+------+------+------+------+----+-----+----+------+------+------+------+
  1:     | A | TP | TAG | VB | D/DP | D/DP | D/DP | D/DP | TP | TAG | VB | D/DP | D/DP | D/DP | D/DP |
         +---+----+-----+----+------+------+------+------+----+-----+----+------+------+------+------+
  2:     | A | TP | TAG | VB | D/DP | D/DP | D/DP | D/DP | TP | TAG | VB | D/DP | D/DP | D/DP | D/DP |
         +---+----+-----+----+------+------+------+------+----+-----+----+------+------+------+------+
  .
  .
  .
         +---+----+-----+----+------+------+------+------+----+-----+----+------+------+------+------+
127:     | A | TP | TAG | VB | D/DP | D/DP | D/DP | D/DP | TP | TAG | VB | D/DP | D/DP | D/DP | D/DP |
         +---+----+-----+----+------+------+------+------+----+-----+----+------+------+------+------+

    where: A =       Allocation bit.  Indicates whether the left or right bank was last allocated.
           TP =      1 bit of even tag parity.
           TAG =     20 bits of tag address.
           VB =      4 valid bits.  Each bit corresponds to 8 bytes of data.
           D/DP =    8 bytes of data with 8 bits of even byte parity (72 total bits).
```

The Pcache is logically organized into 128 direct mapped indexes, where each index consists of two blocks, and each block consists of: 20-bit tag, 1-bit tag parity, 4 valid bits, 256 bits of data, and 32 bits of data parity. In addition, each index also contains a one bit allocation pointer.

The breakdown of address bits for Pcache decoding is shown below:

Figure 12–44: Pcache Address Breakdown

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      tag address                          |    index address   |     |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                                                 <--+-->
                                                                                    |
                                                            subblock address -------+

 where:  tag address =    bits loaded into or compared with tag.
         index address =  addresses 1 of 128 indexes.
         subblk address = addresses 1 of 4 aligned quadwords within the hexaword data block.
```

12.4.1 PCCTL

The PCCTL controls the mode of operation of the Pcache. PCCTL is accessible by IPR_RD and IPR_WR operations. See Figure 12–31 for the definition of this register.

Note that Pcache operation is further qualified by the state of PCSTS<0> (See Section 12.6 for more information about PCSTS). If this bit is non-zero, Pcache operation is automatically forced to behave as if I_ENABLE=0 and D_ENABLE=0, regardless of the actual state of I_ENABLE and D_ENABLE. Effectively, this shuts down normal Pcache operation due to the presence of a previous Pcache parity error.

Note that Pcache invalidate operations are only disabled if both D_ENABLE=0 and I_ENABLE=0, or if PCSTS<0> is set.

Note that the ELEC_DISABLE bit of PCCTL is intended for debug use only. This bit electrically disables the Pcache to reduce power dissipation. This bit should only be set when the Pcache is functionally turned off by the deassertion of both I_ENABLE and D_ENABLE. UNPREDICTABLE operation will result when this bit is set when either I_ENABLE or D_ENABLE is also set. Any further discussion concerning Pcache function assumes that ELEC_DISABLE is inactive.

Also note that all Pcache IPR_RD and IPR_WR operations will function correctly regardless of the state of I_ENABLE or D_ENABLE or PCSTS<0>. However, Pcache array IPRs will not function if ELEC_DISABLE is set.

If either D_ENABLE or I_ENABLE are to be toggled to the on state, the Pcache array must be initialized prior to such action. See Section 12.8.2.1 for more information about Pcache initialization.

When the FORCE_HIT (Force Hit) bit is set and I-stream or D-stream operation is enabled, all enabled memory space read and write references are forced to hit in the Pcache regardless of the value of the stored tag. The BANK_SEL bit specifies which tag of the pair of tags addressed is forced to hit. Thus when FORCE_HIT=1, the Pcache becomes a 4K direct mapped cache with all reads and writes forced to hit in the Pcache. Toggling BANK_SEL causes the other half of the 8K Pcache to become accessible in this direct mapped mode. Note that BANK_SEL never affects bank selection during IPR reads and IPR writes to the Pcache tags or Pcache data parity bits; bank selection for these commands is always determined by the specified IPR address. Also note that the FORCE_HIT bit only affects memory space references. I/O space references still miss in the Pcache regardless of the state of the FORCE_HIT bit.

The FORCE_HIT feature is designed to facilitate testing the Pcache data array and to make diagnostic tests easily loadable within the Pcache by simple WRITE operations. When FORCE_HIT=0, the Pcache is configured as an 8K 2-way set associative cache, no reads or writes are forced to hit, and the BANK_SEL bit is a don't care.

The P_ENABLE (Parity Enable) bit allows the detection of Pcache tag and data parity errors to be enabled or disabled. If P_ENABLE=0, Pcache parity errors will not be detected. Thus when P_ENABLE=0, no Pcache error will be recorded in PCSTS or will be reported to the Ebox.

Note however, that when FORCE_HIT=1, Pcache tag parity is never checked regardless of the state of P_ENABLE.

12.4.2 Pcache Hit/Miss Determination

12.4.2.1 Hit/Miss Determination by Tag Comparison

When an IREAD, DREAD, DREAD_MODIFY, WRITE, WRITE_UNLOCK, or INVAL operation is executed, the Pcache must determine if the referenced data is present in its array. To do this, physical address bits<11:5> are input to the Pcache row decoders in order to determine which one of the 128 direct mapped indexes is being addressed. Subsequently, all 629 bits within the addressed index are accessed by the assertion of the corresponding word line. The two accessed tag values are simultaneously compared to physical address bits<31:12>. A Pcache hit condition occurs when all of the following conditions are simultaneously true:

- The contents of one of the two addressed tags matches the data on M%S6_PA_H<31:12>.
- The valid bit corresponding to both the matched tag and to the addressed subblock (specified by physical address bits<4:3>) is set.
- The stored tag parity corresponding to the matched tag is the same as the value calculated off of M%S6_PA_H<31:12>.

If an address match is detected on one of the tags and the valid bit which corresponds to both the matched tag and the addressed subblock (specified by physical address bits<4:3>) is set, then a Pcache hit condition has been detected on the corresponding Pcache tag. The absence of the Pcache hit condition causes a Pcache miss condition.

12.4.2.2 Conditions which force Pcache Miss

The Pcache miss condition is forced to override the tag determination of hit/miss described above when any one of the following conditions is satisfied:

- If PCSTS<0> is set, the Pcache miss condition is forced due to a previous Pcache parity error.
- If an IREAD or I_CF operation is accessing the Pcache and I_ENABLE=0, the Pcache miss condition is forced.
- If a D-stream read or D_CF operation is accessing the Pcache and D_ENABLE=0, the Pcache miss condition is forced.
- If a DREAD_LOCK operation is executing, the Pcache miss condition is forced. This guarantees that the read will propagate to the Cbox for synchronization purposes.
- If an I_CF operation is executing and the IMISS_LATCH state indicates that the reference cannot be cached, the Pcache miss condition is forced.

- If a D_CF operation is executing and the DMISS_LATCH state indicates that the reference cannot be cached, the Pcache miss condition is forced.

12.4.2.3 Conditions which force Pcache Hit

The Pcache hit condition is forced to override the tag determination of hit/miss described above when any one of the following conditions is satisfied. Note that unless explicitly stated to the contrary, the forced Pcache miss conditions above take precedence over the forced Pcache hit conditions described below.

- If a read reference is tagged as having a memory management fault or hard error associated with it (i.e. **M_QUE_MS2%S6_QUAL_H<0>** = 1 or **M_QUE_MS2%S6_QUAL_H<1>** = 1), a Pcache hit condition is forced. NOTE: This force hit condition takes precedence over any force miss condition described above.
- If the operation is a DREAD, DREAD_MODIFY, WRITE, or WRITE_UNLOCK, and D_ENABLE=1 and FORCE_HIT=1, the Pcache hit condition is forced on the tag corresponding to both the addressed Pcache index and the bank specified by the BANK_SEL bit EXCEPT when the address maps to I/O space. I/O references must never hit in the Pcache regardless of the state of FORCE_HIT.
- If the operation is an IREAD and I_ENABLE=1 and FORCE_HIT=1, the Pcache hit condition is forced on the tag corresponding to both the addressed Pcache index and the bank specified by the BANK_SEL bit.
- If the operation is a D_CF and D_ENABLE=1 and the DMISS_LATCH state indicates that the reference is cacheable, the Pcache hit condition is forced and the bank is specified by the allocation field of the DMISS_LATCH.
- If the operation is a I_CF and I_ENABLE=1 and the IMISS_LATCH state indicates that the reference is cacheable, the Pcache hit condition is forced and the bank is specified by the allocation field of the IMISS_LATCH.

12.4.3 Pcache Read Operation

A Pcache read operation is initiated by a DREAD, DREAD_MODIFY, or IREAD reference. A Pcache read begins by determining the Pcache hit or miss condition described above. If a Pcache hit is detected, the quadword of data corresponding to both the tag in which the hit occurred and to physical address bits<4:3> is driven out of the Pcache.

If a Pcache miss condition is asserted, all the data driven out of the Pcache is ignored except for the allocation bit. The allocation bit is stored in the DMISS_LATCH (in the case of a D-stream read) or in the IMISS_LATCH (for an IREAD). This bit will be used during a cache fill operation to select the appropriate block to be filled (See Section 12.4.6 for information about allocating and filling blocks).

12.4.4 Pcache Write Operation

A Pcache write operation is initiated by a STORE, WRITE or WRITE_UNLOCK reference. A Pcache write begins by determining the Pcache hit or miss condition described above. If a Pcache hit is detected, the data present on **B%S6_DATA_H**<63:0> is selectively written into the quadword corresponding to both the tag in which the hit occurred and to physical address bits<4:3>. The data is selectively written by using **M%S6_BYTE_MASK_H**<7:0> as a write enable for the eight respective bytes of data. The corresponding data parity is also written in the same manner for each corresponding byte which is written.

If a Pcache miss condition occurs, no Pcache write operation takes place. However, the write reference is forwarded to the Cbox for processing regardless of the hit/miss condition in the Pcache.

12.4.5 Pcache Replacement Algorithm

When a Pcache miss occurs during a read operation, it must be decided which one of two blocks will be allocated for the subsequent Pcache fill sequence. When the Pcache miss occurred because no validated tag field matched the read address, the state of the corresponding allocation bit indicates which bank (left or right) should be used for the resulting fill sequence. The value of each allocation bit changes according to the "not-last-used" algorithm. That is, the allocation bit always points to the bank within the index that was not last accessed.

When a read miss occurs because no validated tag field matched the read address, the value of the allocation bit is latched in the MISS_LATCH corresponding to the read miss. This latched value will be used as the bank select input during the subsequent fill sequence. As each fill operation takes place, the inverse of the allocation value stored in the MISS_LATCH is written into the allocation bit of the addressed Pcache index. During Pcache read or write operations, the value of the allocation bit is set to point to the opposite bank that was just referenced because this is now the new "not-last-used" bank.

The one exception to this algorithm occurs during an invalidate. When an invalidate clears the valid bits of a particular tag within an index, it only makes sense to set the allocation bit to point to the bank select used during the invalidate regardless of which bank was last allocated. By doing so, we guarantee that the next allocated block within the index will not displace any valid tag because the allocation bit points to the tag that was just invalidated.

12.4.6 Pcache Fill Operation

A Pcache fill operation is initiated by the I_CF (I-stream cache fill) or D_CF (D-stream cache fill) reference. A fill operation can be considered to be a specialized form of a write operation. A fill is functionally identical to a Pcache write operation except for the following differences:

- The bank within the addressed Pcache index is selected by the following algorithm. If a validated tag field within the addressed index matches the cache fill address, then the block corresponding to this tag is used for the fill operation. If this is not true, then the value of the corresponding allocation bit selects which block will be used for the fill.
- The first fill operation to a block causes all four valid bits of the selected bank to be written such that the valid bit of the corresponding fill data is set and the other three are cleared. All subsequent fills cause only the valid bit of the corresponding fill data to be set.

- Any fill operation causes the fill address bits<31:12> to be written into the tag field of the selected bank. Tag parity is also written in a analogous fashion.
- A fill operation causes the allocation bit to be written with the complement of the value latched by the corresponding MISS_LATCH during the initial read miss event.
- A fill operation forces every bit of the corresponding byte mask field to be set. Thus, all eight bytes of fill data are always written into the Pcache array on a fill operation.

12.4.7 Pcache Invalidate Operation

A Pcache invalidate operation is initiated by the INVAL reference. The invalidate operation is interpreted as a NOP by the Pcache if the address does not match either tag field in the addressed Pcache index. If a match is detected on either tag, an invalidate will occur on that tag. Note that this determination is made based only on a match of the tag field bits rather than on satisfying all criteria for the Pcache hit condition (Pcache hit factors in valid bits and verified tag parity into the equation).

When an invalidate is to occur, the four valid bits of the matched tag are written with zeros and the allocation bit is written with the value of the bank select used during the current invalidate operation.

Also note that an assertion of **C%CBOX_HARD_ERR_H** during a cache fill command causes the cache fill operation to be processed as if it were an INVAL operation.

12.4.8 Pcache IPR Access

For testability reasons it is important to verify that every Pcache storage bit can be read and written in both "0" and "1" states. The easiest way to do this is to provide a mechanism to directly read and write every bit in the Pcache array. The data field is already accessible through read and write commands. The tag field, tag parity, valid bits and data parity are directly accessible through IPR_RD and IPR_WR operations to the Pcache IPRs defined below:

Figure 12–45: IPR Address Space Mapping

```
Normal IPR Address

 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|      SBZ           | 0|                      SBZ                      |       IPR number      |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

Pcache TAG IPR Address

 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|      SBZ           | 1| 1| 0|            SBZ           | B| pcache index addr  |     SBZ      |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

   where: B =    0 ==> select the left bank of the specified index.
                 1 ==> select the right bank of the specified index.

Pcache Data Parity IPR Address

 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|      SBZ           | 1| 1| 1|            SBZ           | B| pcache index addr  | SBA |  SBZ   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

   where: B =    0 ==> select the left bank of the specified index.
                 1 ==> select the right bank of the specified index.
            SBA = subblock address selection
```

The format of a Pcache tag IPR is shown in Figure 12–32.

The tag parity bit is included in the Pcache tag IPR format to allow the user to write bad tag parity into the array in order to verify the tag parity logic. Further, the valid bits and allocation bit are also included so that the Pcache can be initialized to a known state.

The format of a Pcache Data Parity IPR is shown in Figure 12–33. This IPR allows the Pcache data parity to be directly read and written for testability purposes.

12.4.9 Pcache IPR Summary

The following table summaries all IPRs associated with the Pcache:

Table 12–18: Pcache IPRs

Register Name	IPR Address (in hex)
PCADR (quadword address of reference causing Pcache parity error)	F0
PCSTS (status of Pcache parity error)	F1
PCCTL (control state of Pcache operation)	F2
PCTAG	01800000..01801FE0
PCDAP	01C00000..01C01FF8

See Section 12.6 for a description of the PCADR and PCSTS registers. Note that with the exception of the Pcache tag IPRs, the addresses of the three other Pcache IPRs are driven into the Mbox shifted left two bits. This fact is not reflected in the above table.

12.4.10 Pcache States Resulting in UNPREDICTABLE operation

The capability of arbitrarily altering Pcache state through IPR write operations allows for the possibility of putting the Pcache into obscure states which cannot be achieved by "normal" operation. Two of these states will cause UNPREDICTABLE behavior:

1. Setting the ELEC_DISABLE bit in PCCTL will cause IPR read operations to the Pcache tag or Pcache data parity bits to return incorrect data. Setting the ELEC_DISABLE bit will cause IPR write operations to the Pcache tag or Pcache data parity bits to be disabled. Setting ELEC_DISABLE with either I_ENABLE or D_ENABLE set may cause Pcache read operations to return incorrect data. Setting ELEC_DISABLE with either I_ENABLE or D_ENABLE set will cause Pcache write, invalidate and cache fill functions to be disabled.
2. Through explicit Pcache tag IPR write operations, a user could write both blocks of a Pcache index with the same tag, tag parity and valid bit data. If this condition occurs with one or more sub-block valid bits set, the Pcache will return invalid data on references corresponding to the written tag (note that normal Pcache operation precludes this situation from ever occurring).

12.4.11 Pcache Redundancy Logic

Due to the extreme density of the Pcache array, the Pcache has a high susceptibility to manufacturing defects. As a result, redundancy logic was designed in order to provide a mechanism which would allow the Pcache to function correctly in the presence of a small number of manufacturing defects.

The redundancy logic consists of hardware which supports the operation of sixteen extra indicies which exist in addition to the 128 "regular" indicies. If a defect exists in an index which does not disturb the function of any column logic, the redundancy logic allows the bad index to be replaced by one of the 16 extra indicies. If an index is determined to be malfunctioning during chip test, a redundant index can be substituted for the bad index by blowing specific fuses on the chip through the use of a lazer. Blowing these fuses creates logic state transitions on redundancy control signals which disable the operation of a set of 4 "regular" indicies and will enable the operation of 4 redundant indicies in their place.

Four sets of four redundancy fuses exist. Each set controls 4 of the 16 redundant indicies. Each set can map its 4 redundant indicies into one of 8 different sets of 4 "regular" indices. The redundancy mapping is shown below:

Figure 12–46: Pcache Address Redundancy Mapping

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                                           | RS  | X|RED_ADDR| X|              |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

where:  RS represents the address bits corresponding to the four sets of four redundancy fuses.

        The two X's represent the address bits corresponding to the set of four indicies which
        get replaced.

        RED_ADDR represents the lazer-programmable address bits that specify which one of 8 sets
        of 4 "regular" indicies are to be replaced.
```

Each set of 4 redundancy fuses consists of three bits to specify the address mapping (specified by RED_ADDR above) and 1 bit to enable the redundant indicies to operate in place of the specified set of "regular" indicies. When one or more redundancy elements are blown, another fuse is also blown which will set the RED_ENABLE bit in PCCTL (see Figure 12–31). Thus, by reading the PCCTL IPR one can determine if one or more redundancy elements has been enabled.

12.5 MEMORY MANAGEMENT

The Mbox, the Ebox microcode, and the VMS memory management software implement VAX memory management. The Mbox performs the hardware memory management functions necessary to process most references in a quick efficient manner. The operating system software performs all other functions. For a description of the hardware end of VAX memory management, the reader is referred to the Memory Management chapter of the "VAX Architecture Standard" (DEC STD 032). For a complete description of the software end of VAX/VMS memory management, the reader is referred to the Memory Management chapters of "VAX/VMS Internals and Data Structures".

The Mbox is responsible for the following memory management functions:

- Performing virtual-to-physical address translations.
- Maintaining a cache of PTEs to perform the quick translations.
- Performing access mode checks on memory references.
- Performing TNV checks on memory references.
- Performing M=0 checks on memory references.
- Directly or indirectly invoking a software memory management exception handler due to ACV (Access Violation) or TNV (Translation not Valid) or M=0 faults.
- Detecting cross-page conditions and performing the corresponding access mode checks.

12.5.1 NVAX MEMORY STRUCTURE

12.5.1.1 Virtual Address Space

The NVAX virtual address space conforms with the description of the VAX virtual address space. The space contains four gigabytes (2**32) of memory divided into four regions as shown below:

Figure 12–47: Virtual Address Space Layout

```
          +--------------------------+
00000000  |                          |   length of P0 Region in
          |                          |   pages (P0LR)
          | P0      -----------------|
          | Region         |         |
3FFFFFFF  |                v         |   P0 Region growth direction
          +--------------------------+
40000000  |                ^         |   P1 Region growth direction
          |                |         |
          | P1      -----------------|
          | Region                   |   length of P1 Region in
7FFFFFFF  |                          |   pages (2**21-P1LR)
          +--------------------------+
80000000  |                          |   length of System Region
          |                          |   in pages (SLR)
          | System  -----------------|
          | Region         |         |
          |                |         |   System Region growth
          |                |         |   direction
          |                |         |
          |                v         |
          +--------------------------+
FFFFFE00  | Reserved                 |
FFFFFFFF  | Page                     |
          +--------------------------+
```

NOTE

NVAX CPU chips at revision 1 implement the original VAX memory management architecture in which any reference to a virtual address above BFFFFFFF (hex) falls into a reserved region and causes a length violation. NVAX CPU chips at revision 2 or later implement the extended S0 space addressing described above.

12.5.1.2 Physical Address Spaces

The NVAX hardware addresses a physical address space defined by another four gigabyte region. The first seven-eighths of it addresses physical memory. The top one-eighth of this space addresses I/O space. Thus, all I/O space addresses can be distinguished by physical address bits<31:29> = 111 (binary).

Figure 12–48: Physical Address Space of the NVAX Hardware

```
              +-------------------------+
00000000      |                         |
              |                         |
              +-                       -+
              |                         |
              |                         |
              +-                       -+
              |                         |
              |                         |
              +-                       -+
              |         Memory          |
              |          Space          | 3.5 Gigabytes
              +-                       -+
              |                         |
              |                         |
              +-                       -+
              |                         |
              |                         |
              +-                       -+
              |                         |
DFFFFFFF      |                         |
              +-------------------------+
E0000000      |           I/O           | 512 Megabytes
FFFFFFFF      |          Space          |
              +-------------------------+
```

12.5.1.2.1 Physical Address Space Mappings

The Mbox is designed to accommodate both a 30-bit and 32-bit physical address space as seen at the program level while maintaining one physical address space as seen by all NVAX hardware external to the Mbox (shown above). These two program level physical address spaces are mapped by Mbox hardware into the NVAX physical address space according to the value of the PAMODE register. See Figure 12–23 for a description of PAMODE.

The PAMODE register is accessible by the IPR_RD and IPR_WR commands. When PAMODE=0, the 30-bit physical address space seen at the program level is translated into the NVAX physical address space as follows:

Figure 12–49: 30-bit Physical Address Mapping

```
              Program Level 30-bit     mapped when       NVAX Physical
                 Address Space          PAMODE=0         Address Space

         +-------------------------+             +-------------------------+
00000000 |         Memory          |    00000000 |         Memory          |
1FFFFFFF |         Space           |    1FFFFFFF |         Space           |
         +- - - - - - - - - - - - -+             +-------------------------+
20000000 |          I/O            |             |                         |
3FFFFFFF |         Space           |             |                         |
         +-------------------------+             +-                       -+
         |                         |             |                         |
         |                         |             |                         |
         +-                       -+             +-                       -+
         |                         |             |                         |
         |                         |             |      Inaccessable       |
         +-                       -+             +-       Region          -+
         |                         |             |      When PAMODE=0      |
         |       Outside of        |             |                         |
         +-     30-bit Space      -+             +-                       -+
         |                         |             |                         |
         |                         |             |                         |
         +-                       -+             +-                       -+
         |                         |             |                         |
         |                         |    DFFFFFFF |                         |
         +-                       -+             +-------------------------+
         |                         |    E0000000 |          I/O            |
FFFFFFFF |                         |    FFFFFFFF |         Space           |
         +-------------------------+             +-------------------------+
```

Logically speaking, this mapping is accomplished by the Mbox by sign-extending physical address<29> into physical address<31:29>.

When PAMODE=1, the 32-bit physical address space seen at the program level is directly translated into the NVAX physical address space:

Figure 12–50: 32-bit Physical Address Mapping

```
           Program Level 32-bit     mapped when      NVAX Physical
             Address Space            PAMODE=1        Address Space

         +-------------------------+         +-------------------------+
00000000 |                         | 00000000 |                         |
1FFFFFFF |                         | 1FFFFFFF |                         |
         +-                       -+         +-                       -+
         |                         |         |                         |
         |                         |         |                         |
         +-                       -+         +-                       -+
         |                         |         |                         |
         |                         |         |                         |
         +-                       -+         +-                       -+
         |                         |         |                         |
         |         Memory          |         |         Memory          |
         +-        Space          -+         +-        Space          -+
         |                         |         |                         |
         |                         |         |                         |
         +-                       -+         +-                       -+
         |                         |         |                         |
         |                         |         |                         |
         +-                       -+         +-                       -+
         |                         |         |                         |
         |                         |         |                         |
         +- - - - - - - - - - - - -+         +-------------------------+
E0000000 |           I/O           | E0000000 |           I/O           |
FFFFFFFF |          Space          | FFFFFFFF |          Space          |
         +-------------------------+         +-------------------------+
```

12.5.1.3 ADDRESS TRANSLATION AND THE TB

For a complete description of VAX virtual address translation, the reader is referred to the Memory Management chapter of the "VAX Architecture Standard" (DEC STD 032). An overview of this process can be found in Section 2.6 of this specification.

The Mbox performs virtual-to-physical address translations in the S5 pipe when the following two conditions are satisfied:

1. The MAPEN bit is set (MAPEN enables virtual address translations).
2. **M_QUE%S5_QUAL_H<6>** indicates that the S5 reference is a virtual reference.

When both of these conditions are met, the address in **M_QUE%S5_VA_H<31:0>** is translated by the Mbox, and the resulting physical address is driven on **M_QUE%S5_PA_H<31:0>**. If both these conditions are not satisfied, the contents of **M_QUE%S5_VA_H<31:0>** is treated as a physical address and is directly transferred to **M_QUE%S5_PA_H<31:0>**.

The TB (translation buffer) is the mechanism by which the Mbox performs quick virtual-to-physical address translations. It is a 96-entry read allocate fully associative cache of PTEs (Page Table Entries).

The format of a page table entry and a TB entry are shown below.

Figure 12–51: PTE and TB format

```
Page Table Entry

      31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
     +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
     | V|   PROT    | M| S|                      Physical Page Frame Address                            |
     +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

 where:  V = valid bit
  PROT = authorized access modes
  M =  modify bit
  S =  reserved bit


TB Entry

        5    5       5    5       2 2   2  2    2  2                 
        4    3       2    1       9 8   5  4    3  2                0
      +-----+----+--------+---------+------+---+----+-----------------+
      | TBV | TP | TP_BAR |   TAG   | PROT | M | DP |       PFN       |
      +-----+----+--------+---------+------+---+----+-----------------+

 where:  TBV =        TB entry valid bit
               TP =         even tag parity bit
               TP_BAR =     complement of TP
               TAG =        virtual address<31:9>
               PROT =       authorized access modes
               M =          modify bit
               DP =         even parity for validated PTE field
               PFN =        physical page frame address
```

Note that the TB entry stores all but three bits of the PTE field. The TB entry does not store the S bit because it is not used, and the TB entry does not store the upper two bits of the PTE PFN because these bits correspond to a larger physical address space than NVAX uses. The tag field stores the virtual page frame address. The TBV bit indicates whether the corresponding entry is valid. If TBV is set, then PTE<31> is valid because the TB only caches PTEs whose valid bit is set.

The associativity of each TB entry is implemented by the use of comparators on the TBV and tag fields. When a virtual address is driven onto **M_QUE%S5_VA_H**<31:0> at the start of a cycle, each TB tag comparator, whose corresponding TBV bit is set, looks for a match between the **M_QUE%S5_VA_H** virtual page frame address and its corresponding tag. If no comparator finds a match, the TB_MISS condition has occurred indicating that no TB entry contains a translation for the specified address (see Section 12.5.1.5.2 for discussion of TB_MISSes).

If one of the entries detects a match (TB_HIT condition), the PFN, PROT, and M fields of the corresponding TB entry are read out of the TB. **M_QUE%S5_PA_H**<31:9> are driven with the contents of the accessed PFN. **M_QUE%S5_PA_H**<8:0> are the untranslated bits addressing a byte within a page; therefore, these bits are driven directly from **M_QUE%S5_VA_H**<8:0>.

The PROT, and M fields, which were driven out of the TB with the PFN, are used by the memory management exception detection logic to determine ACV and M=0 conditions (See Section 12.5.1.5.3).

TB entries are allocated using a NLU (Not-Last-Used) TB allocation pointer. The TB entry pointed to by the NLU allocation pointer is allocated and validated during a TB_TAG_FILL/TB_PTE_FILL sequence. The allocation pointer increments in round robin fashion around every TB entry when a TB lookup accesses the entry pointed to by the allocation pointer or when a TB_PTE_FILL operation is done. Because the allocation pointer is guaranteed not to point to the last entry referenced, this scheme implements a not-last-used allocation scheme.

TB entries can be invalidated in the following ways:

- An entry can be invalidated by being displaced from the TB by allocation of another PTE to the same TB entry.
- An entry can be invalidated by execution of the TBIS (TB Invalidate Single) command. If the specified TBIS virtual address matches a TB tag, the TBV bit corresponding to the matched tag is cleared. Clearing the TBV bit invalidates the TB entry (See Section 12.3.13).
- Entries can be invalidated by execution of the TBIP (TB Invalidate Process) command. TBIP causes the most significant bit of all the tag fields to be examined. If this bit is cleared, the corresponding TBV bit is cleared. The effect of this operation is to invalidate all PTEs corresponding to P0 or P1 space translations (See Section 12.3.14).
- All entries can be invalidated by the execution of the TBIA (TB Invalidate All) command. This command resets the TBV bit of every TB entry (See Section 12.3.15).

12.5.1.4 30-bit to 32-bit Physical Address Translations

When PAMODE=0, the NVAX system is configured such that only 30-bit physical addresses are processed at the program level. Since the Mbox and Cbox hardware is designed assuming a 32-bit hardware address space, the Mbox must appropriately translate all 30-bit physical addresses into 32-bit physical addresses based on the mapping scheme shown in Figure 12–49. This is done in two ways.

1. When the Mbox receives a physical address from one of its reference sources, the mapping is implemented by an address sign extension scheme involving the upper three address bits. In this scheme, address<31:30> are forced to the state of address<29>.
2. When the Mbox receives a virtual address, virtual address translation occurs normally without any sign extension of the resulting physical address. This is possible because the corresponding sign extension function is preprocessed on the upper three bits of page frame address which is written into the TB during the TB_TAG_FILL operation.

Note that restrictions exist about how the PAMODE register can be modified. See Section 12.8.2 for more information.

12.5.1.5 MEMORY MANAGEMENT EXCEPTIONS

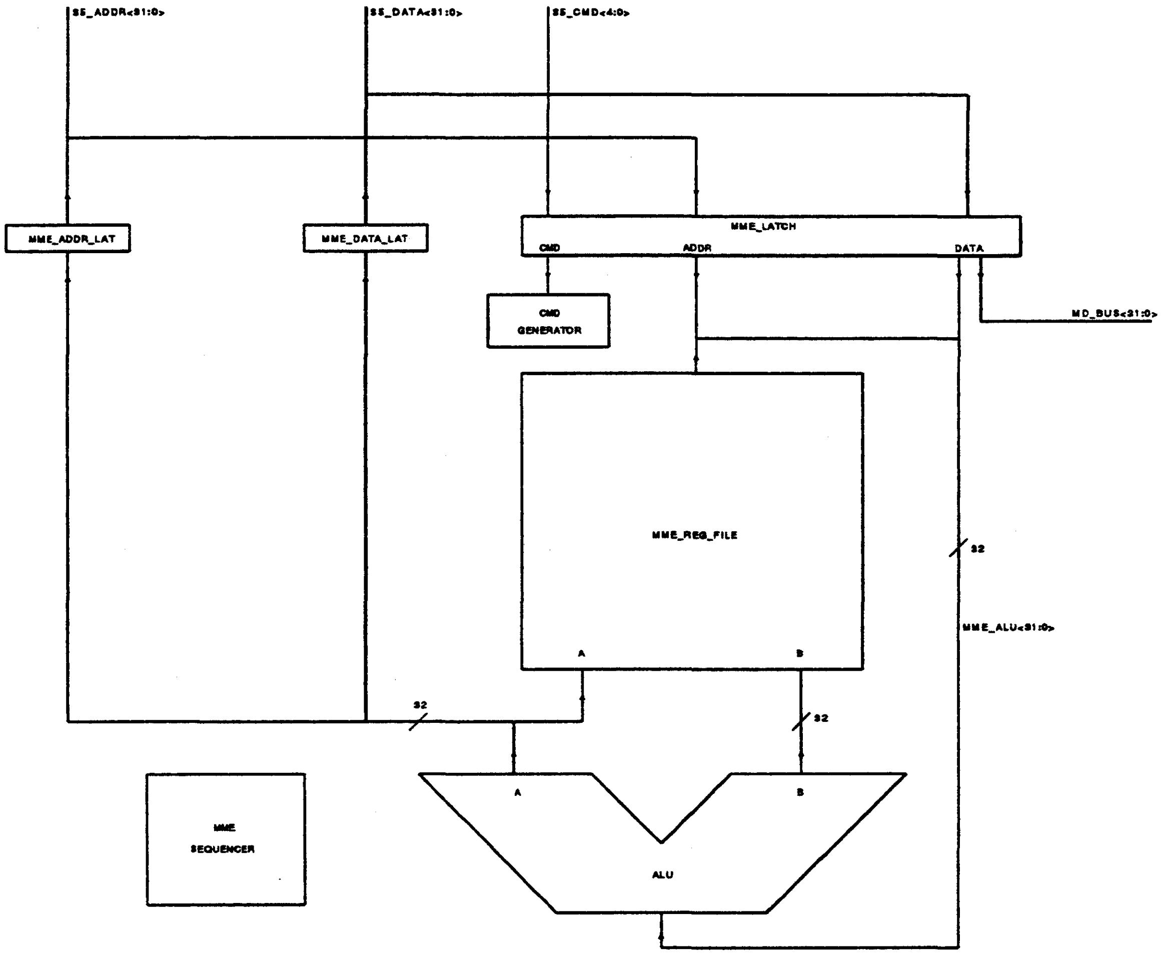
12.5.1.5.1 MME_DATAPATH

The MME_DATAPATH (Memory Management Datapath) is used to process most memory management functions performed by the Mbox. Specifically, it performs the following functions:

- Creates read references of PTEs in order to obtain virtual address translations not currently cached in the TB (See VAX Architecture Standard, DEC STD 032, for a description of this process).
- Creates TB fill operations in order to fill tag and PTE data in the TB.
- Stores most Mbox internal processor registers.
- Stores virtual addresses associated with memory management faults.
- Stores PTE addresses associated with M=0 faults.

The MME_DATAPATH is illustrated below:

Figure 12–52: MME Datapath



12.5.1.5.1.1 MME Register File

The register file has one write port and two read ports (one for each input to the ALU). The register file contains the following longword registers:

Reg Name	Definition
PAMODE	Address Mode Register: enables 30 or 32-bit address mapping

Reg Name	Definition
MMAPEN[12]	Mbox Map Enable Register: turns on/off virtual translations
MSLR[12]	Mbox System Length Register: Length of System Page Table
MSBR[1]	Mbox System Base Register: Addr of System Page Table
MP0LR[12]	Mbox P0 Length Register: Length of P0 Page Table
MP0BR[1]	Mbox P0 Base Register: Addr of P0 Page Table
MP1LR[12]	Mbox P1 Length Register: Length of P1 Page Table
MP1BR[1]	Mbox P1 Base Register: Addr of P1 Page Table
MMEADR[1]	MME Faulting Address Register
MMEPTE[1]	PTE Address Register
MMESTS[1]	Status of memory management exception
TBADR	Address of reference causing TB parity error
TBSTS	Status of TB parity error
TMP1	Scratch Register 1
TMP2	Scratch Register 2

[1] Testability and diagnostic use only; not for software use in normal operation.

[2] Ebox ucode sends and receives this data to/from the MME reg file shifted left 9-bits.

Note that the datapath associated with this register file performs all bit shifts associated with MME processing except for 9-bit shifts required on MMAPEN, MSLR, MPOLR, and MP1LR registers. The Ebox microcode sends pre-formatted data to these registers such that the data has been pre-shifted left nine bit positions. This facilitates the MME datapath implementation. IPR_RD operations from these registers send data back to the Ebox in the same format. Thus, the Ebox microcode will re-format the data back into the standard formats illustrated in Table 12–3.

Note that a 9-bit left shift is performed on MMAPEN so that the contents of MMAPEN can be used to increment a virtual address by a page in order to perform cross page check operations.

The MME_ADDR latch stores the address which was driven on **M_QUE%S5_VA_H**<31:0> during the previous cycle. The MME_DATA latch stores the data which was driven on **M_QUE%S5_DATA_H**<31:0> during the previous cycle. The A input to the ALU is either driven from MME_ADDR, MME_DATA, or the A read port of the register file.

12.5.1.5.1.2 MME ALU

The ALU (Arithmetic Logic Unit) performs the following functions:

- pass A: used for receiving addresses and data from main S5 pipe.
- pass B: used for reading/writing registers
- A + B: used to generate PTE addresses (note 9-bit right shift on A input)
- A - B: used for page table length checks of P0 and S0 space references (note 7-bit right shift on A input)

The output of the ALU can write the following:

- address field of the MME_LATCH (to generate PTE reads, TB tag fills and TB pte fills)
- data field of the MME_LATCH (to return requested IPR read data)
- the register file

12.5.1.5.1.3 MME_SEQ

The MME_SEQ is a state machine which controls sequencing of the MME_DATAPATH. It controls which devices drive and latch data in the MME_DATAPATH, what ALU function is to be executed, and what command gets generated and latched in the MME_LATCH, The possible MME state sequences of the MME_SEQ are illustrated by the following two diagrams below:

Figure 12–53: MME Sequences

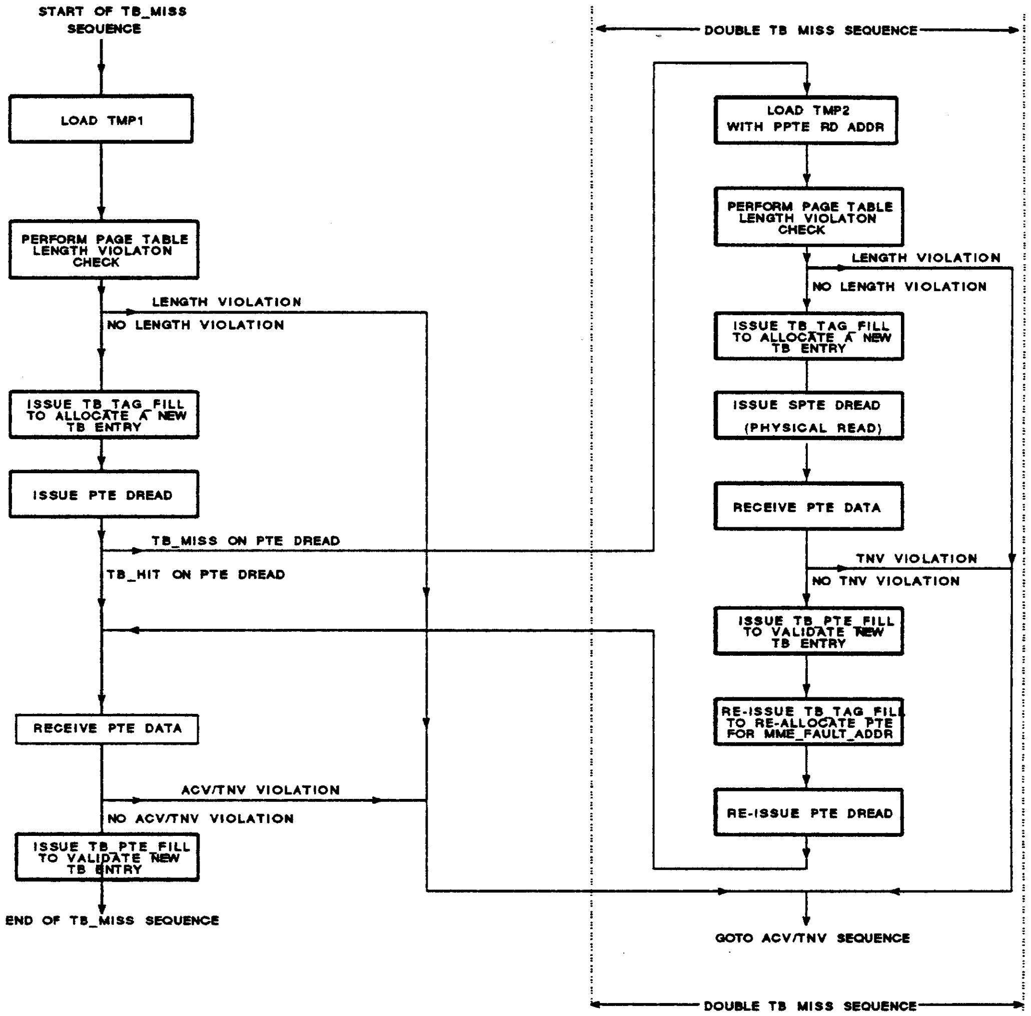
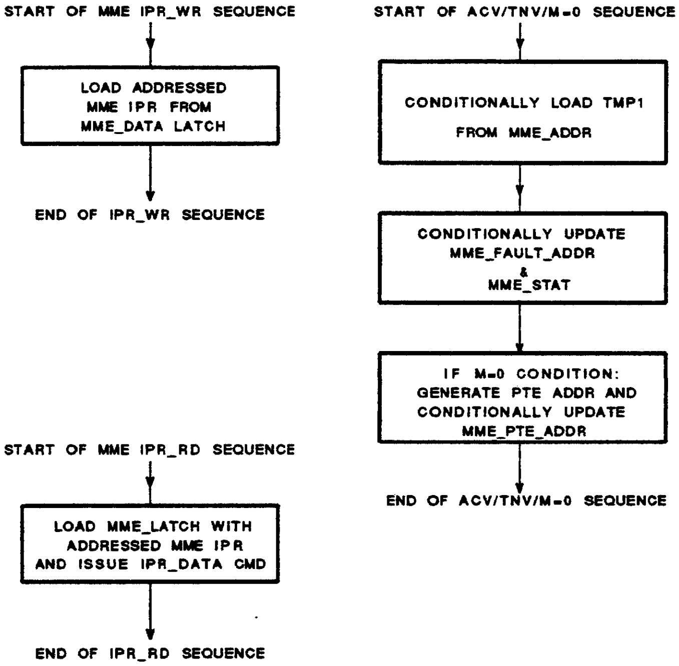

START OF TB_MISS SEQUENCE
LOAD TMP1
PERFORM PAGE TABLE LENGTH VIOLATON CHECK
LENGTH VIOLATION
NO LENGTH VIOLATION
ISSUE TB TAG FILL TO ALLOCATE A NEW TB ENTRY
ISSUE PTE DREAD
TB_MISS ON PTE DREAD
TB_HIT ON PTE DREAD
RECEIVE PTE DATA
ACV/TNV VIOLATION
NO ACV/TNV VIOLATION
ISSUE TB PTE FILL TO VALIDATE NEW TB ENTRY
END OF TB_MISS SEQUENCE
DOUBLE TB MISS SEQUENCE
LOAD TMP2 WITH PPTE RD ADDR
PERFORM PAGE TABLE LENGTH VIOLATON CHECK
LENGTH VIOLATION
NO LENGTH VIOLATION
ISSUE TB TAG FILL TO ALLOCATE A NEW TB ENTRY
ISSUE SPTE DREAD (PHYSICAL READ)
RECEIVE PTE DATA
TNV VIOLATION
NO TNV VIOLATION
ISSUE TB PTE FILL TO VALIDATE NEW TB ENTRY
RE-ISSUE TB TAG FILL TO RE-ALLOCATE PTE FOR MME_FAULT_ADDR
RE-ISSUE PTE DREAD
GOTO ACV/TNV SEQUENCE
DOUBLE TB MISS SEQUENCE

Figure 12–54: MME Sequences Cont'd


START OF MME IPR_WR SEQUENCE
LOAD ADDRESSED MME IPR FROM MME_DATA LATCH
END OF IPR_WR SEQUENCE
START OF MME IPR_RD SEQUENCE
LOAD MME_LATCH WITH ADDRESSED MME IPR AND ISSUE IPR_DATA CMD
END OF IPR_RD SEQUENCE
START OF ACV/TNV/M=0 SEQUENCE
CONDITIONALLY LOAD TMP1 FROM MME_ADDR
CONDITIONALLY UPDATE MME_FAULT_ADDR & MME_STAT
IF M=0 CONDITION: GENERATE PTE ADDR AND CONDITIONALLY UPDATE MME_PTE_ADDR
END OF ACV/TNV/M=0 SEQUENCE

There are five distinct entry points into the MME sequences:

- TB_MISS Entry Point: Whenever a TB_MISS condition is detected on an Ibox or Ebox reference, the MME_SEQ executes the sequence defined by the TB_MISS Entry Point.
- Cross Page Entry Point: The MME_SEQ executes the Cross Page Sequence in order to check for MME faults which may exist on the upper page of a reference that crosses a page boundary.
- ACV/TNV/M=0 Entry Point: The MME_SEQ can execute this sequence when an ACV, TNV, or M=0 condition is detected on an S5 reference, or when an ACV or TNV condition is detected during the TB miss sequence.
- MME IPR_RD Entry Point: The MME_SEQ executes this flow when an Mbox IPR register located in the MME_DATAPATH is addressed by an IPR_RD command.
- MME IPR_WR Entry Point: The MME_SEQ executes this flow when an Mbox IPR register located in the MME_DATAPATH is addressed by an IPR_WR command.

Once an MME sequence starts, the processing of all Ibox and Ebox references is inhibited until the sequence completes. Once the MME sequence terminates, normal processing resumes and the original reference which initiated the MME sequence will be retried.

12.5.1.5.2 TB MISS SEQUENCE

When memory management is enabled (MAPEN=1) and no valid tag entry in the TB matches the corresponding virtual page frame address applied on **M_QUE%S5_VA_H**<31:9>, the TB does not contain the necessary translation information to convert the address to physical space. In this situation, the TB asserts its TB_MISS signal which initiates a series of sequential events that will cause the proper PTE to be written into the TB.

12.5.1.5.2.1 Single Miss Sequence

A single miss sequence is defined as a TB miss sequence with only one TB miss occurring during the sequence. The following series of events characterizes a single TB miss sequence (see Figure 12–53 for a flow chart description of this sequence):

- cycle 1: TB asserts TB_MISS. S5 reference is aborted (will be retried later). MME_ADDR latches **M_QUE%S5_VA_H**.
- cycle 2: TMP1 is loaded from MME_ADDR in order to store the TB miss address in the MME register file.
- cycle 3: The proper page table length check is performed using TMP1, the appropriate XLR and a subtract ALU operation. If a length violation exists, the execution sequence continues in the ACV/TNV/M=0 sequence (See Section 12.5.1.5.3.6).
- cycle 4: The address field of the MME_LATCH is loaded with the TMP1 fault address and the MME_LATCH is validated with a TB_TAG_FILL command.
- cycle 5: The TB_TAG_FILL command executes in S5 (assuming no Cbox reference took priority) to allocate a TB entry corresponding to the TB miss address.

  The corresponding PTE address is formed using TMP1, the appropriate XBR and the A+B ALU operation. The PTE DREAD is loaded into the MME_LATCH.
- cycle 6: The PTE DREAD is started in S5 (assuming no Cbox reference took priority). If this is an SPTE (System Page Table Entry) DREAD, this reference is physical and, therefore, cannot have a TB_MISS and/or TNV condition associated with it. If this is a PPTE DREAD (Process Page Table Entry) DREAD, this reference is virtual and can have a TB_MISS and/or TNV condition associated with it. Since a single miss sequence is being described here, a PPTE DREAD hits in the TB by definition (see Section 12.5.1.5.2.2 for a description of when this reference misses).

  Note that no ACV protection checks are performed on this DREAD because it is an Mbox PTE DREAD. No TNV checks are performed because only PTEs with PTE<31> set are cached in the TB. No M=0 check is performed since this is strictly a read operation. Assuming TB miss problems occurred, the address is now properly translated and the DREAD continues into S6.
- cycle x: The PTE data is available on the **M%MD_BUS_H**<31:0>. This data is latched in the address field of the MME_LATCH. ACV/TNV checks are performed on the protection and valid bit fields of the incoming PTE data. If an ACV/TNV condition is detected, the memory management sequence continues in the ACV/TNV/M=0 sequence (See Section 12.5.1.5.3.6). If neither condition is detected, the MME_LATCH is validated with the TB_PTE_FILL command.
- cycle x+1: The TB_PTE_FILL command is executed in S5 (assuming no other Cbox command took priority) to load the PTE into the TB and validate the TB entry. Normal processing resumes and the reference which causes the original TB miss will be retried.

12.5.1.5.2.2 Double Miss Sequence

When the MME_DATAPATH generates a PPTE DREAD in order to resolve a TB miss, the PPTE address is itself a system virtual address. Therefore, it is possible for the PPTE DREAD to generate a second TB miss. In this case, the PPTE DREAD TB miss must be processed first in order to translate the PPTE DREAD address. Following this, the original TB miss sequence can resume in order to translate the initial faulting address. This scenario is called a double TB miss and is shown below (see Figure 12–53 for a flow chart description of this sequence):

- cycle 1: TB asserts TB_MISS. S5 reference is aborted (will be retried later). MME_ADDR latches **M_QUE%S5_VA_H**.
- cycle 2: TMP1 is loaded from MME_ADDR in order to store the TB miss address in the MME register file.
- cycle 3: The proper page table length check is performed using TMP1, the appropriate PXLR and a subtract ALU operation. If a length violation exists, the execution sequence continues in the ACV/TNV/M=0 sequence (See Section 12.5.1.5.3.6).
- cycle 4: The data field of the MME_LATCH is loaded with the TMP1 fault address as the MME_LATCH is validated with a TB_TAG_FILL command.
- cycle 5: The TB_TAG_FILL command executes in S5 (assuming no Cbox reference took priority) to allocate a TB entry corresponding to the TB miss address.

  The corresponding PPTE address is formed using TMP1, the appropriate PXBR and the A+B ALU operation. The PPTE DREAD is loaded into the MME_LATCH. Note that because the Mbox generated a PPTE DREAD as part of a TB miss sequence, the virtual reference is loaded into the MME_LATCH with the ACV/M=0 reference qualifier cleared so that ACV checks will not be performed on the reference.
- cycle 6: The PPTE DREAD is started in S5 (assuming no Cbox reference took priority). The TB asserts TB_MISS again because the PPTE address translation was not present in the TB. MME_ADDR latches the PPTE DREAD address and the DREAD is aborted.
- cycle 7: TMP2 is loaded from the MME_ADDR with the PPTE DREAD address.
- cycle 8: The system page table length check is performed using TMP2, SLR and the A-B ALU operation. If a length violation exists, the execution sequence continues in the ACV/TNV/M=0 sequence (See Section 12.5.1.5.3.6).
- cycle 9: The address field of the MME_LATCH is loaded with the TMP2 PPTE fault address as the MME_LATCH is validated with a TB_TAG_FILL command.
- cycle 10: The TB_TAG_FILL command executes in S5 (assuming no Cbox reference took priority) to allocate a TB entry corresponding to the TB miss address. Note that the TB entry that is allocated destroys the previous TB entry allocation for the original TB miss because the NLU TB allocation pointer has not moved.

  The corresponding SPTE address is formed using TMP2, SBR and the A+B ALU operation. The SPTE DREAD is loaded into the MME_LATCH.
- cycle 11: The SPTE DREAD is started in S5 (assuming no Cbox reference took priority). Note that this DREAD has a physical address. Therefore, no memory management problem can occur on this read.
- cycle x: The SPTE data is available on the **M%MD_BUS_H**<31:0>. This data is latched in the address field of the MME_LATCH. ACV/TNV checks are performed on the protection and valid bit fields of the incoming PTE data. If an ACV/TNV condition is detected, the memory management sequence continues in the ACV/TNV/M=0 sequence (See Section 12.5.1.5.3.6).

If neither condition is detected, the MME_LATCH is validated with the TB_PTE_FILL command.

- cycle x+1: The TB_PTE_FILL command is executed in S5 (assuming no other Cbox command took priority) to load the SPTE into the TB and validate the TB entry. Note that the NLU TB allocation pointer is incremented on a TB_PTE_FILL operation.

  In order to re-allocate a TB entry for the original TB miss address, the address field of the MME_LATCH is loaded with TMP1 while the command field is loaded with a TB_TAG_FILL command.
- cycle x+2: The TB_TAG_FILL command is executed in S5 (assuming no other Cbox command took priority) to re-allocate a TB entry corresponding to the original TB miss.

  The original PPTE address is re-generated using TMP1, the appropriate PXBR and the A+B ALU operation. The PPTE DREAD is loaded into the MME_LATCH (ACV checks are once again disabled for this reference).
- cycle x+3: The PPTE DREAD is started in S5 (assuming no Cbox reference took priority).

  Note that no ACV protection checks are performed on this DREAD because it is an Mbox PTE DREAD. No TNV checks are performed because only PTEs with PTE<31> set are cached in the TB. No M=0 check is performed since this is strictly a read operation. The PPTE DREAD address is now properly translated.
- cycle y: The PPTE data is available on M%MD_BUS_H<31:0>. This data is latched in the address field of the MME_LATCH. ACV/TNV checks are performed on the protection and valid bit fields of the incoming PTE data. If an ACV/TNV condition is detected, the memory management sequence continues in the ACV/TNV/M=0 sequence (See Section 12.5.1.5.3.6). If neither condition is detected, the MME_LATCH is validated with the TB_PTE_FILL command.
- cycle y+1: The TB_PTE_FILL command is executed in S5 (assuming no other Cbox command took priority) to load the PPTE into the TB and validate the TB entry. Normal processing resumes and the reference which caused the original TB miss will be retried.

MICROCODE RESTRICTION

To avoid a potential infinite loop case whereby the Mbox is stuck in the TB double miss sequence forever, the Ebox microcode must guarantee that it issues a non-STORE instruction other than TBIA, TBIS, or TB_TAG_FILL during the cycle immediately preceding the cycle it issues either a TBIA, TBIS or TB_TAG_FILL instruction.

12.5.1.5.3 ACV/TNV/M=0

12.5.1.5.3.1 ACV/TNV/M=0 Fault Handling:

In order for an ACV, TNV, or M=0 fault to be processed, the following steps must occur:

1. The Mbox must detect the ACV/TNV/M=0 condition.
2. The Ebox microcode must be invoked to start processing the condition.
3. The Ebox microcode must probe Mbox state in order to determine which fault occurred and how it should be processed.
4. The Ebox microcode must service the fault condition directly, or it must invoke an operating system memory management service routine to service the fault.

5. If the memory management fault was not fatal to the process, normal instruction execution resumes by restarting the instruction corresponding to the memory management fault after servicing the fault.

12.5.1.5.3.2 ACV detection:

The protection field of a PTE indicates the authorized access rights for each execution mode. When a reference causes the TB to access a PTE, the protection field of the PTE corresponding to the reference is driven out of the TB. The ACV (Access Violation) detection logic uses the PTE protection field, **M_QUE%S5_AT_H**<1:0>, and the appropriate CPU execution mode from the Ebox (i.e. user, supervisor, executive, kernel) to detect access violations. If, for example, the protection field indicates a "read-only" access in user mode, the CPU execution mode specifies user mode, and **M_QUE%S5_AT_H**<1:0> indicates write access, then an ACV condition is flagged since a write reference is not allowed to this page in user mode.

A 2:1 MUX controls the source of the CPU execution mode. The CPU execution mode information is normally taken directly from the current mode field of the PSL (**PSL**<25:24>). On PROBE references, however, the CPU execution mode is driven from **E%MMGT_MODE_H**<1:0> in order to check for ACV conditions for an execution mode which the CPU is not currently in.

An ACV condition is also generated when a PTE reference fails to satisfy the page length check corresponding to the virtual space of the reference or when the virtual reference falls into reserved page region of virtual memeory (FFFFFE00-FFFFFFFF). Either condition is reported as an ACV length violation.

An ACV check is also performed on the protection field of all PTEs which have just been sent to the Mbox due to an earlier Mbox DREAD issued during the TB_MISS sequence.

ACV protection and length checks are performed on all Ibox and Ebox references and on all MME_CHKs. ACV page length checks are performed on all PTE addresses. However, ACV protection checks are never performed on PTE read references generated by the Mbox.

Note that the ACV protection condition is disabled from occurring during any cycle where the reference is aborted.

When an ACV condition occurs, the MME_SEQ is invoked to execute the ACV/TNV/M=0 sequence. ACV checks only occur on virtual addresses when memory management is enabled and when the reference indicates that memory management checks should be done (i.e. **M_QUE%S5_QUAL_H<2>** = 1).

12.5.1.5.3.3 TNV detection

When the PTE valid bit is clear, it indicates that the corresponding PTE page frame address translation is not valid. This is called a Translation Not Valid Fault (TNV). TNV detection only occurs during the TB_MISS sequence when the Mbox receives PTE data from the Pcache or Cbox such that the PTE valid bit (PTE<31>) is clear. When a TNV fault is detected, the MME_SEQ interrupts the TB_MISS sequence and invokes the ACV/TNV/M=0 sequence. By doing so, the invalid PTE is never cached in the TB and a memory management fault is recorded (See Section 12.5.1.5.3.5 on recording memory management faults).

12.5.1.5.3.4 M=0 detection:

When a virtual reference causes the TB to access a PTE, the modify bit of the PTE is read out of the TB. A cleared modify bit indicates that the corresponding page has not been written to. If the valid bit of the PTE is set, and the modify bit is clear and the access type of the S5 reference indicates an intention to modify the page (e.g. write or modify access type), then the Mbox must initiate the proper sequence of events to process this "M=0" condition. The M=0 check is performed when memory management is enabled and a virtual reference hits in the TB.

Note that the M=0 condition is disabled from occurring during any cycle where the reference is aborted.

12.5.1.5.3.5 Recording ACV/TNV/M=0 Faults

In order for the microcode to determine the nature of the memory management fault detected by the Mbox, the Mbox must record the necessary fault information. The fault information is recorded in Mbox IPRs which can be read by Ebox microcode. The fault information is stored in three of the registers in the MME register file which are accessible to microcode by IPR reads and writes:

- The MMEADR register stores the virtual address associated with the ACV, TNV or M=0 fault. As per SRM requirements, if the ACV/TNV fault occurred by referencing a PTE during a TB miss sequence, the MMEADR stores the original address and not the PTE address.
- The MMEPTE register stores the virtual or physical address of the Page Table Entry corresponding to a virtual reference upon which an M=0 condition has been detected.
- The MMESTS register stores state which indicates to the microcode the context and type of fault corresponding to the ACV/TNV/M=0 condition. The format of MMESTS is shown below:

Figure 12–55: IPR EA (hex), MMESTS

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|           |  SRC      | 0| 0| 0| 0| 0| 0| 0| 0| 0| 0|FAULT| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| M|   |LV|:MMESTS
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
<---+---->                                                                                          |
    |                                                                                               |
    +---- LOCK                                                                            PTE_REF--+
```

Table 12–19: MMESTS Field Descriptions

Name	Extent	Type	Description
LV	0	RO	Indicates ACV fault occurred due to length violation.
PTE_REF	1	RO	Indicates ACV/TNV fault occurred on PTE reference corresponding to MMEADR.
M	2	RO	Indicates corresponding reference had write or modify intent.
FAULT	15:14	RO	Indicates nature of memory management fault. See Fault bit encodings below

Table 12–19 (Cont.): MMESTS Field Descriptions

Name	Extent	Type	Description
SRC	28:26	RO	Complemented shadow copy of LOCK bits. However, the SRC bits do not get reset when the LOCK bits are cleared.
LOCK	31:29	RO	Indicates the lock status of MMESTS. See LOCK encodings below. This field is cleared on E%FLUSH_MBOX_H.

Table 12–20: LOCK Encodings

Defined LOCK values (binary)	Definition
000	MMESTS, MMEADR and MMEPTE are unlocked.
001	valid IREAD fault is stored (no other IREAD fault can overwrite MMESTS, MMEADR, or MMEPTE).
011	valid Ibox specifier fault is stored (only an Ebox reference fault can overwrite MMESTS, MMEADR, or MMEPTE).
111	valid Ebox fault is stored (MMESTS, MMEADR, and MMEPTE are completely locked).

Note that the encodings for the SRC bits are the complemented version of the the LOCK bits. Thus, for example, a fully locked SRC encoding is 000.

Table 12–21: FAULT Encodings

Defined FAULT values (binary)	Definition
01	ACV Fault. This is the highest priority fault in the presence of multiple simultaneous faults.
10	TNV Fault. This is the next highest priority fault.
11	M=0 Fault. This is the lowest priority fault.

Due to the macropipeline design, the MMEADR, MMEPTE and MMESTS registers must be conditionally loaded in a prioritized fashion. These registers are loaded depending on the relative states of their current contents and on the context of the current fault. If the MMESTS register is empty, the current fault state is always loaded. If the MMESTS register contains a valid fault condition, the MMEADR, MMEPTE and MMESTS are only loaded if the current fault is associated with a pipe stage further along in the pipe than the stage corresponding to the stored MMESTS state. This loading priority is necessary because these memory management faults must be reported within the context of the execution of the instruction they are associated with. A fault detected on an Ebox reference is loaded provided that another Ebox reference fault is not already loaded. Faults detected on Ibox specifier references are only loaded if no Ebox or Ibox specifier reference fault is currently stored. Faults on Ibox I-stream references are only loaded if the MMESTS register is empty. In effect, the MMESTS register captures the first memory management exception that will be associated with Ebox execution. Stated differently, it captures the fault which occurs farthest along in the macropipeline.

The LOCK field of MMESTS specifies the source of the faulting reference currently stored in MMESTS. Thus, the decision to load another faulting reference into MMESTS is made by examining the bits of the LOCK field.

The FAULT field is set in a prioritized manner. That is, an ACV fault takes precedence over a TNV or M=0 fault. A TNV fault takes precedence over an M=0 fault. Therefore, if multiple pending fault conditions are true, only the fault condition with the highest priority is reported in the MMESTS register.

When the Ebox starts the memory management exception microflow, it issues an IPR_RD to the MMESTS to determine the nature of the memory management fault. The MMESTS register is automatically unlocked by resetting the LOCK field when the **E%FLUSH_MBOX_H** signal is asserted by the Ebox.

12.5.1.5.3.6 ACV/TNV/M=0 MME_DATAPATH Sequence

When an ACV/TNV/M=0 condition occurs the MME_DATAPATH performs the following actions in order to record the fault for subsequent use by the Ebox microcode.

- cycle 1: ACV, TNV, or M=0 condition is detected. MME_ADDR latches **M_QUE%S5_VA_H** address. Note that the S5 reference is NOT aborted.

  If the faulting reference is associated with an Ebox reference, **M%MME_TRAP_L** is asserted to the microsequencer to generate a memory management microtrap. If the faulting reference was associated with a DEST_ADDR command, the MME fault is logged in the corresponding PA_QUEUE entry. In all other cases (IREADs and Ibox D-stream reads) **M%MME_FAULT_H** qualifies the **M%MD_BUS_H** indicating that the requested data had a memory management problem.
- cycle 2: If this ACV/TNV/M=0 sequence was not invoked from a previous MME_SEQ flow, the contents of MME_ADDR are loaded into TMP1. If this sequence was invoked from another MME_SEQ flow, TMP1 is not loaded because it already contains the original address that must be reported for this ACV/TNV condition.
- cycle 3: The source of the reference which directly/indirectly invoked the MME fault is compared to MMESTS<31:29> (the LOCK field) to determine whether this fault should be recorded in MMEADR, MMEPTE, and in MMESTS. If a previous fault of equal or greater priority is already stored in MMESTS, MMESTS, MMEADR, and MMEPTE are not updated.

  If the LOCK field indicates that this fault should be recorded, MMEADR is loaded from TMP1 and MMESTS is updated as follows:

Table 12–22: MMESTS State Update

fault type	MMESTS<15:14>	MMESTS<2:0>
ACV without MME_SEQ active (no modify intent)	01	000
ACV without MME_SEQ active (modify intent)	01	100
M=0	11	100
length violation on ref during TB_MISS seq (no modify)	01	001
length violation on ref during TB_MISS seq (modify intent)	01	101

Table 12–22 (Cont.): MMESTS State Update

fault type	MMESTS<15:14>	MMESTS<2
length violation on PTE ref during TB_MISS seq (no modify intent on original reference)	01	011
length violation on PTE ref during TB_MISS seq (modify intent on original reference)	01	111
TNV on PTE ref during TB_MISS seq (no modify intent on original reference)	10	010
TNV on PTE ref during TB_MISS seq (modify intent on original reference)	10	110

The LOCK field of MMESTS is updated appropriately.

- cycle 4: If MMESTS was updated during cycle 3, and the fault was M=0, the corresponding PTE address is formed using TMP1, the appropriate XBR and A+B ALU operation. The PTE address is then loaded into MMEPTE.

12.5.1.5.3.7 Microcode Invocation of ACV/TNV/M=0

Microcode is invoked for ACV/TNV/M=0 faults in three different ways:

- If the faulting reference originated from the Ebox, then the Mbox asserts **M%MME_TRAP_L** to invoke a memory management microtrap. **M%MME_TRAP_L** is asserted at the end of the cycle in which the ACT/TNV/M=0 fault was detected. Thus, from a microcode point of view, the microtrap happened before the EM_LATCH contents were retired. This microtrap invokes the ACV/TNV/M=0 microflow which handles the fault in the context of the reference executing in the Ebox.
- If the faulting reference is a read sourced by the Ibox (either a D-stream or I-stream read), **M_QUE%S5_QUAL_H<0>** is set indicating that a memory management fault should be forced on this read. When the read propagates into S6, the Mbox forces the Pcache to hit and returns invalid data. This data, however, will be qualified with the **M%MME_FAULT_H** signal to indicate that the data is invalid and that an ACV/TNV/M=0 fault is associated with this data. When the Ebox references the corresponding D-stream operand, or requires the decode of the corresponding I-stream data, a microtrap is generated by the Ebox to invoke the ACV/TNV/M=0 microflow.

  If an MME fault occurs on the address of the address of an operand (i.e. Ibox decoding a deferred specifier), the Mbox records the fault in MMEADR and MMESTS in the usual way and returns data qualified by **M%MME_FAULT_H**. In some instances, the Ibox must issue a second reference to the Mbox based on the address returned by the first reference. Due to the fault however, the Ibox cannot issue a valid operand read address since the data returned by the first reference was invalid. In this case, the Ibox issues a read qualified with the **I%FORCE_MME_FAULT_H** signal. This causes the Mbox to "fake" an ACV/TNV violation by qualifying the returned data with **M%MME_FAULT_H**. This reference is trapped on when the Ebox references the operand.

  Note that when the Mbox "fakes" an ACV/TNV/M=0 violation, the MME_DATAPATH does not invoke a memory management response to either an ACT/TNV/M=0 problem or to a TB_MISS. Further, no state update is performed for either the MMESTS or MME_ADDR. Thus, these registers still record the true ACV/TNV error.

- If the faulting reference is a DEST_ADDR, an ACV/TNV/M=0 bit in the PA_QUEUE is set in the corresponding PA_QUEUE entry. When the Ebox microcode checks for the validity of the PA_QUEUE in order to send the corresponding STORE data, the Ebox detects the ACV/TNV/M=0 condition and generates the microtrap.

  The PA_QUEUE hardware must guarantee that the first PA_QUEUE entry of an unaligned pair of entries must be marked with the ACV/TNV/M=0 condition regardless of which of the two references caused the fault. This is necessary so that the microcode takes the proper action at the start of the reference.

If an ACV length violation or a TNV fault is generated on an Mbox PTE reference, the original reference (i.e. the reference that caused the memory management sequence which generated the PTE reference) must be marked as having an MME fault associated with it. Thus, when the original reference is retried after the memory management sequence completes, the reference will be treated as if the MME fault was associated with it. Note that the MMESTS register records the fact that the actual fault was associated with the PTE reference and not the original reference.

12.5.1.5.3.8 Microcode Processing of ACV/TNV/M=0:

The NVAX macropipeline design can cause synchronization problems related to operating system processing of PTEs. The SRM states that "software is not required to flush TB entries after changing PTEs that were already invalid." Consider the case where an Ibox read prefetches an invalid PTE from a page table. Just after this read, the Ebox completes the previous macroinstruction by updating, validating and writing the same PTE back to memory. When the Ebox references the prefetched PTE operand, an invalid TNV fault will be generated because the PTE has just been validated.

To prevent this scenario from occurring, the memory management fault microcode must re-test for fault conditions before invoking the actual fault sequence. If no fault is detected at this time, no fault processing occurs. Microcode re-tests the fault conditions by first asserting E%FLUSH_MBOX_H, which unlocks MMESTS and clears pending Mbox references. Following this, the microcode reads the fault address from MMEADR via an IPR_RD command and then issues a TBIS command corresponding to this faulting reference. The TBIS will clear out the potentially out-of-date PTE in the TB which is associated with the fault. The microcode will then issue a PROBE command to the same address. The PROBE will cause the updated PTE to be cached in the TB (unless a TNV fault is detected) and will record the new fault status in MMESTS and return the status to the Ebox. Note that the PROBE command does not lock MMESTS. If the microcode detects a valid fault upon reading the PROBE status, microcode fault processing continues. Otherwise, the instruction is restarted without causing a memory management fault.

If a real ACV or TNV fault was detected, it re-reads MMESTS to get the updated status based on the last PROBE operation. The microcode constructs and pushes the memory management fault stack frame consisting of the fault status, the contents of MMEADR, the PC of the corresponding instruction, and the PSL at the time of the fault. The microcode then reads the appropriate SCB (System Control Block) vector corresponding to either the ACV or TNV fault. Based on this vector, the microcode sets the appropriate CPU execution mode and redirects the PC to the appropriate operating system memory management macrocode fault handler. This software fault handler reads the fault status and the faulting address from the stack and processes the ACV or TNV fault based on this information. Once the fault is processed, an REI is executed, the macropipeline is flushed, and normal instruction processing resumes by restarting the instruction that originally caused the fault.

If the microcode read MMESTS and determined the fault to be an M=0 condition, the microcode processes the fault without the aid of operating system sofware. To do this, the microcode performs the following actions:

1. A TBIS command is issued which references the faulting address. This reference will cause the PTE, which was used to detect the M=0 fault, to be invalidated from the TB.
2. The microcode will then test the faulting address to determine whether it was a process or system space reference. If it was a system space reference, the corresponding SPTE address must be a physical address. If it was a process space reference, the corresponding PPTE address must be a virtual address.
3. The microcode then issues a DREAD using the PTE address it read from MMEPTE. If the microcode determined the PTE to be an SPTE, the read is issued with **M_QUE%S5_QUAL_H**<6>=0 indicating a physical read. If the microcode determined the PTE to be a PPTE, the read is issued with **M_QUE%S5_QUAL_H**<6>=1 and **M_QUE%S5_QUAL_H**<2>=1 indicating a virtual read with ACV and M=0 checks disabled because the Mbox must not perform M=0 checks and ACV protection checks on PTE references.
4. When the PTE data is received, the Ebox sets the modify bit of the PTE indicating that the corresponding page is written. The new PTE is then written back into the page table in memory by issuing a physical WRITE or a virtual write with ACV/M=0 checks disabled, depending on the physical or virtual nature of the PTE.
5. The microcode then flushes the macropipeline and resumes normal instruction processing by restarting the instruction corresponding to the M=0 fault.

Note that when the address which caused the M=0 fault is restarted after the M=0 fault was serviced, the Mbox will generate a TB_MISS condition since the old PTE was invalidated from the TB. Subsequently, a TB_MISS sequence will be invoked which will cause the new PTE to be read into the Mbox and cached in the TB.

12.5.1.5.3.9 Pipeline Implications of ACV/TNV/M=0 condition

12.5.1.5.3.9.1 Pipeline Effects for MME Faults on Write References

If an ACV, TNV or M=0 condition occurs on a write reference, the faulting write is transformed into a NOP command in the S6 pipe. Thus, the Pcache and Bcache are prevented from modifying any memory state as a result of a memory management fault detected in S5.

12.5.1.5.3.9.2 Pipeline Effects for MME Faults on Read References

If the faulting reference is a read, the read must be prevented from leaving the Mbox pipe since a read to I/O space could cause detrimental state changes. This is handled by forcing the deassertion of **M%CBOX_REF_ENABLE_L** which causes the Cbox to ignore the read.

12.5.1.5.3.9.3 Pipeline Effects of E%FLUSH_MBOX_H on MME State

A more subtle implication involving the NVAX macropipeline exists which affects updating recorded Mbox MME state. Since the MME_SEQ executes independently of the Ebox microcode, the MME_SEQ must appropriately synchronize to Ebox execution such that MME state will not be updated for references that will never be processed by the Ebox.

Consider the following situation. A tb_miss sequence has begun on a specifier reference. During this sequence, the Ebox detects a branch mispredict which causes redirection of the processing stream. As the PTE data is returned to the Mbox, a TNV condition is detected. This TNV must not be recorded because it corresponds to a reference which the Ebox will not see due to the redirection of the execution stream.

From the Mbox point of view, handling this scenario can be generalized as follows. If the Mbox receives a **E%FLUSH_MBOX** signal during any memory management sequence which may update mme state, one of three possibilities will happen:

1. If **E%FLUSH_MBOX** is received after MME state has been updated, **E%FLUSH_MBOX** will unlock MMESTS so that only MME state corresponding to the redirected execution stream will be recorded.
2. If **E%FLUSH_MBOX** is received during the cycle that an mme state update is being done, the functional effect of **E%FLUSH_MBOX** will predominate, thus causing the MMESTS to be unlocked.
3. If **E%FLUSH_MBOX** is received before the state update, MMESTS will be cleared by **E%FLUSH_MBOX** and a state bit will be set which will inhibit any mme state updates during the remaining mme sequence.

Note that the analogous problem exists when processing a memory management sequence on an IREAD when **I%FLUSH_IREF_LAT_H** is asserted. In this case, the following three possibilities can occur:

1. If **I%FLUSH_IREF_LAT_H** is asserted when MMESTS contains a validated fault on an IREAD, **I%FLUSH_IREF_LAT_H** will unlock MMESTS.
2. If **I%FLUSH_IREF_LAT_H** is asserted during the cycle that an mme state update is being done on an IREAD reference, the functional effect of **I%FLUSH_IREF_LAT_H** will predominate, thus causing MMESTS to be unlocked.
3. If **I%FLUSH_IREF_LAT_H** is received before a MMESTS update but during a memory management fault sequence invoked from an IREAD, MMESTS will be cleared by **I%FLUSH_IREF_LAT_H** and a state bit will be set which will inhibit the subsequent mme state update.

Note that while a special state bit is necessary to synchronize MME updates with Ebox execution stream redirection, no special mechanism is required to keep TB state synchronized. There are two reasons for this. First, the TB never validates a PTE whose PTE valid bit is clear. Secondly, the Mbox arbitration logic prevents Ebox references such as TBIS, TBIP, and TBIA from executing when a memory management sequence is executing. Therefore, TB state updates are always serialized with respect to TB invalidates generated by the Ebox microcode.

12.5.1.5.3.9.4 Pipeline Effects of E%FLUSH_MBOX_H on M%MME_TRAP_L

Just as **E%FLUSH_MBOX_H** must be examined in order that MME state remains synchonized to Ebox execution, **E%FLUSH_MBOX_H** must also be factored into the logic which generates **M%MME_TRAP_L**. This prevents the following scenario from occurring. If the Ebox has issued a DREAD which misses in the Pcache as a result of a MOVC instruction, the Mbox will propagate the reference forward to the Cbox. While the read is pending, the Ebox issues an MME_CHK command which TB misses causing the Mbox to initiate a TB miss sequence. During this sequence, the Cbox returns the read data qualified by **C%CBOX_HARD_ERR_H**. This causes the Ebox to microtrap into the error handler resulting in the assertion of **E%FLUSH_MBOX_H**. If the

Mbox were to subsequently assert **M%MME_TRAP_L** based on a memory management fault on the MME_CHK command, the Ebox would microtrap out of the error handler and initiate MME fault sequence that should never occur.

Thus, the assertion of **E%FLUSH_MBOX_H** during a memory management sequence inhibits the assertion of **M%MME_TRAP_L** during that cycle or any subsequent cycles of the memory management sequence.

12.5.1.5.4 Cross Page Sequence

When an unaligned virtual reference falls across a page boundary, ACV/TNV/M=0 checks must be performed on both pages before the Mbox can determine if the reference passes or fails ACV checks. The function of the cross-page sequence is to generate an MME_CHK reference to check the second page (i.e. the upper page) for ACV/TNV/M=0 problems. As long as the MME_CHK clears memory management checks before the reference is allowed to execute, the reference can be processed in the normal manner because ACV/TNV/M=0 checks on the first page (i.e. the lower page) will naturally occur as they do on all virtual references. If an ACV/TNV problem is found on either page, an ACV/TNV condition is flagged for the reference.

When the cross-page detection logic flags a cross-page condition, the following cross-page sequence is invoked:

- cycle 1: The cross-page condition is detected. The S5 reference is aborted. The MME_ADDR latches the **M_QUE%S5_VA_H** address.
- cycle 2: The MME_DATAPATH adds 512 to the address in MME_ADDR. The resulting address is guaranteed to fall into the upper page of the original reference for all byte, word, longword and quadword references. This address is loaded into the MME_LATCH qualified by an MME_CHK command. The MME_CHK reference (with DL=byte) will perform memory management checks on the upper page.
- cycle 3: The MME_CHK is executed in S5 (assuming no Cbox reference took priority). If a TB_MISS occurs, the TB_MISS sequence is first invoked to obtain the proper translation. Once the TB has been updated based on the TB_MISS, the original MME_CHK reference will be restarted and the cross-page sequence will be re-invoked from the beginning.

  When the translation of the MME_CHK reference has properly occurred, ACV/M=0 checks are performed (note that TNV checks are only performed when the PTE is to be filled in TB). If an ACV/TNV/M=0 fault is detected during the MME_CHK processing, **M_QUE%S5_QUAL_H**<0> of the original reference, which caused the cross-page sequence, is set. Thus, when this reference is restarted, an MME fault will be reported. If no ACV/TNV/M=0 condition was detected on the upper page, the original reference is marked as having passed the cross-page condition (**M_QUE%S5_QUAL_H**<5> is set).
- cycle x: The original reference is restarted. If no ACV/TNV/M=0 fault occurred on the upper page the reference executes normally without further cross-page checks.

  If the reference was marked as having an MME fault, the reference fault will be reported in the previously-described fashion (see Section 12.5.1.5.3.7).

The cross-page sequence is only invoked on a virtual reference when memory management is enabled.

12.6 MBOX ERROR HANDLING

12.6.1 Types of Errors Handled

Mbox plays a role in the processing of the following types of errors:

- TB tag parity errors.
- TB data parity errors.
- Pcache tag parity errors.
- Pcache data parity errors.
- Errors encountered by the Cbox while processing a memory read, I/O space read, or IPR_RD which were transferred from the Mbox to the Cbox. Note that these errors could originate from the Bcache, NDAL or memory subsystem.

All other possible errors are handled without Mbox involvement.

12.6.2 TB parity error detection

12.6.2.1 TB tag parity error detection

Conceptually, a single bit of even parity representing TB tag parity is stored in each TB entry. Whenever a valid tag entry matches the S5 virtual page address, the corresponding tag parity data is accessed and driven out of the TB array for a subsequent parity check. Thus tag parity errors are only detected on the entry which causes a TB hit condition.

The value of tag parity with which the stored parity data is compared to is calculated in parallel with the TB access by using the virtual page address found on **M_QUE%S5_VA_H<31:9>**. This scheme eliminates the need to drive out the matched tag entry in order to calculate parity. If the tag matched the virtual page address, then the correct parity value can be derived from **M_QUE%S5_VA_H<31:9>** instead of from the stored tag. This scheme is called predicted parity.

Tag parity in a fully associative cache can cause several different failure modes since the tag state directly determines which entry (or entries) are selected during each TB access. Assuming a single bit soft failure occurs in a single TB tag (i.e. a tag bit accidentally toggles due to some transient failure mode), three possible failure modes are possible:

1. A single bit tag error can cause no TB entry to match because the tag no longer compares with the virtual page address that it should have compared to. Thus, a TB_MISS condition is generated which causes the PTE data to be accessed from memory. This PTE data, along with its corresponding tag, will be written into a TB entry. In effect, this scenario causes the single bit tag error to remain undetected, but does not corrupt the virtual address translation process.
2. A single bit tag error may cause exactly one TB entry to match because the incorrect tag entry happens to match a virtual page address which is not already cached in the TB. In this situation, the tag parity read out of the TB is guaranteed not to match the virtual page address parity. Thus a TB tag parity error will be correctly detected.
3. A single bit tag error may cause two TB entries to match because the incorrect tag entry happens to match a virtual page address which is already cached in the TB. Thus, the correct tag entry detects a match at the same time as the incorrect tag entry detects a match.

Due to the wired OR function implicit in accessing data off of a shared bit line within the TB array, it is possible that the tag parity read out of the array matches the parity of the virtual page address causing no tag parity error to be detected. In this case, the wired OR function on the PTE bit lines will OR the two accessed PTE entries together causing an incorrect PTE to be read out. If an even number of PTE bits were corrupted by the simultaneous PTE access, the parity logic associated with the PTE data will not detect a problem. This is a disatrous situation to the currently-executing CPU process because the TB will produce an incorrect translation without producing a parity error.

As a result of the undetected fatal parity error discussed in this third case, a single bit of tag parity is stored in both its true and complement form in each TB entry. For a single entry match, these two parity lines always produce a "01" or "10" value. Due to the wired OR access, a two-entry TB match due to a single bit tag parity error, produces a "11" parity access indicating a multiple tag match and a tag parity error.

TB tag parity is written along with the tag during a TB_TAG_FILL operation.

12.6.2.2 TB data parity error detection

Data parity error detection is conceptually simpler than tag parity detection. When a TB hit condition occurs the accessed PTE data is driven out of the TB along with the corresponding stored data parity. Parity is then calculated on the data and compared with the stored parity. A miscompare results in a TB data parity error. TB data parity is a single bit corresponding to the entire stored PTE field.

TB data parity is written along with the PTE data during a TB_PTE_FILL operation.

12.6.3 Pcache parity error detection

12.6.3.1 Pcache tag parity error detection

Pcache tag parity is stored and checked as a single bit representing even parity across the entire 20-bit tag field. Unlike the TB implementation however, true and complement versions of single bit tag parity are not implemented–only the true version is implemented.

There are two separate aspects to Pcache tag parity error detection. The first aspect employs the "predicted parity" scheme which was used for the TB. However, the Pcache does not use predicted parity to directly detect tag parity errors. Instead, predicted tag parity is factored into the Pcache hit logic such that a Pcache miss will be forced if the tag parity does not agree with the parity calculated on the input address. By doing so, the tag parity design does not have to handle the case of a Pcache hit causing data to be returned to the Ibox, Ebox or Mbox in the presence of a Pcache tag parity error. Pcache predicted tag parity works by generating parity on M%S6_PA_H<31:12> at the same time as the Pcache access is taking place. If a validated tag matches the address on M%S6_PA_H<31:12>, but the tag parity does not match the predicted parity, a Pcache miss is forced.

The second aspect of Pcache tag parity error detection explicitly detects the tag error condition after the Pcache access has completed. Both banks of the tag store have their own tag parity generator. When both tags of the addressed Pcache index are driven out of the tag store, the two parity generators calculate tag parity based on the two accessed tags. These calculated values are compared to the corresponding stored tag parity which was accessed from the tag store with the tag data. If a miscompare occurs, a tag parity error is flagged. Note that this mechanism allows miscomparing tags to be flagged as tag parity errors while the other tag may simultaneously generate a Pcache hit or miss.

Pcache tag parity is checked on both tags on all Pcache I-stream read operations only when I_ENABLE=1 and FORCE_HIT=0 in the PCCTL. Pcache tag parity is checked on both tags on all Pcache D-stream read and write operations only when D_ENABLE=1 and FORCE_HIT=0 in the PCCTL. When FORCE_HIT=1, tag parity is never checked. Pcache tag parity is never checked on an IPR_RD operation to a Pcache tag. Tag parity is written on a cache fill operation or on an IPR_WR to a Pcache tag.

12.6.3.2 Pcache data parity error detection

Byte parity is maintained for each Pcache hexaword block. Therefore, each block contains 32 bits of parity–one bit of even parity for each byte of data.

Pcache data parity is checked on the same conditions as Pcache tag parity checks except for two differences:

1. Unlike tag parity, Pcache data parity errors are only detected during a Pcache hit condition. One exception to this rules exists though. If the Pcache force hit condition exists due to a memory management fault or hard fault, then Pcache data parity is not checked in spite of the Pcache hit condition.
2. Unlike tag parity, data parity is written into the array during a Pcache write operation rather than checked. **M%S6_BYTE_MASK_H<7:0>** enables writing data parity into the Pcache in the same manner as **M%S6_BYTE_MASK_H<7:0>** enables writing data into the Pcache. Therefore, each data parity bit is only updated as its corresponding byte of data is updated in the Pcache array.

The Pcache data parity check begins following the completion of the Pcache read access. Correct parity is generated on all eight data bytes read out of the Pcache. Each bit of generated data parity is compared to its corresponding stored parity. If one or more mismatches is found, a Pcache data parity error has occurred. Note that the parity check is independent of which bytes of the eight accessed bytes were actually requested by the read reference. Therefore, a Pcache data parity error can occur even though the requested bytes of data have correct parity.

12.6.4 Recording Mbox errors

When any hard error is detected within the system, the error is recorded in one of many error status registers located throughout the NVAX system. When the operating system error handler routine is invoked from a microtrap or interrupt, the handler can read the state of all the error registers through IPR_RD operations to determine what error or errors were present when the error handler was invoked.

The Mbox contains four of these error registers. Two are used to record TB parity errors and the other two are used to record Pcache parity errors.

12.6.4.1 TBSTS and TBADR

The TB status register is shown below:

Figure 12–56: IPR ED (hex), TBSTS

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  SRC   | 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0|     CMD      |  |  |  |  |:TBSTS
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                                                      |  |  |  |
                                                                      EM_VAL----------+  |  |  |
                                                                      TPERR--------------+  |  |
                                                                      DPERR-----------------+  |
                                                                      LOCK---------------------+
```

Table 12–23: TBSTS Field Descriptions

Name	Extent	Type	Description
LOCK	0	WC	Lock Bit. When set, validates TBSTS contents and prevents any other field from further modification. When clear, indicates that no TB parity error has been recorded and allows TBSTS and TBADR to be updated.
DPERR	1	RO	Data Error Bit. When set, indicates a TB data parity error.
TPERR	2	RO	Tag Error Bit. When set, indicates a TB tag parity error.
EM_VAL	3	RO	EM_LATCH valid bit. Indicates if EM_LATCH was valid at the time of the error TB parity error detection. This helps the software error handler determine if a write operation may have been lost due to the TB parity error.
CMD	8:4	RO	S5 command corresponding to TB parity error.
SRC	31:29	RO	Indicates the original source of the reference causing TB parity error.

Table 12–24: SRC Encodings

Defined SRC values	Definition
110	valid IREAD error is stored
100	valid Ibox specifier reference error is stored
000	valid Ebox reference error is stored

See Figure 12–27 for the format description of TBADR.

When a TB parity error is detected with LOCK=0, TBADR is loaded with the virtual address which caused the TB parity error, and all fields of TBSTS are updated to record the nature of the TB parity error. Note that both the TPERR and DPERR bits can be set at the same time if these two error conditions occurred during the same cycle. When a TB parity error is recorded, the LOCK bit is set to validate the contents of both TBSTS and TBADR registers. When LOCK is set, all bits of both registers are frozen and cannot be changed until the LOCK bit is cleared. Thus, any subsequent error is not recorded if LOCK=1.

When the operating system error handler is invoked, TBSTS and TBADR will be read through an IPR_RD command in order to determine if any TB parity errors were recorded. If the state of the LOCK bit was read to be a zero, then no error has occurred and the remaining state information in these two registers is invalid. If the LOCK bit was found to be set, then the remaining error state of these two registers characterizes the nature of the recorded error.

Once the error handler has read these registers, it re-enables TBSTS to record any new errors by clearing the LOCK bit. Clearing the LOCK bit is accomplished by writing a "1" to LOCK through an IPR_WR operation.

12.6.4.2 PCSTS and PCADR

The PCSTS register is shown below:

Figure 12–57: IPR F4 (hex), PCSTS

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1| 1|   |   |        CMD        |   |   |   |   |:PCSTS
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                                                                                      |   |                       |   |   |   |
                                                                PTE_ER----------------+   |                       |   |   |   |
                                                                PTE_ER_WR-----------------+                       |   |   |   |
                                                                LEFT_BANK-----------------------------------------+   |   |   |
                                                                RIGHT_BANK--------------------------------------------+   |   |
                                                                DPERR-----------------------------------------------------+   |
                                                                LOCK----------------------------------------------------------+
```

Table 12–25: PCSTS Field Descriptions

Name	Extent	Type	Description
LOCK	0	WC	Lock Bit. When set, validates PCSTS<8:1> contents and prevents modification of these fields. When clear, invalidates PCSTS<8:1> and allows these fields and PCADR to be updated.
DPERR	1	RO	Data Error Bit. When set, indicates a Pcache data parity error.
RIGHT_BANK	2	RO	Right Bank Tag Error Bit. When set, indicates a Pcache tag parity error on the right bank.
LEFT_BANK	3	RO	Left Bank Tag Error Bit. When set, indicates a Pcache tag parity error on the left bank.
CMD	8:4	RO	S6 command corresponding to Pcache parity error.
PTE_ER_WR	9	WC	Indicates a hard error on a PTE DREAD which resulted from a TB miss on a WRITE.
PTE_ER	10	WC	Indicates a hard error on a PTE DREAD.

The PCSTS and PCADR record Pcache tag and data parity errors. The function and operation of these registers is identical to the TBSTS and TBADR registers except that the PCADR stores physical quadword addresses rather than virtual byte addresses, and it also records PTE hard error events. The definitions of these registers are shown in Figure 12–29 and Figure 12–30.

Note however, that when PCSTS<0> is set, Pcache memory reads, writes and invalidates are disabled.

The PCSTS is a partial misnomer in that it also records hard error state associated with fatal errors occurring on Mbox PTE DREAD references. These hard errors have nothing to do with Pcache parity errors, however, they are included in PCSTS for implementation simplicity.

The PTE_ER bit of PCSTS will set whenever the Cbox has returned fatal error status on a requested PTE DREAD. The PTE_ER_WR bit of PCSTS will set whenever the Cbox has returned fatal error status on a requested PTE DREAD which was due to a TB miss on a WRITE reference. Both of these bits may be set independently of the LOCK bit of PCSTS. Further, the state of these bits are always valid regardless of the state of the LOCK bit. These two bits can only be cleared by a write-one-to-clear operation to each bit.

12.6.5 Mbox Error Processing

12.6.5.1 Processing TB parity errors

TB tag parity errors can be detected on all commands which cause a TB tag lookup to occur (See Section 12.6.5.4). TB data parity errors can be detected on all commands in which data can be read out of the TB (See Section 12.6.5.4).

For hardware simplicity, the detection of any TB parity error will cause the Mbox to generate a hard error microtrap and will cause the faulting reference and all pending Ibox, Ebox and Mbox references to be cleared. Thus, any TB parity error is fatal in the sense that it is non-recoverable and will cause a machine check.

The following describes the specific sequence of events which occur following the detection of a TB tag parity error, or a TB data parity error:

1. If the TBSTS register is locked, TBSTS state is not updated. Assuming the TBSTS is not locked, the TB parity condition is recorded in the TBSTS and the associated virtual address is loaded into TBADR. TBSTS and TBADR are subsequently locked by setting TBSTS<0>.

   The Mbox asserts M%TB_PERR_TRAP_L to invoke a hard error microtrap.

   The valid bits of the IREF_LATCH, SPEC_QUEUE, EM_LATCH, VAP_LATCH, and RTY_DMISS_LATCH are unconditionally cleared to eliminate all pending references which might involve a subsequent TB operation.
2. The TB parity error detection causes the MME_DATAPATH to invoke the TB parity error sequence. As a result, the MME_DATAPATH issues a TBIA command.

   The reference which caused the TB parity error is transformed into a NOP command as it propagates into the S6 pipe. Thus, this reference will not modify any Pcache, Bcache or Cbox state.
3. The TBIA command executes in S5 causing all TB entries to be invalidated and for the NLU pointer to be reset. All TB entries are invalidated rather than just the one which caused the parity error. This is done based on the premise that a single soft failure in the TB may affect more than one entry. Thus, each distinct soft failure will only be detected and reported once.

12.6.5.2 Processing Pcache parity errors

Pcache tag parity errors can be detected on all commands which cause a Pcache tag lookup to occur (See Section 12.6.5.4). Pcache data parity errors can be detected on all commands in which data is read out of the Pcache (See Section 12.6.5.4).

The strategy behind processing Pcache parity errors is to turn off the Pcache and let the Cbox process the reference from the Bcache or from main memory. Thus, in the absence of any of errors from the Cbox or memory subsystem, a Pcache parity error never causes an error fatal to the currently executing process.

The following describes the specific sequence of events which occur following the detection of a PCACHE tag parity error:

1. The Pcache tag parity error is recorded in it and the corresponding physical address is recorded in PCADR. PCADR and PCSTS are subsequently locked by setting the LOCK bit of PCSTS. Locking PCSTS automatically disables the Pcache from performing any subsequent non-IPR operations.

   The Mbox asserts **M%MBOX_S_ERROR_H** to flag an interrupt which will guarantee that the parity error will be recorded as a soft error at some future time.

   If the Pcache operation is a write, the Cbox will automatically continue processing the reference independent of any parity error condition. In the case of read operations, the predicted parity mechanism guarantees that a Pcache miss condition will occur when a tag parity error is detected. Thus, **M%CBOX_REF_ENABLE_L** is asserted in response to the Pcache miss condition causing the Cbox to continue to process the read reference.

The following describes the specific sequence of events which occur following the detection of a PCACHE data parity error:

1. The Pcache data parity error is recorded in it and the corresponding physical address is recorded in PCADR. PCADR and PCSTS are subsequently locked by setting the LOCK bit of PCSTS. Locking PCSTS automatically disables the Pcache from performing any subsequent non-IPR operations.

   The Mbox asserts **M%MBOX_S_ERROR_H** to flag an interrupt which will guarantee that the parity error will be recorded as a soft error at some future time.

   If the Pcache operation was a read in the absence of an outstanding fill operation, then **M%CBOX_LATE_EN_H** is asserted to inform the Cbox that it must continue to process the S6 reference because of the Pcache data parity error. **M%CBOX_LATE_EN_H** may be asserted in spite of the fact that **M%CBOX_REF_ENABLE_L** was deasserted earlier in the cycle because **M%CBOX_REF_ENABLE_L** is dependent on the Pcache hit condition but not on the parity error detection. The Pcache read reference is loaded into the corresponding MISS_LATCH and the read is treated in subsequent cycles as a normal Pcache miss sequence.

   If the Pcache operation was a D-stream read which occurred during an outstanding fill operation, **M%CBOX_LATE_EN_H** is not asserted because the Mbox and Cbox are unable to handle another fill at this point. When the the fill sequence completes, this reference will be retried (from the RTY_DMISS_LATCH), and **M%CBOX_LATE_EN_H** will be issued.

   Note that **M%CBOX_LATE_EN_H** is never asserted during a Pcache write operation.

12.6.5.3 Processing Cbox errors on Mbox-initiated read-like sequences

The Cbox detects errors that occur in the Bcache, NDAL or memory subsystem. When the Cbox detects one of these errors, and it is associated with an Mbox-initiated reference that requires data to be returned (e.g. memory read, I/O space read, or IPR read), the Mbox must transfer the error status of the reference back to the destination corresponding to the reference. The Mbox never records a Cbox-detected error in Mbox error registers because the error is logged in Cbox error registers.

12.6.5.3.1 Cbox-detected ECC errors

The Cbox returns requested data through a I_CF or D_CF command to the Mbox while simultaneously checking the error-correction code to check for a possible Bcache error. If an ECC error is found, the Cbox asserts **C%CBOX_ECC_ERR_H**. This causes the Mbox to latch a NOP in the CBOX_LATCH rather than the cache fill. As a result, the Mbox does not perform any Pcache state updates resulting from the bad data nor does it assert **M%VIC_DATA_L**, **M%IBOX_DATA_L**, **M%EBOX_DATA_H**, or **M%MBOX_DATA** to indicate the presence of valid data.

During subsequent cycles, the Cbox will determine if the ECC error is correctable or not. If it is, the data will be corrected and returned. If the data is not correctable, a Cbox-detected hard error has occurred and will be dealt with as described below.

Note that the ECC detection mechanism is what verifies the validity of the data. The Cbox does not send any parity information in order for the Mbox to check the validity of the received data.

12.6.5.3.2 Cbox-detected hard errors on requested fill data

If the Cbox has determined that the requested data cannot be returned for some reason, the Cbox drives a cache fill command qualified by **C%CBOX_HARD_ERR_H**. When this happens, the Mbox performs the following actions:

1. The assertion of **C%CBOX_HARD_ERR_H** indicates to the Mbox that the cache fill data is invalid. Thus, the Mbox returns the invalid data on the **M%MD_BUS_H** in the same manner that all data is returned except that the data is further qualified by **M%HARD_ERR_H**. **M%HARD_ERR_H** informs the receiver that the data is invalid and that the requested data cannot be returned due to a hard error.
2. Once the Cbox detects a hard error on the requested data, the Cbox immediately terminates the pending fill sequence by the assertion of **C%LAST_FILL_H**. Thus, no further data corresponding to the same fill sequence will be returned and the Mbox fill sequence corresponding to the error is terminated by invalidating the corresponding MISS_LATCH.
3. An I_CF or D_CF command which is qualified by **C%CBOX_HARD_ERR_H** is interpreted by the Pcache as an INVAL command. Thus the invalid data is not filled in the Pcache.

12.6.5.3.3 Cbox-detected hard errors on non-requested fill data

The Cbox performs the same actions as described above to indicate the presence of a hard error regardless of whether the data is the requested data or just one of the other three pieces of fill data for the corresponding Pcache block. If the data is non-requested fill data, the Mbox performs the following actions:

1. Once the Cbox detects a hard error on the non-requested data, the Cbox immediately terminates the pending fill sequence by the assertion of **C%LAST_FILL_H**. Thus, no further data corresponding to the same fill sequence will be returned and the Mbox fill sequence corresponding to the error is terminated by invalidating the corresponding MISS_LATCH.
2. An I_CF or D_CF command which is qualified by **C%CBOX_HARD_ERR_H** is interpreted by the Pcache as an INVAL command. Thus the invalid fill data is not filled in the Pcache and all previous fills to the same block are invalidated. This is necessary in order to maintain coherency between the Pcache and Bcache because a Bcache data block will only be validated if all the data within the block is error-free.

12.6.5.3.4 Microcode Invocation on Cbox-detected Hard Errors

When the Cbox indicates a hard error on requested read data, invalid data is driven on the **M%MD_BUS_H** qualified by **M%HARD_ERR_H** to indicate that the data is invalid due to a hard error. When the Ebox references the corresponding data a microtrap is generated by the Ebox to invoke the hard error microflow.

If the hard error occurs on the address of the address of an operand (i.e. Ibox decoding a deferred specifier), the Mbox returns data qualified by **M%HARD_ERR_H** in the normal manner. However, in some instances, the Ibox must issue a second reference to the Mbox based on the address returned by the first reference. Due to the hard error however, the Ibox cannot issue a valid operand read since the data returned by the first reference was invalid. In this case, the Ibox issues a read qualified with the **I%FORCE_HARD_FAULT_H** signal.

If this deferred specifier is a source operand, the Mbox "fakes" a hard error on this read by forcing a Pcache hit and by qualifying the returned data with **M%HARD_ERR_H**. This reference is trapped on when the Ebox references the operand.

If this deferred specifier is a destination specifier, the Mbox sets the corresponding hard error bit in the the PA_QUEUE. The hard error condition is then propagated to the Ebox through **M%PA_Q_STATUS_H<2>**.

If a hard error is generated on an Mbox PTE reference, this fact is recorded in the PCSTS register (see Section 12.6.4.2), the tb_miss sequence is immediately terminated, and the original reference (i.e. the reference that caused the memory management sequence which generated the PTE reference) is tagged as having the hard error associated with it.

When the original reference is retried after the memory management sequence completes, the reference will be treated as if the hard error actually occurred on it.

If the original reference was a read from the Ibox, the Mbox asserts **M%HARD_ERR_H** as it returns the invalid data to notify the Ibox or Ebox of the problem. The error handler will be invoked by the Ebox once the Ebox references the invalid data. The error handler will then read all error registers in the system to determine the nature of the error (note that the Cbox has recorded the physical PTE address of the fatal read).

Hard errors on PTE DREADs resulting from a TB miss on a DEST_ADDR get reported through the **M%PA_Q_STATUS_H<2>** mechanism described above.

Thus, any hard error on a PTE reference invoked by an Ibox reference will always be reported within the context of the executing instruction. However, fatal errors on PTE DREADs resulting from MME_CHK and WRITE references pose a more difficult problem than PTE errors resulting from reads. Since both of these references do not cause the Ebox to wait for a response from the Mbox, a more involved sequence is implemented in order to maximize the ability to report the fatal error within the context of the corresponding instruction execution.

Thus, when a PTE error is detected on ANY Ebox reference except for PROBEs, the following sequence will take place:

1. The Mbox will immediately assert **M%MME_TRAP_L** (unless the Ebox has previously asserted **E%FLUSH_MBOX_H** during the tb miss sequence).

   The MME sequencer will update MMEADR to record the original address of the reference which resulted in the tb miss sequence–it does not record the PTE address. The MME sequencer will update MMESTS<2> to indicate whether the original address had modify intent. The FAULT, PTE_REF, and LV fields of MMESTS are UNPREDICTABLE in this context.
2. The assertion of **M%MME_TRAP_L** will cause the Ebox to immediately trap to the mme microflow.
3. The mme microflow will examine MMESTS<2> and issue a PROBE command to the address in MMEADR to determine to nature of the mme fault.
4. The PROBE will invoke another TB miss. If the PTE error does not reoccur, valid PROBE status will be returned to the Ebox indicating the absence or presence of a true mme fault. In this case, Ebox processing of the current instruction will continue with no consequences due to the transient hard error.

   If the PTE error does reoccur on the TB miss during PROBE processing, the PROBE status returned to the Ebox will be qualified with **M%HARD_ERR_H** indicating that a fatal error occurred during the PROBE reference. This will invoke the error handler within the context of the executing instruction.

12.6.5.4 Mbox Error Processing Matrix

The following table summaries all Mbox error handling. A blank entry in the table means that the corresponding error cannot occur for the given reference.

Table 12–26: Mbox Error Handling Matrix

Command	TB tag parity error	TB data parity error	Pcache tag parity error	Pcache data parity error	Cbox hard error
Ibox references					
IREAD	A	A	B	D	F

Table 12–26 (Cont.): Mbox Error Handling Matrix

Command	TB tag parity error	TB data parity error	Pcache tag parity error	Pcache data parity error	Cbox hard error
DREAD	A	A	B	D	F
DREAD_MODIFY	A	A	B	D	F
DEST_ADDR	A	A			
STOP_SPEC_Q					
Ebox references					
DREAD	A	A	B	D	F
DREAD_LOCK	A	A	B		F
STORE			C		
WRITE	A	A	C		
WRITE_UNLOCK	A	A	C		
IPR_RD (to Pcache)					
IPR_RD (non-Mbox)					F
IPR_WR (to Pcache)					
IPR_WR (non-Mbox)					
PROBE	A	A			
MME_CHK	A	A			
TB_TAG_FILL					
TB_PTE_FILL					
TBIS					
TBIP					
TBIA					
LOAD_PC					
Mbox references					
PTE DREAD	A	A	B	D	G
TB_TAG_FILL					
TB_PTE_FILL	A				
IPR_DATA					

Table 12–26 (Cont.): Mbox Error Handling Matrix

Command	TB tag parity error	TB data parity error	Pcache tag parity error	Pcache data parity error	Cbox hard error
MME_CHK	A	A			
Cbox references					
INVAL			E		
D_CF					H
I_CF					H

LEGEND:

A.

- Mbox microtraps Ebox by assertion of **M%TB_PERR_TRAP_L** during cycle error was detected.
- The faulting reference and all pending Ibox and Ebox references are blown away.
- TBIA command is issued to invalidate entire TB.
- TBSTS and TBADR are updated appropriately.

B.

- A Pcache miss condition is forced to occur on this read reference causing the assertion of **M%CBOX_REF_ENABLE_L**. This instructs the Cbox to continue processing the read reference.
- **M%MBOX_S_ERROR_H** is asserted to post a soft error interrupt.
- PCSTS and PCADR are updated appropriately (a side effect of this operation turns off the Pcache).

C.

- The Cbox continues to process the write reference, as is done on all write operations regardless of a Pcache parity error.
- **M%MBOX_S_ERROR_H** is asserted to post a soft error interrupt.
- PCSTS and PCADR are updated appropriately (a side effect of this operation turns off the Pcache).

D.

- **M%CBOX_LATE_EN_H** is asserted to instruct the Cbox to continue processing the reference which caused the Pcache parity error.
- **M%MBOX_S_ERROR_H** is asserted to post a soft error interrupt.
- PCSTS and PCADR are updated appropriately (a side effect of this operation turns off the Pcache).

E.

- The invalidate operation takes place in spite of the tag parity error because the invalidate is only a function of matching all tag bits.
- M%MBOX_S_ERROR_H is asserted to post a soft error interrupt.
- PCSTS and PCADR are updated appropriately (a side effect of this operation turns off the Pcache).

F.

- The Cbox indicated a hard error for a non-PTE read or IPR_RD operation by the assertion of C%CBOX_HARD_ERR_H and C%LAST_FILL_H.
- If the hard error corresponded to the data explicitly requested by the Mbox reference, M%HARD_ERR_H qualifies M%MD_BUS_H data indicating to the M%MD_BUS_H receiver that a hard error occurred while accessing the requested data.
- The fill sequence is immediately terminated by the assertion of C%LAST_FILL_H. and the entire Pcache block corresponding to the fill is invalidated.

G.

- The hard error detected by the Cbox on this Mbox-issued PTE DREAD is recorded in PCSTS. The tb miss sequence is immediately terminated.

  IF the error resulted from an Ibox reference, the error is tagged back to the appropriate Ibox reference latch. The error is then signaled via M%HARD_ERR_H when the requested data is returned on M%MD_BUS_H, or is reported through PA_Q_STATUS<2> (for DEST_ADDR commands).

  If the original reference came from the Ebox, M%MME_TRAP_L is asserted (in all cases except for PROBE references). This will invoke the memory management fault handler in order to try to report the hard error within the context of the execution of the instruction (see Section 12.6.5.3.4 for more information).
- The fill sequence is immediately terminated by the assertion of C%LAST_FILL_H. and the entire Pcache block corresponding to the fill is invalidated.

H. C%CBOX_HARD_ERR_H was asserted by the Cbox during an I_CF or D_CF command. This is the mechanism by which the Cbox informs the Mbox of a hard error during a read or IPR_RD operation where the Cbox must return data. Thus, see the error responses specified by F and G for the error response within context of the original read operation.

12.7 MBOX INTERFACES

The Mbox passes data and/or control information to four other sections of the NVAX chip. These sections are: 1) Ibox, 2) Ebox, 3) Useq and 4) Cbox. This section will describe the interfaces to each of these sections.

12.7.1 IBOX INTERFACE

12.7.1.1 Signals from Ibox

- **I%IBOX_CMD_L**<4,1:0>: Command field of reference sent by Ibox.
- **I%IBOX_ADDR_H**<31:0>: Transfers addresses of Ibox references to Mbox.
- **I%IBOX_TAG_L**<2:0>: Ebox reg file destination of reference sent by Ibox.
- **I%IBOX_AT_L**<1:0>: Access type of reference sent by Ibox.
- **I%IBOX_DL_L**<1:0>: Data length of reference sent by Ibox.
- **I%IBOX_REF_DEST_L**<1:0>: Indicates the destination(s) of the requested Ibox reference.
-
- **I%IREF_REQ_H**: When asserted, indicates that a valid IREAD reference is present on the **I%IBOX_ADDR_H**<31:0> bus.
- **I%SPEC_REQ_H**: When asserted, indicates that a valid specifier reference is being issued to the Mbox.
- **I%FORCE_MME_FAULT_H**: Indicates that the associated Ibox reference should be forced to "look" like a memory management fault from the Ibox point of view.
- **I%FORCE_HARD_FAULT_H**: Indicates that the associated Ibox reference should be forced to "look" like a hardware fault from the Ibox point of view.
- **I%FLUSH_IREF_LAT_H**: Indicates that any current IREAD sequence in Mbox should be immediately cleared.

12.7.1.2 Signals to Ibox

- **M%SPEC_Q_FULL_H**: Informs Ibox that the SPEC_QUEUE is full and cannot accept any new references.
- **M%LAST_FILL_H**: Qualifies I_CF data being returned to Ibox. It indicates that this data is the last fill data for the current fill sequence.
-
- **M%MD_BUS_H**<63:0>: Transfers data back to Ibox.
- **M%MD_BUS_QW_PARITY_L**: Quadword parity for **M%MD_BUS_H**.
- **M%QW_ALIGNMENT_H**<1:0>: Indicates the relative aligned quadword position of VIC fill data within the aligned hexaword.
- **M%VIC_DATA_L**: When asserted, indicates that **M%MD_BUS_H**<63:0> contains VIC fill data.
- **M%IBOX_DATA_L**: When asserted, indicates that **M%MD_BUS_H**<31:0> contains requested Ibox data.
- **M%IBOX_IPR_WR_H**: When asserted, indicates that **M%MD_BUS_H**<31:0> contains Ibox IPR write data.

- **M%MME_FAULT_H**: When asserted in conjunction with **M%VIC_DATA_L** or **M%IBOX_DATA_L**, indicates that data on **M%MD_BUS_H** is invalid and that the corresponding reference was associated with a memory management exception.
- **M%HARD_ERR_H**: When asserted in conjunction with **M%VIC_DATA_L** or **M%IBOX_DATA_L**, indicates that data on **M%MD_BUS_H** is invalid and that the corresponding reference was associated with a hard error condition.

12.7.2 EBOX INTERFACE

12.7.2.1 Signals from Ebox

- **E%EBOX_CMD_H**<4:0>: Command field of reference sent by Ebox.
- **E%VA_BUS_L**<31:0>: Transfers addresses of Ebox references to Mbox.
- **E%WBUS_H**<31:0>: Transfers data of Ebox references to Mbox.
- **E%EBOX_TAG_H**<4:0>: Ebox reg file destination of reference sent by Ebox.
- **E%EBOX_AT_H**<1:0>: Access type of reference sent by Ebox.
- **E%EBOX_DL_H**<1:0>: Data length of reference sent by Ebox.
- **E%EBOX_VIRT_ADDR_H**: Indicates whether address is virtual or physical.
-
- **E%MMGT_MODE_H**<1:0>: Execution mode to be used for ACV checks on PROBE references.
- **E%CUR_MODE_H**<1:0>: Execution mode to be used for ACV checks on all non-PROBE references.
- **E%EREF_REQ_H**: When asserted, indicates that a valid Ebox reference is currently being issued.
- **E%EM_ABORT_L**: Indicates that the current EM_LATCH reference should be disregarded.
- **E%FLUSH_MBOX_H**: Indicates that certain references and reference state in the Mbox should be cleared (See Section 12.3.21.2).
- **E%FLUSH_PA_QUEUE_H**: Indicates that the PA_QUEUE should be flushed (See Section 12.3.21.2).
- **E%START_IBOX_IO_RD_H**: Indicates that the Ebox is md stalling on the corresponding SPEC_QUEUE read. If this SPEC_QUEUE read is an I/O space read and **E%START_IBOX_IO_RD_H** is not asserted, the read is aborted until it is asserted.
- **E%RESTART_SPEC_QUEUE_H**: Indicates that Ebox has sent all explicit writes for the current instruction to the Mbox and, therefore, causes the SPEC_Q_SYNC_CTR to be incremented.
- **E%NO_MME_CHECK_H**: Indicates that the corresponding EM_LATCH reference should not be tested for ACV or M=0 conditions.

12.7.2.2 Signals to Ebox

- **M%EM_LAT_FULL_H**: Indicates that EM_LATCH is currently full and cannot accept any new references.
- **M%PA_Q_STATUS_H**<2>: indicates that the corresponding address in the PA_QUEUE is associated with a hard error.
- **M%PA_Q_STATUS_H**<1>: indicates that the corresponding address in the PA_QUEUE is associated with a memory management exception.

- **M%PA_Q_STATUS_H**<0>: indicates that sufficient physical address data is present in the PA_QUEUE to initiate an Ebox STORE command.
- **M%MD_BUS_H**<31:0>: Transfers data back to Ebox.
- **M%MD_TAG_H**<4:0>: Ebox reg file destination of reference on **M%MD_BUS_H**<31:0>.
- **M%EBOX_DATA_H**: When asserted, indicates that **M%MD_BUS_H**<31:0> contains requested Ebox data.
- **M%MME_FAULT_H**: When asserted in conjunction with **M%EBOX_DATA_H**, indicates that data on **M%MD_BUS_H** is invalid and that the corresponding reference was associated with a memory management exception.
- **M%HARD_ERR_H**: When asserted in conjunction with **M%EBOX_DATA_H**, indicates that data on **M%MD_BUS_H** is invalid and that the corresponding reference was associated with a hard error condition.
- **M%PMUX0_H**: Mbox performance data signal (see Section 12.10).
- **M%PMUX1_H**: Mbox performance data signal (see Section 12.10).

12.7.3 INTERRUPT SECTION INTERFACE

12.7.3.1 Signals to Interrupt Section

- **M%MBOX_S_ERROR_H**: Indicates that the Mbox has logged a hard error in the PCSTS register and thus, is posting an interrupt.

12.7.4 USEQ INTERFACE

12.7.4.1 Signals to Useq

- **M%MME_TRAP_L**: Indicates to the Useq that a memory management exception is to be invoked.
- **M%TB_PERR_TRAP_L**: Indicates to the Useq that a tb parity error has been detected.

12.7.5 CBOX INTERFACE

12.7.5.1 Signals from Cbox

- **C%CBOX_CMD_H**<1:0>: Command field of Cbox reference sent to Mbox.
- **C%CBOX_ADDR_H**<31:5>: Hexaword address of Cbox reference sent to Mbox.
- **C%MBOX_FILL_QW_H**<4:3>: Indicates the aligned quadword within the aligned hexaword.
- **C%REQ_DQW_H**: Qualifies the current D_CF to indicate that this is the requested data.
- **B%S6_DATA_H**<63:0>: Data of Mbox reference seen by Cbox.
- **C%S6_DP_H**<7:0>: Even data parity corresponding to **B%S6_DATA_H**<63:0> during cache fill references.
-
- **C%LAST_FILL_H**: When asserted, indicates that this is the last fill sent for the current sequence.
- **C%CBOX_HARD_ERR_H**: When asserted when Cbox is driving data onto the **B%S6_DATA_H** Bus, it indicates that data on **M%MD_BUS_H** is associated with a non-recoverable hard error.

- **C%CBOX_ECC_ERR_H**: Indicates that an ECC error is associated with the Cbox data being returned.
- **C%WR_BUF_BACK_PRES_H**: Indicates that Cbox cannot accept any more entries in its write buffer.

12.7.5.2 Signals to Cbox

- **M%S6_CMD_H**<4:0>: Command field of Mbox reference seen by Cbox.
- **M%S6_PA_H**<31:3>: Quadword physical address of Mbox reference seen by Cbox.
- **M%C_S6_PA_H**<2:0>: Address within addressed quadword of Mbox reference seen by Cbox.
- **B%S6_DATA_H**<63:0>: Data of Mbox reference seen by Cbox.
- **M%S6_BYTE_MASK_H**<7:0>: Byte mask field of Mbox reference seen by Cbox.
- **M%CBOX_REF_ENABLE_L**: Indicates that current S6 read reference packet should be latched and processed by the Cbox. This signal is a don't care on write operations.
- **M%CBOX_LATE_EN_H**: Asserted at the end of a cycle to indicate that a Pcache parity error was detected. As a result, the Cbox must continue to process this reference regardless of what **M%CBOX_REF_ENABLE_L** indicated.
-
- **M%ABORT_CBOX_IRD_H**: Indicates that any IREAD which the Cbox may be processing should be immediately terminated.
- **M%CBOX_BYPASS_ENABLE_H**: Indicates that the Cbox may drive **B%S6_DATA_H**<63:0> during the following cycle in order to attempt a data bypass.

12.8 INITIALIZATION

12.8.1 Power-up Initialization

The signal, **K_M%RESET_L** is asserted during the power-up reset sequence. The following state is forced whenever **K_M%RESET_L** is asserted:

- EM_LATCH valid bit is cleared.
- VAP_LATCH valid bit is cleared.
- MME_LATCH valid bit is cleared.
- RTY_DMISS_LAT valid bit is cleared.
- DMISS_LATCH valid bit is cleared.
- MME state machine is forced to the home state.
- PCCTL<8:0> are cleared (this disables the Pcache).

The power-up reset sequence also causes the assertion of **E%FLUSH_MBOX**. **E%FLUSH_MBOX** will cause the following state to be forced within the context of the power-up sequence:

- The SPEC_QUEUE valid bits are cleared.
- The SPEC_Q_SYNC_CTR is reset to 0. Note that a subsequent **E%RESTART_SPEC_Q** signal is expected to enable SPEC_QUEUE arbitration.
- MMESTS<31:29> are cleared. This invalidates and unlocks the MMESTS register.

See Section Section 12.3.21.2 for a complete description of all state changes due to **E%FLUSH_MBOX**.

Once **E%FLUSH_MBOX** has been asserted, **E%FLUSH_PA_QUEUE** will be asserted during a subsequent cycle. **E%FLUSH_PA_QUEUE** will cause all PA_QUEUE valid bits to be cleared.

The power-up reset sequence also causes the assertion of **I%FLUSH_IREF_LAT**. **I%FLUSH_IREF_LAT** will cause the following state to be forced within the context of the power-up sequence:

- The IREF_LATCH valid bit is cleared.
- The IMISS_LATCH valid bit is cleared.

See Section Section 12.3.21.1 for a complete description of all state changes due to **I%FLUSH_IREF_LAT**.

12.8.2 Initialization by Microcode and Software

It is the responsibility of the power-up microcode to perform an IPR_WRITE operation to clear MAPEN before any virtual memory references are issued to the Mbox from either the Ebox or Ibox. Failure to clear MAPEN could result in UNDEFINED behavior prior to complete memory management state initialization.

PAMODE is also cleared by the power-up microcode via an IPR_WRITE command. If the system configuration requires a 32 bit program-visible physical address space, setting the PAMODE value via an IPR_WRITE must be done under very controlled conditions because writes to the PAMODE processor register affect both physical address generation and interpretation of PTEs. With the possible exception of certain diagnostic code, writes to the PAMODE processor register should not be performed while memory management is enabled. With memory management disabled, writes to the PAMODE processor register should not be performed unless the PC of the MTPR instruction which writes to the register is in one of the following (hex) address ranges:

00000000..1FFFFFFF
E0000000..FFFFFFFF

By restricting PC to one of these address ranges, changes to the PAMODE register do not cause the generated physical address to change in going from 30-bit mode to 32-bit mode, or vice versa. At powerup, microcode fetches the initial instruction from the boot ROM at address E0040000 (hex), which is in the second of the ranges shown above. Therefore, the console code in the boot ROM may write to the PAMODE processor register, and it is expected that this is the place where the PAMODE processor register will be initialized.

In uncontrolled conditions, writes to the PAMODE processor register can cause UNDEFINED results.

12.8.2.1 Pcache Initialization

The Pcache is disabled by the power-up initialization sequence. In order to enable the Pcache, the following sequential actions must be performed:

1. Pcache IPR_WRITE operations must be performed to each Pcache tag to write the tag field to a known state, set the tag parity bit to the corresponding value, and clear the subblock valid bits.
2. The lock bit in PCSTS must be cleared so that a locked PCSTS will not inhibit turning on the Pcache.
3. An IPR_WRITE to the PCCTL must be done to enable the Pcache in the desired operation mode. This step effectively turns the Pcache on.

Note that the data array need not be initialized because correct parity will be written into the data array whenever fill data is validated, and data parity is only checked on validated sub-blocks.

12.8.2.2 Memory Management Initialization

Memory management is disabled by MAPEN being cleared by the power-up microcode. Before memory management can be turned on, the following actions must be performed:

- The Ebox must issue a TBIA command to invalidate the TB and reset the NLU pointer to a known state. This is done as part of the microcode processing of an MTPR to MAPEN.
- The Ebox must write the appropriate values into the six memory base and length registers via IPR_WRITE commands.

Once this is done, the Ebox may turn on memory management by setting MAPEN through an IPR_WRITE command.

12.9 Mbox Testability Features

This section describes what testability features are made use of for Mbox testability, and what Mbox signals are used for each testability function. For a global understanding of NVAX testability, and for a detailed description of each testability strategy and hardware mechanism, the reader is referred to Chapter 19.

12.9.1 Internal Scan Register and Data Reducers

The following lists Mbox signals which are captured in the internal scan chain. The signals are listed in the order in which they are serially shifted out. Therefore, the first signal listed is the first signal shifted out. If a bus of signals is listed in the form signal<x:y>, y represents the first bit to be shifted out; x represents the last bit of the bus to be shifted out.

Captured Signal Name	Description
M_QUE_QU3%SQ_VAL0_LAST_H<>	cycle-delayed valid bit for 0th entry Spec Queue
M_QUE_QU3%SQ_VAL1_LAST_H<>	cycle-delayed valid bit for 1st entry Spec Queue
M_QUE_QU1%EM_VAL_LAST_H<>	cycle-delayed valid bit for EM_LATCH
M_QUE_QU5%PAQ_STATUS_P3_H<2:0>	Status bits for PA_QUEUE
M_QUE_QU5%MME_TRAP_P3_H<>	Memory Management Exception Trap signal
M_QUE_QU5%RTY_VAL_LAST_P2_H<>	cycle-delayed valid bit for RTY_DMISS_LATCH
M_S5C_TST%CBOX_REF_EN_P2_H<>	Indicates S6 read reference is for Cbox
M%CBOX_BYPASS_ENABLE_H<>	Enables bypassing of Cbox cache fill data
M_S5C_TST%EM_LAT_FULL_P2_H<>	Indicates EM_LATCH backpressure status to Ebox
M_S5C_TST%VAP_VAL_LAST_H<>	cycle-delayed valid bit for VAP_LATCH
M_QUE_QU3%IREF_VAL_LAST_H<>	cycle-delayed valid bit for IREF_LATCH
M_QUE_QU1%MME_VAL_LAST_H<>	cycle-delayed valid bit for MME_LATCH
M_QUE_S6L%S5_PA_L3_H<9:31>	samples S5_PA Bus
M_QUE_S6L%S5_PA_L2_H<0:8>	samples S5_PA Bus
M_QUE%S5_AT_H<1:0>	Access type for S5 reference
M_QUE%S5_TAG_H<4:0>	Ebox tag address for S5 reference
M_QUE%S5_DEST_H<1:0>	Box destination code for S5 reference
M_QUE%S5_CMD_H<4:0>	Command for S5 reference
M_QUE%S5_DL_H<1:0>	Data length for S5 reference
M_QUE%S5_QUAL_H<6:0>	Qualifier bits for S5 reference

Note that only **M_QUE%S5_PA_H**<31:0> contains a data reducer. Implementing a data reducer on this bus should provide coverage for the Mbox S5 pipe as well as coverage for the Ibox, Ebox and Cbox logic which issue references to the Mbox.

12.9.2 Nodes on Parallel Port

The following signals are observable via the Parallel Port:

— **M_QUE%S5_CMD_H**<4:0>

— Current Reference Source (3 encoded bits). The encodings are as follows:

Reference Source	Encoding
NOP or PA_QUEUE (when cmd = STORE)	000
IREF_LATCH	001
SPEC_QUEUE	010
EM_LATCH (when cmd ^= STORE)	011
VAP_LATCH (when cmd ^= STORE)	100
MME_LATCH	101
RTY_DMISS_LATCH	110
CBOX_LATCH	111

— **M_QUE_QU5%ABORT_P4_H**

— **M_MME_MMD%TB_MISS_L3_H**

— **M_PC_BSL%PCACHE_HIT_P4_H**

— MME state machine state bits (4 encoded bits). The encodings are as follows:

State Name	Encoding
home	0000
tb_miss_1	0001
tb_miss_2	0010
tb_miss_3	0011
tb_miss_4	0100
tb_miss_5	0101
doub_tb_miss_1	0110
doub_tb_miss_2	0111
doub_tb_miss_3	1000
doub_tb_miss_4	1001
mme_1	1010
mme_2	1011
ipr_rd_1_tb_per_2	1100
xpage_1	1101
tb_per_1	1110
undefined	1111

— MD_BUS Qualifiers (3 encoded bits). The encodings are as follows:

Event	Encoding
undefined	000
Ibox data	001
Ebox data	010
Ibox and Ebox data	011
VIC data	100
Ibox IPR data	101
undefined	110
Mbox data	111

— **M%MME_FAULT_H**

12.9.3 Nodes on Top Metal

tbd

12.9.4 Architectural features

The following is a brief description of all the Mbox architectural features which are relevant to verification, debug, and chip test. All of these features are invoked through the use of IPRs which are defined at the NVAX instruction set level. All of these IPRs can be invoked through the use of MTPR or MFPR macroinstructions. See the Architectural Summary Chapter for a list of all Mbox IPR addresses. Note that Mbox IPR addresses referenced through the MxPR instruction are translated by the Ebox microcode into IPR_RD, IPR_WR, TBIS, TBIA, or PROBE operations before being issued to the Mbox.

12.9.4.1 Translation Buffer Testability

The diagnostic user can invalidate the entire TB array by executing an MTPR instruction which addresses the TBIA IPR. This operation will also reset the NLU pointer. The user can invalidate any virtual page address which may cached in the TB by executing a MTPR addressing the TBIS IPR.

The diagnostic user can explicitly query the TB to determine if a given tag is validated and stored in the TB. This is accomplished by addressing the Translation Buffer Check IPR through the MTPR instruction.

Every TB entry can be explicitly filled and validated by the diagnostic user through the use of the TB_TAG_FILL and TB_PTE_FILL commands. The entry on which these two commands operate at any given time is addressed by the NLU pointer. The NLU pointer is a round robin pointer which increments when a TB_PTE_FILL is executed or when a tag match is detected on the entry which the NLU pointer is currently pointing to. The NLU pointer is reset to point to the 0th entry whenever a TBIA command is executed.

It is the responsbility of the diagnostic user to set his/her tests up such that normal I-stream and D-stream references generated in the macropipeline do not interfere with the TB state under test. Specifically, the user must guarantee that all relevant pages of the diagnostic program reside in the TB before the test begins, such that accessing these pages will not cause modification of the TB state while the diagnostic program is explicitly probing and changing TB state.

See Section 12.5.1.3 for a complete description of TB function as it relates to testability. See Section 12.3.11.2 for a description of the PROBE command which can be invoked through the Translation Buffer Check IPR.

12.9.4.2 Pcache Testability

Every bit in the Pcache can be read and written by the user through DREAD, WRITE, IPR_RD and IPR_WR operations. Pcache is accessed by DREADs and WRITEs. All other bits (tag, valid bits and parity bits) are accessed through Mbox IPRs.

The operational mode of the Pcache can be changed to accomodate testing the array. The mode is controlled by the Pcache Control Register (PCCTL) which can be read and written as an Mbox IPR. The PCCTL allows the user to:

1. Enable/disable D-stream and/or I-stream operations to the Pcache.
2. Allow the Pcache to operate in a direct mapped force hit mode.
3. Enable/disable Pcache parity checks.

See Section 12.4 for a complete description of Pcache function as it relates to testability.

12.9.5 M-BOX Miscellaneous Features

— tbd

12.10 Mbox Performance Monitor Hardware

Hardware exists in the Mbox to support the NVAX Performance Monitoring Facility. See Chapter 18 for a global description of this facility.

The Mbox hardware generates two signals, **M%PMUX0_H** and **M%PMUX1_H**, which are driven to the central performance monitoring hardware residing in the Ebox. These two signals are used to supply Mbox performance data for the purpose of recording performance statistics. Seven Mbox performance monitoring functions exist. The function to be executed is specified by the PMM field of the PCCTL register (see Figure 12–31).

The following describes the seven Mbox performance monitor modes:

Table 12–27: Mbox Performance Monitor Modes

PCCTL<7:5>	Performance Monitor Mode
000	TB hit rate for S0 Space I-stream Reads[1]
001	TB hit rate for S0 Space D-stream Reads[1]
010	TB hit rate for P0/P1 Space I-stream Reads[1]
011	TB hit rate for P0/P1 Space D-stream Reads[1]
100	Pcache hit rate for I-stream Reads
101	Pcache hit rate for D-stream Reads
110	illegal mode–Results are UNPREDICTABLE
111	ratio of unaligned virtual reads and virtual writes to total virtual reads and virtual writes

[1] TB hit count is unconditionally incremented when MAPEN=0

12.10.1 TB hit rate Performance Monitor Modes

The TB hit rate modes work by asserting **M%PMUX0_H** during the cycle in which a specific type of virtual read reference is first attempted in the S5 execution pipe. During the same cycle, **M%PMUX1_H** will transfer the TB hit status corresponding to this read execution event.

It is important to capture this data only on the first execution of the read in order that the TB hit statistics are not skewed by multiple retries of the same reference due to aborted cycles and tb_miss sequences.

One low probability scenario exists in which this scheme will not accurately record the TB hit/miss data for the reference. Consider the case where the read is initially executed and is found to hit in the TB while simultaneously being aborted due to some abort condition (e.g. Pcache Index Conflict). During the following cycle, another reference is executed which invokes a TB miss sequence. If the TB miss sequence displaces the PTE corresponding to the first read, then the read will subsequently be retried as a TB miss event even though it has already been recorded as a TB hit event. However, the frequency of this scenario should normally be so low that the accuracy of the TB hit ratio statistics will not be affected.

12.10.1.1 TB hit rate for P0/P1 I-stream Reads

In this mode, **M%PMUX0_H** is asserted during the cycle in which the IREF_LATCH first attempts to drive a virtual process space IREAD into the S5 pipe. Note that **M%PMUX0_H** is only asserted in response to IREAD execution events caused by Ibox-generated IREADs. This avoids recording Mbox-generated "fill forward" IREADs which would abnormally boost the TB hit rate. During the same cycle, **M%PMUX1_H** will transfer the TB hit status corresponding to the same IREAD execution event.

12.10.1.2 TB hit rate for P0/P1 D-stream Reads

In this mode, **M%PMUX0_H** is asserted during the cycle in which the SPEC_QUEUE, EM_LATCH, VAP_LATCH or MME_LATCH first attempts to drive a virtual process space read into the S5 pipe. During the same cycle, **M%PMUX1_H** will transfer the TB hit status corresponding to the same read execution event.

12.10.1.3 TB hit rate for S0 I-stream Reads

In this mode, **M%PMUX0_H** is asserted during the cycle in which the IREF_LATCH first attempts to drive a system space IREAD into the S5 pipe. Note that **M%PMUX0_H** is only asserted in response to IREAD execution events caused by Ibox-generated IREADs. This avoids recording Mbox-generated "fill forward" IREADs which would abnormally boost the TB hit rate. During the same cycle, **M%PMUX1_H** will transfer the TB hit status corresponding to the same IREAD execution event.

12.10.1.4 TB hit rate for S0 D-stream Reads

In this mode, **M%PMUX0_H** is asserted during the cycle in which the SPEC_QUEUE, EM_LATCH, VAP_LATCH or MME_LATCH first attempts to drive a virtual system space read into the S5 pipe. During the same cycle, **M%PMUX1_H** will transfer the TB hit status corresponding to the same read execution event.

12.10.2 Pcache hit rate Performance Monitor Modes

The Pcache hit rate modes work by asserting **M%PMUX0_H** during the cycle in which a specific type of S6 physical read reference is executed in the Pcache. During the same cycle, **M%PMUX1_H** will transfer the Pcache hit status corresponding to this read execution event.

12.10.2.1 Pcache hit rate for I-stream Reads

In this mode, **M%PMUX0_H** is asserted during the cycle in which an IREAD is executing in the S6 pipe. **M%PMUX0_H** is only asserted in response to IREAD execution events caused by Ibox-generated IREADs. This avoids recording Mbox-generated "fill forward" IREADs which would abnormally boost the Pcache hit rate. **M%PMUX1_H** will transfer the Pcache hit status corresponding to the same IREAD execution event during the cycle which **M%PMUX0_H** is asserted.

12.10.2.2 Pcache hit rate for D-stream Reads

In this mode, M%PMUX0_H is asserted during the cycle in which a D-stream read is executing in the S6 pipe. M%PMUX0_H is only asserted in response to the first Pcache lookup attempt of a D-stream read executing in the S6 pipe. This avoids skewing the performance data based on the same reference being retried in the Pcache due to the "read under fill" function. Therefore, S6 reads originating from the RTY_DMISS_LATCH do not cause the assertion of M%PMUX0_H. M%PMUX1_H will transfer the Pcache hit status corresponding to the same read execution event during the cycle which M%PMUX0_H is asserted.

12.10.3 Unaligned reference statistics

This mode allows the user to obtain the percentage of references processed by the Mbox which are unaligned.

In this mode, M%PMUX0_H is asserted on any virtual read, virtual DEST_ADDR, or virtual WRITE reference driven from the SPEC_QUEUE or EM_LATCH. The reference must virtual to be recorded due to the nature of the hardware implementation. M%PMUX1_H is asserted on the same conditions as M%PMUX0_H, except that it is further qualified by the fact that the reference is unaligned.

12.11 Mbox Signal Name Cross-Reference

All signal names referenced in this chapter have appeared in bold and reflect the actual name appearing in the NVAX schematic set. For each signal appearing in this chapter, the table below lists the corresponding name which exists in the behavioral model.

Table 12-28: Cross-reference of all names appearing in the Mbox chapter

Schematic Name	Behavioral Model Name
B%S6_DATA_H<63:0>	B%S6_DATA_H<63:0>
C%CBOX_CMD_H<1:0>	C%CBOX_CMD_H<1:0>
C%CBOX_ADDR_H<31:5>	C%CBOX_ADDR_H<31:5>
C%MBOX_FILL_QW_H<4:3>	C%MBOX_FILL_QW_H<4:3>
C%REQ_DQW_H<>	C%REQ_DQW_H
C%S6_DP_H<7:0>	C%S6_DP_H<7:0>
C%LAST_FILL_H	C%LAST_FILL_H
C%CBOX_HARD_ERR_H	C%CBOX_HARD_ERR_H
C%CBOX_ECC_ERR_H	C%CBOX_ECC_ERR_H
C%WR_BUF_BACK_PRES_H	C%WR_BUF_BACK_PRES_H
E%EBOX_CMD_H<4:0>	E%EBOX_CMD_H<4:0>
E%VA_BUS_L<31:0>	E%VA_BUS_H<31:0>
E%WBUS_H<31:0>	E%WBUS_H<31:0>
E%EBOX_TAG_H<4:0>	E%EBOX_TAG_H<4:0>
E%EBOX_AT_H<1:0>	E%EBOX_AT_H<1:0>
E%EBOX_DL_H<1:0>	E%EBOX_DL_H<1:0>
E%EBOX_VIRT_ADDR_H	E%EBOX_VIRT_ADDR_H
E%MMGT_MODE_H<1:0>	E%MMGT_MODE_H<1:0>
E%CUR_MODE_H<1:0>	E%CUR_MODE_H<1:0>
E%EREF_REQ_H	E%EREF_REQ_H
E%EM_ABORT_L	E%EM_ABORT_H
E%FLUSH_MBOX_H	E%FLUSH_MBOX_H
E%FLUSH_PA_QUEUE_H	E%FLUSH_PA_QUEUE_H
E%START_IBOX_IO_RD_H	E%START_IBOX_IO_RD_H
E%RESTART_SPEC_QUEUE_H	E%RESTART_SPEC_QUEUE_H
E%NO_MME_CHECK_H	E%NO_MME_CHECK_H
I%IBOX_CMD_L<4,1:0>	I%IBOX_CMD_H<4:0>
I%IBOX_ADDR_H<31:0>	I%IBOX_ADDR_H<31:0>
I%IBOX_TAG_L<2:0>	I%IBOX_TAG_H<2:0>
I%IBOX_AT_L<1:0>	I%IBOX_AT_H<1:0>

Table 12–28 (Cont.): Cross-reference of all names appearing in the Mbox chapter

Schematic Name	Behavioral Model Name
I%IBOX_DL_L<1:0>	I%IBOX_DL_H<1:0>
I%IBOX_REF_DEST_L<1:0>	I%IBOX_REF_DEST_H<1:0>
I%IREF_REQ_H	I%IREF_REQ_H
I%SPEC_REQ_H	I%SPEC_REQ_H
I%FORCE_MME_FAULT_H	I%FORCE_MME_FAULT_H
I%FORCE_HARD_FAULT_H	I%FORCE_HARD_FAULT_H
I%FLUSH_IREF_LAT_H	I%FLUSH_IREF_LAT_H
M%ABORT_CBOX_IRD_H	M%ABORT_CBOX_IRD_H
M%C_S6_PA_H<2:0>	M%C_S6_PA_H<2:0>
M%CBOX_BYPASS_ENABLE_H	M%CBOX_BYPASS_ENABLE_H
M%CBOX_LATE_EN_H	M%CBOX_LATE_EN_H
M%CBOX_REF_ENABLE_L	M%CBOX_REF_ENABLE_H
M%EBOX_DATA_H	M%EBOX_DATA_H
M%EM_LAT_FULL_H	M%EM_LAT_FULL_H
M%HARD_ERR_H	M%HARD_ERR_H
M%IBOX_DATA_L	M%IBOX_DATA_H
M%IBOX_IPR_WR_H	M%IBOX_IPR_WR_H
M%LAST_FILL_H	M%LAST_FILL_H
M%MBOX_S_ERROR_H	M%MBOX_S_ERROR_H
M%MD_BUS_H<63:0>	M%MD_BUS_H<63:0>
M%MD_BUS_QW_PARITY_L	M%MD_BUS_QW_PARITY_H
M%MD_TAG_H<4:0>	M%MD_TAG_H<4:0>
M%MME_FAULT_H	M%MME_FAULT_H
M%PA_Q_STATUS_H<2:0>	M%PA_Q_STATUS_H<2:0>
M%PMUX0_H	M%PMUX0_H
M%PMUX1_H	M%PMUX1_H
M%QW_ALIGNMENT_H<1:0>	M%QW_ALIGNMENT_H<1:0>
M%SPEC_Q_FULL_H	M%SPEC_Q_FULL_H
M%S6_BYTE_MASK_H<7:0>	M%S6_BYTE_MASK_H<7:0>
M%S6_CMD_H<4:0>	M%S6_CMD_H<4:0>
M%S6_PA_H<31:0>	M%S6_PA_H<31:0>
M%VIC_DATA_L	M%VIC_DATA_H
M_QUE%S5_AT_H<1:0>	M_QUE%S5_AT_H<1:0>
M_QUE%S5_CMD_H<4:0>	M_QUE%S5_CMD_H<4:0>
M_QUE%S5_DATA_H<31:0>	M_QUE%S5_DATA_H<31:0>
M_QUE%S5_DEST_H<1:0>	M_QUE%S5_DEST_H<1:0>

Table 12–28 (Cont.): Cross-reference of all names appearing in the Mbox chapter

Schematic Name	Behavioral Model Name
M_QUE%S5_DL_H<1:0>	M_QUE%S5_DL_H<1:0>
M_QUE%S5_PA_H<31:0>	M_QUE%S5_PADP_H<31:0>
M_QUE%S5_QUAL_H<6:0>	M_QUE%S5_QUAL_H<6:0>
M_QUE%S5_TAG_H<4:0>	M_QUE%S5_TAG_H<4:0>
M_QUE%S5_VA_H<31:0>	M_QUE%S5_VA_H<31:0>
M_S5C_ABT%ABORT_L	M_S5C%ABORT_H

12.12 Revision History

Who	When	Description of change
Mike Uhler	12-Sep-1991	Correct TB selections in the PMM field of PCCTL
Bill Wheeler	26-Jul-1991	correct an inconsistency in spec
Bill Wheeler	21-May-1991	add an mbox ucode restriction
Bill Wheeler	26-Apr-1991	final tweaks
Bill Wheeler	25-Feb-1991	tweaked description of ucode biasing of mm regs
Bill Wheeler	22-Feb-1991	described ucode biasing of mm regs
Bill Wheeler	20-Feb-1991	Changed text to reflect expaned S0 space configuration
Bill Wheeler	20-Sep-1990	Other tweaks; add signal xref table
Bill Wheeler	8-May-1990	Other tweaks
Bill Wheeler	27-Feb-1990	Add perf monitor hardware. Other tweaks
Bill Wheeler	15-Jan-1990	Signal name change
Bill Wheeler	20-Nov-1989	Final Changes prior to review for Rev 1.0 Release
Bill Wheeler	23-Aug-1989	More Updates
Bill Wheeler	31-Jul-1989	Spec Update
Bill Wheeler	06-Mar-1989	For External Release
Bill Wheeler	30-Nov-1988	Initial Release

Chapter 13

The Cbox

13.1 Terminology

Term	Meaning
Error transition mode (ETM)	Mode where the backup cache only services CPU requests to blocks which are valid-owned. All other CPU requests, including those to valid-unowned blocks, are ignored by the backup cache and are forwarded to memory. The purpose is to use the cache as little as possible because of previously detected errors.
Cache coherence transaction	A transaction from the external system which interrogates the backup cache and may cause a block invalidate and/or a block writeback.
Deallocate	The actions necessary to allocate a new block because of a read miss or a write miss. A writeback is required if the block is valid-owned. An invalidate is required if the block is valid, whether owned or unowned. A cache coherency request which results in a hit also causes a deallocate.
Longword	4 bytes of data
Quadword	8 bytes of data
Hexaword	32 bytes of data

13.2 Functional Overview of the Cbox and Backup Cache

The Cbox is that section of the NVAX CPU chip which controls the backup cache and interfaces to the external bus. The Cbox includes the BIU functions for the NVAX CPU. The backup cache is a writeback cache. Cache tags and cache data are stored in off-chip static RAMs (off-the-shelf parts). The Cbox implements the control for the cache tags; control for the cache data; and control for the external pin bus, the NDAL.

The Mbox sends read requests and writes to the Cbox; the Cbox sends fills and invalidates to the Mbox. The Cbox ensures that the Pcache is a subset of the backup cache through invalidates.

The Cbox communicates with the memory subsystem (everything beyond the backup cache) via the NDAL. The Cbox generates reads and receives fills; it receives cache coherence transactions from the NDAL to which it responds with invalidates and writebacks, as appropriate.

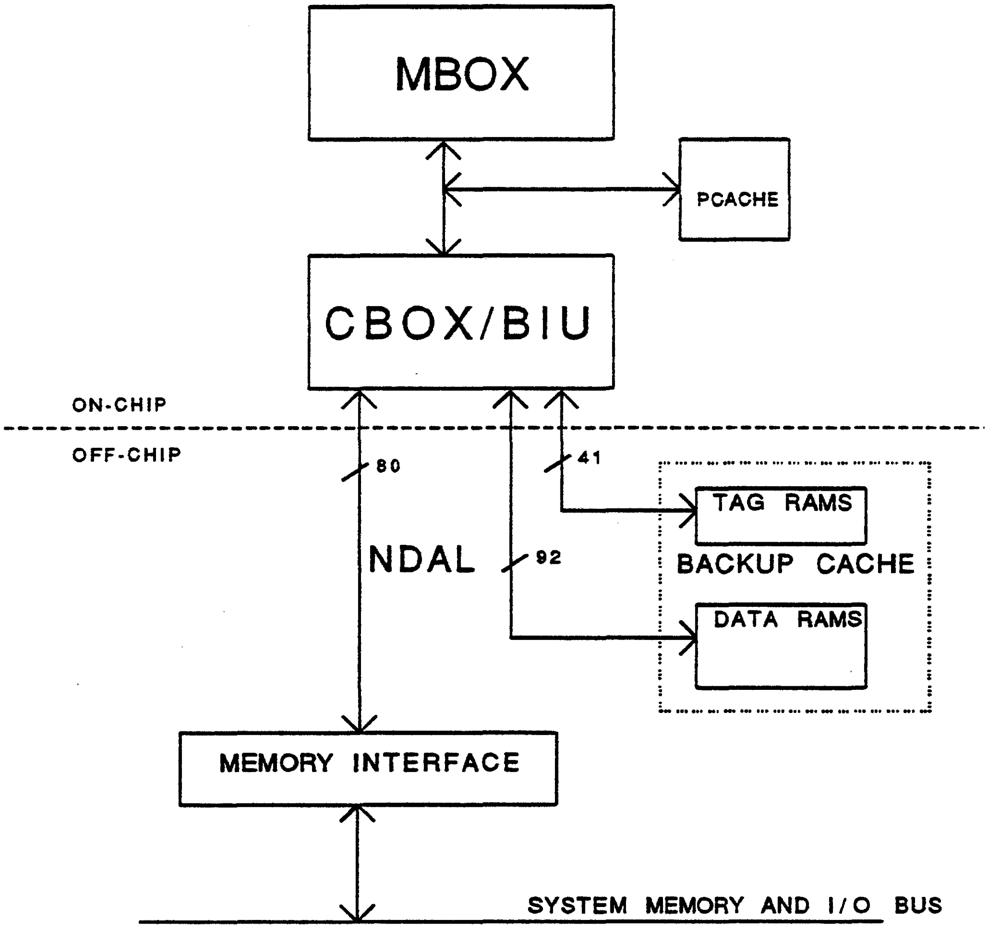
The reader is assumed to be familiar with Chapter 3, which describes the NDAL.

Cache coherence in an NVAX system is based upon the concept of ownership. A hexaword block of memory may be owned either by memory or by an NVAX backup cache. In a multiprocessor system, only one of the caches or memory can own the block at a time. Several of the planned NVAX systems implement an explicit ownership bit for each hexaword block of memory; it would also be possible to build an NVAX system without explicit ownership bits in memory.

13.2.1 The Cbox and the System

The Cbox has a tightly coupled internal interface with the Mbox. It has separate external busses which communicate with the backup cache tag RAMs, the backup cache data RAMs, and the memory interface, as shown in Figure 13–1.

Figure 13–1: The Cbox in the System



13.2.2 Writeback Cache and Ownership Concepts

There is one fundamental difference between a writeback cache and a writethrough cache. When a write is received by a write-through cache, the data may be written into the cache and is always written to memory as well. When a write is received by a writeback cache, the write is not necessarily forwarded to memory; the write may be done only into the cache. The data is written back to memory only if another element in the system needs that data, or if the block is displaced (deallocated) from the cache.

The NVAX backup cache is a writeback design in which a cache block may exist in one of three states: invalid, valid-unowned, and valid-owned. A block which is valid-unowned is a read-only copy of memory data. A block which is valid-owned may be written by NVAX, and if it has been written since being put into the cache, is the only up-to-date copy of the data in the system. The NVAX cache makes no distinction between valid-owned blocks it has written and those which it has not written.

A valid-unowned copy of a given cache block may reside in one or more backup caches in an NVAX multiprocessor system. No NVAX backup cache may contain a valid cache block which is valid-owned by another backup cache in the system. The Cbox design relies upon the system bus and/or the system bus interface to support NDAL Ownership Read/Disown Write pairs to ensure cache coherency.

The most straightforward way to implement a memory for NVAX is to have an ownership bit associated with each hexaword of data. When this memory receives an Ownership Read (OREAD) for a hexaword, ownership is passed to the requesting CPU, and the data is returned to the CPU. If another Ownership Read arrives for that hexaword from a second CPU, memory does not return the data since the hexaword is not owned by memory but by the first CPU. The first CPU recognizes the second OREAD as a cache coherence transaction and writes back the data from its cache, using the Disown Write command. The data is then available for the second CPU.

During normal operation, the Cbox issues an OREAD to the memory interface and receives ownership of the block before it performs a write to that block in the backup cache. The Cbox relinquishes ownership of the data when a cache coherence transaction requesting a writeback appears on the NDAL.

13.2.3 Backup Cache Operating Modes

The backup cache has four distinct modes of operation.

- Cache ON. Normal operation. Most of this chapter describes Cbox operation when the backup cache is on.
- Cache OFF. Reset puts the backup cache into the OFF state. The backup cache may be enabled/disabled (turned ON/OFF) by software through the Cbox control IPR. Cache off mode is described in Section 13.9.1.
- Force Hit. The Cbox forces all memory space reads and writes to hit in the backup cache. This mode is used for testing and initialization purposes. Force Hit mode is described in Section 13.9.2.
- Error Transition Mode. The Cbox enters Error Transition Mode upon recognition of some error conditions or when put into ETM explicitly by an IPR write. Error Transition Mode is described in Section 13.9.3.

13.3 NVAX Backup Cache Organization and Interface

The backup cache is configurable based on the size and speed of the cache RAMs used to implement the cache on the board.

The backup cache may be configured to be one of four sizes: 128 kilobytes, 256 kilobytes, 512 kilobytes, or 2 megabytes. This is controlled by the SIZE field in the CCTL register, as described in Section 13.5.1. The smallest RAMs which may be used to achieve each configuration are shown in Table 13–1.

Table 13–1: Backup Cache Size and RAMs Used

Cache size	Tag RAM Size	Data RAM Size	Number of Tags	Valid Bits Per Tag
128 Kilobytes	4K x 4	16K x 4	4K	1
256 Kilobytes	8K x 8[1]	32K x 8[1]	8K	1
512 Kilobytes	16K x 4	64K x 4	16K	1
2 Megabytes	64K x 4	256K x 4	64K	1

[1] Using x8 parts means the cache no longer takes advantage of the nibble protection feature of the cache ECC design.

Regardless of configuration, the cache has a block size of 32 bytes and has no subblocks. The data bus to the cache is 8 bytes wide, so in order to read out an entire block, 4 accesses are done. Each block contains 32 bytes of data and has associated with it a tag, a valid bit, and an owned bit. ECC protection is provided on each quadword in the cache. ECC protection is also provided on the tag store.

Each of address bits <20:17> serves either as an index bit or as a tag bit, based on the cache size configured. Table 13–2 shows how the bits are used.

Table 13–2: Tag and Index Interpretation based on cache size

Cache size	Tag bits used	Index bits used
128 kilobytes	Tag<31:17>	Index<16:5>
256 kilobytes	Tag<31:18>	Index<17:5>
512 kilobytes	Tag<31:19>	Index<18:5>
2 megabytes	Tag<31:21>	Index<20:5>

The backup cache speed may also be configured based on the access time of the RAMs used to implement the tag store and the data store. The TAG_SPEED and DATA_SPEED fields of the Cbox control register, CCTL, are used to control the number of NVAX cycles used by the Cbox to access the RAMs. The relationship between TAG_SPEED, DATA_SPEED, NVAX cycle time, and the cache RAM access times required is shown in Table 13–3.

NOTE

Table 13–3 is based upon simulations of the XNP (XMI-based system) board. These numbers may only be applied directly to an environment which is very close to that of the XNP.

Table 13–3: Backup Cache RAM Speeds and NVAX Cycle Time

	Tag RAM	tag read			Data RAM		
CCTL	access	(access)	tag write	CCTL	access	data read	data write
TAG_SPEED	time	rep rate	rep rate	DATA_SPEED	time	rep rate	rep rate
RAM Speeds required for 16 ns NVAX cycle time							
0	0 - 21ns	(2) 3 cycles	3 cycles	00[1]	0 - 19.5ns	2 cycles	3 cycles
1[1]	22 - 37ns	(3)4 cycles	4 cycles	01	20 - 35.5ns	3 cycles	4 cycles
				10	36 - 51.5ns	4 cycles	5 cycles
RAM Speeds required for 14 ns NVAX cycle time							
0	0 - 17.5ns	(2) 3 cycles	3 cycles	00[1]	0 - 16 ns	2 cycles	3 cycles
1[1]	18 - 31.5ns	(3)4 cycles	4 cycles	01	17 - 30 ns	3 cycles	4 cycles
				10	31 - 44 ns	4 cycles	5 cycles
RAM Speeds required for 12 ns NVAX cycle time							
0	0 - 14ns	(2) 3 cycles	3 cycles	00[1]	0 - 13 ns	2 cycles	3 cycles
1[1]	15 - 26ns	(3)4 cycles	4 cycles	01	14 - 25 ns	3 cycles	4 cycles
				10	26 - 37 ns	4 cycles	5 cycles
RAM Speeds required for 10 ns NVAX cycle time							
0	0 - 10.5ns	(2) 3 cycles	3 cycles	00[1]	0 - 9.5 ns	2 cycles	3 cycles
1[1]	11 - 20.5ns	(3)4 cycles	4 cycles	01	10 - 19.5 ns	3 cycles	4 cycles
				10	20 - 29.5 ns	4 cycles	5 cycles

[1] TAG_SPEED=1 cannot be used with DATA_SPEED=00, as the NVAX Cbox cannot function with tag rams whose read access time is longer than the data ram read access time.

Extensive simulations of the NVAX chip, package, and XNP board were done in order to determine the drive times of the cache pins in this environment. The drive times are measured from the internal NVAX clock to the signal being valid at the cache pin. The drive times for TT (typical speed) parts under worst-case conditions are shown in Table 13–4. These drive times would be met under worst-case conditions in the 14ns system. These drive times only apply to the XNP board, and cache drive times and performance would be different in a different environment.

Table 13–4: Cache pin drive times in the XNP environment

NVAX Cache Interface Pin	Starting clock	Time to signal valid at cache RAM
P%TS_TAG_H<31:17>, P%TS_ECC_H<5:0>, P%TS_OWNED_H, P%TS_VALID_H	K_PAD%PHI_4_H	8.5 ns
P%TS_TAG_H<31:17>, P%TS_ECC_H<5:0>, P%TS_OWNED_H, P%TS_VALID_H	K_PAD%PHI_4_H	1.5 ns (tristate time)
P%TS_INDEX_H<10:0>	K_PAD%PHI_3_H	8.0 ns
P%TS_INDEX_H<20:11>	K_PADL%PHI_3_H	8.0 ns
P%TS_OE_L	K_PADL%PHI_1_H (assertion), K_PADL%PHI_4_H (deassertion)	8.0 ns
P%TS_WE_L	K_PADL%PHI_3_H (assertion), K_PADL%PHI_1_H (deassertion)	8.0 ns
P%DR_INDEX_H<20:3>	K_PADL%PHI_3_H	8.0 ns
P%DR_OE_L	K_PADL%PHI_1_H (assertion), K_PADL%PHI_4_H (deassertion)	8.0 ns
P%DR_WE_L	K_PADL%PHI_3_H (assertion), K_PADL%PHI_1_H (deassertion)	8.0 ns
P%DR_DATA_H<63:0>, P%DR_ECC_H<7:0>	K_PADL%PHI_4_H	8.5 ns
P%DR_DATA_H<63:0>, P%DR_ECC_H<7:0>	K_PADL%PHI_4_H	1.5 ns (tristate time)

Figure 13–2 and Figure 13–3 show the timing of cache tag transactions and of cache data transactions. The symbols shown in the timing diagrams are defined in Table 13–5.

Table 13–5: Cache pin timing symbol definitions

Symbol	Meaning
Taa	RAM address access time: valid index to RAM output valid
Toe	Assertion of output enable to RAM output valid
Toh	RAM output hold from address change
Tohz	Output disable to RAM output in high Z
Taw	Valid index to end of RAM write
Tdw	Data valid to end of RAM write
Tnz	NVAX tristate time
Twr	Write enable deassert to address change (write recovery)
Tdh	NVAX data hold time after write enable deassert
Twp	Write enable pulse width
Tas	RAM address setup time to write enable assertion

Figure 13–2: Backup Cache Tag RAM Pin Timing

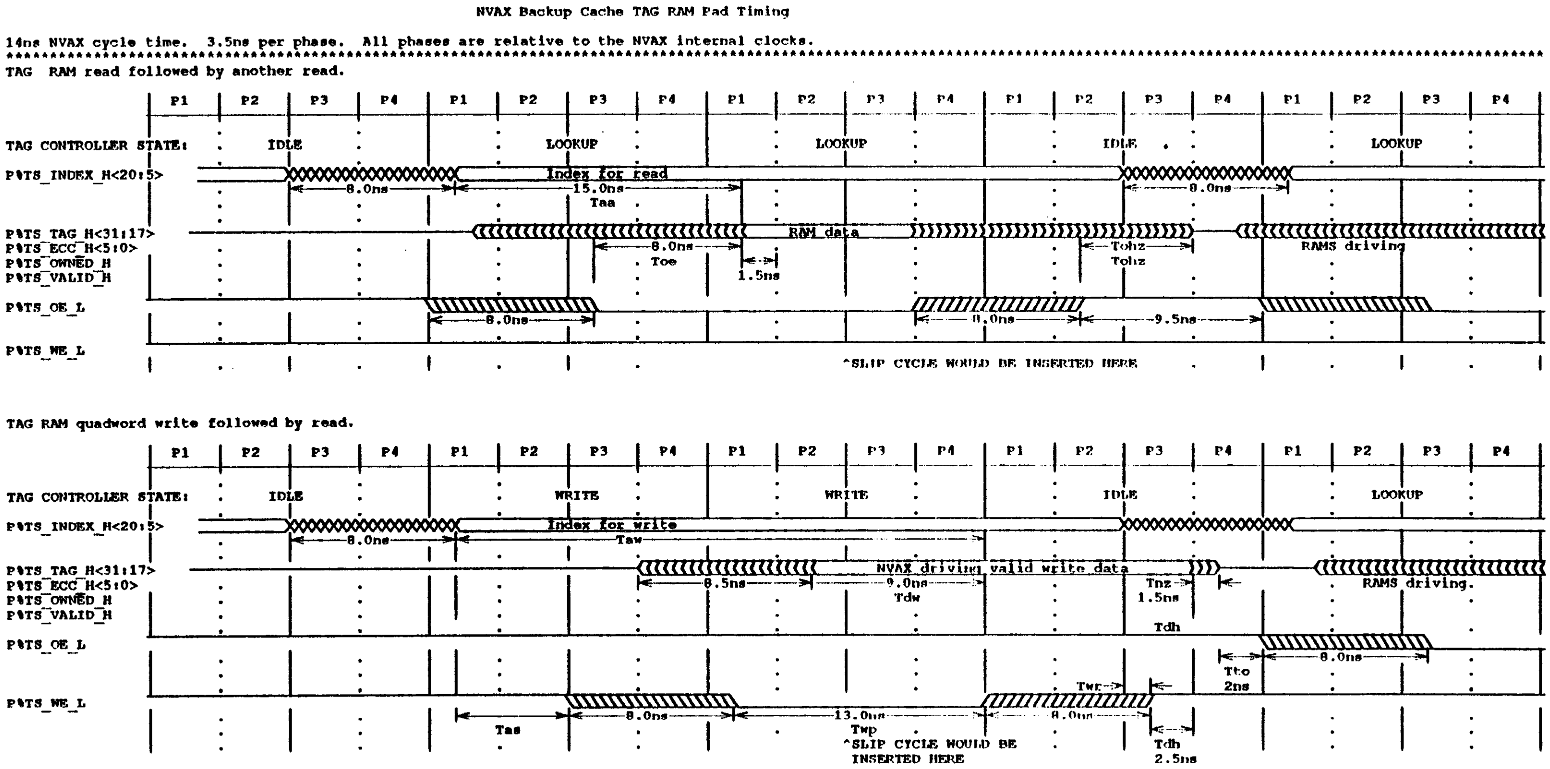
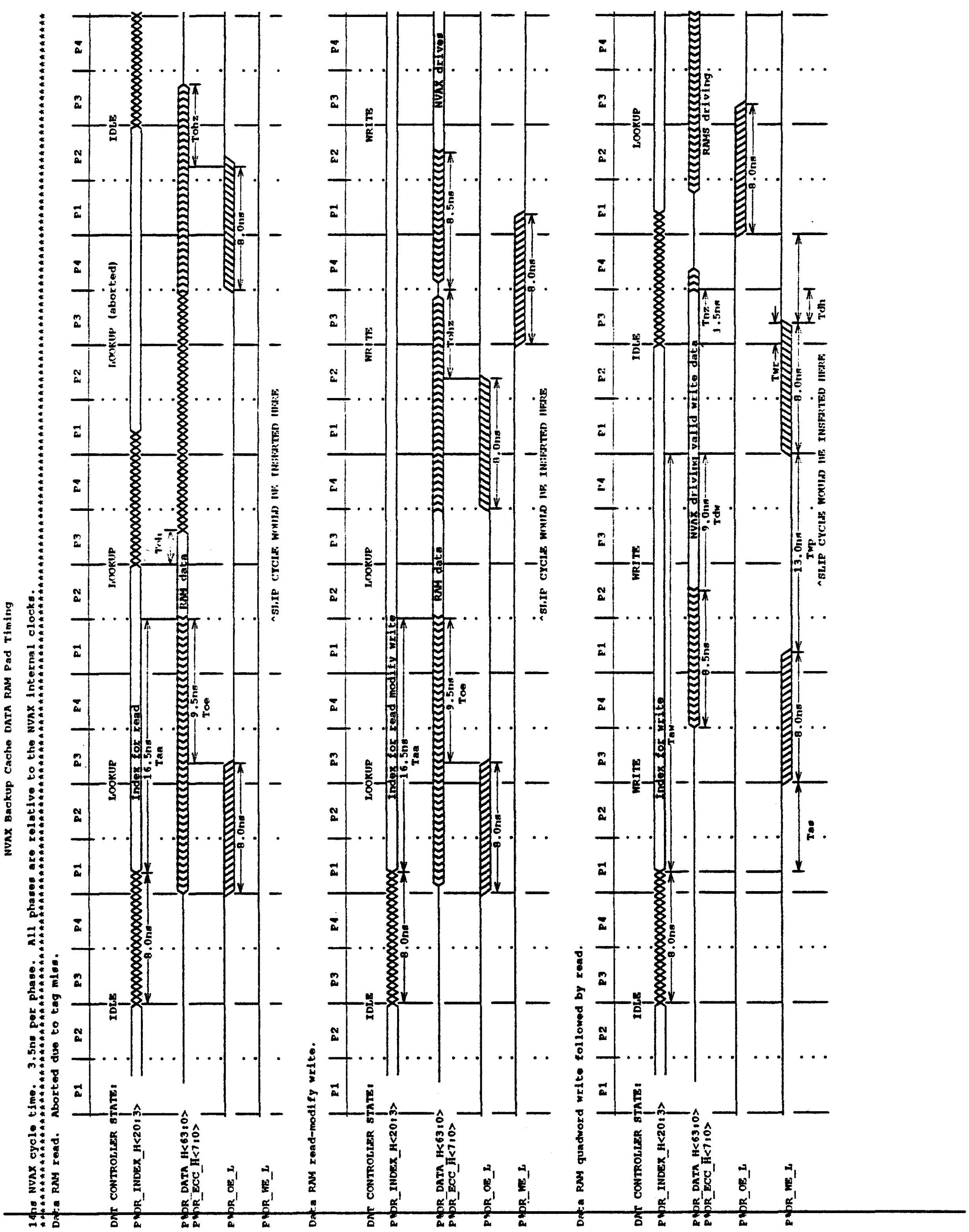


Figure 13–3: Backup Cache Data RAM Pin Timing


NVAX Backup Cache DATA RAM Pad Timing
14ns NVAX cycle time. 3.5ns per phase. All phases are relative to the NVAX internal clocks.
Data RAM read. Aborted due to tag miss.
P1 P2 P3 P4 P1 P2 P3 P4 P1 P2 P3 P4 P1 P2 P3 P4 P1 P2 P3 P4
DAT CONTROLLER STATE:
IDLE
LOOKUP
LOOKUP
LOOKUP (aborted)
IDLE
P%DR_INDEX_H<20:3>
Index for read
8.0ns
16.5ns
Taa
Toh
P%DR_DATA_H<63:0>
P%DR_ECC_H<7:0>
RAM data
9.5ns
Toe
Tohz
P%DR_OE_L
8.0ns
8.0ns
P%DR_WE_L
^SLIP CYCLE WOULD BE INSERTED HERE
Data RAM read-modify write.
P1 P2 P3 P4 P1 P2 P3 P4 P1 P2 P3 P4 P1 P2 P3 P4 P1 P2 P3 P4
DAT CONTROLLER STATE:
IDLE
LOOKUP
LOOKUP
WRITE
WRITE
P%DR_INDEX_H<20:3>
Index for read modify write
8.0ns
16.5ns
Taa
P%DR_DATA_H<63:0>
P%DR_ECC_H<7:0>
RAM data
NVAX drives
9.5ns
Toe
Tohz
8.5ns
P%DR_OE_L
8.0ns
8.0ns
P%DR_WE_L
8.0ns
^SLIP CYCLE WOULD BE INSERTED HERE
Data RAM quadword write followed by read.
P1 P2 P3 P4 P1 P2 P3 P4 P1 P2 P3 P4 P1 P2 P3 P4 P1 P2 P3 P4
DAT CONTROLLER STATE:
IDLE
WRITE
WRITE
IDLE
LOOKUP
P%DR_INDEX_H<20:3>
Index for write
8.0ns
Taw
P%DR_DATA_H<63:0>
P%DR_ECC_H<7:0>
NVAX driving valid write data
RAMS driving.
8.5ns
9.0ns
Tdw
Tnz
1.5ns
P%DR_OE_L
8.0ns
P%DR_WE_L
Tas
8.0ns
13.0ns
Twp
8.0ns
Twr
Tdh
^SLIP CYCLE WOULD BE INSERTED HERE

13.3.1 Backup Cache Interface

This section describes the NVAX pins dedicated to the backup cache interface. These are listed in Table 13–6.

Table 13–6: NVAX Backup Cache Interface Pins

Signal	Number	Input/output	Type
BACKUP CACHE TAG STORE SIGNALS (41 total)			
P%TS_INDEX_H<20:5>	16	Output	One driver, six receivers
P%TS_OE_L	1	Output	One driver, six receivers
P%TS_WE_L	1	Output	One driver, six receivers
P%TS_TAG_H<31:17>	15	Input/Output	Tristate, seven drivers/receivers
P%TS_ECC_H<5:0>	6	Input/Output	Tristate, seven drivers/receivers
P%TS_OWNED_H	1	Input/Output	Tristate, seven drivers/receivers
P%TS_VALID_H	1	Input/Output	Tristate, seven drivers/receivers
BACKUP CACHE DATA RAM SIGNALS (92 total)			
P%DR_INDEX_H<20:3>	18	Output	One driver, eighteen receivers
P%DR_OE_L	1	Output	One driver, eighteen receivers
P%DR_WE_L	1	Output	One driver, eighteen receivers
P%DR_DATA_H<63:0>	64	Input/Output	Tristate, nineteen drivers/receivers
P%DR_ECC_H<7:0>	8	Input/Output	Tristate, nineteen drivers/receivers

The pins listed are described in the sections which follow.

13.3.1.1 P%TS_INDEX_H<20:5>

These pins drive the address lines of the tag RAMs, thus indexing into one row of the tag store. The value driven depends upon the corresponding bits in the address of the memory or IPR reference being done.

P%TS_INDEX_H<16:5> are used for every cache configuration. P%TS_INDEX_H<20:17> are used based on the cache size selected. When the cache size selected is smaller than 2 megabytes, some or all of these four bits are driven to 0 rather than to the value given in the address. This is shown in Table 13–7.

Table 13–7: Usage of P%TS_INDEX_H<20:5> based on cache size

Cache size	P%TS_INDEX_H bits driven unconditionally to 0	P%TS_INDEX_H bits used
128 kilobytes	**P%TS_INDEX_H<20:17>**	**P%TS_INDEX_H<16:5>**
256 kilobytes	**P%TS_INDEX_H<20:18>**	**P%TS_INDEX_H<17:5>**
512 kilobytes	**P%TS_INDEX_H<20:19>**	**P%TS_INDEX_H<18:5>**
2 megabytes	None	**P%TS_INDEX_H<20:5>**

P%TS_INDEX_H<20:5> are driven by NVAX and received by up to 6 RAM chips.

13.3.1.2 P%TS_OE_L

P%TS_OE_L (Tag Store Output Enable) is an output pin which controls the tag store RAMs. It enables the RAMs to drive their outputs. It is asserted (driven low) when the tag store is being read, and allows the tag store to drive **P%TS_TAG_H<31:17>**, **P%TS_ECC_H<5:0>**, **P%TS_OWNED_H** and **P%TS_VALID_H**. When the tag store is being written, **P%TS_OE_L** is deasserted (driven high).

P%TS_OE_L is driven by NVAX and received by up to 6 RAM chips.

13.3.1.3 P%TS_WE_L

P%TS_WE_L (Tag Store Write Enable) is an output pin which, when asserted, enables the tag store RAMs to be written. It is asserted (driven low) during writes of the tag store.

P%TS_WE_L is driven by NVAX and received by up to 6 RAM chips.

13.3.1.4 P%TS_TAG_H<31:17>

P%TS_TAG_H<31:17> are I/O pins which are used to transfer the cache tag to and from the tag store RAMs. When the tag store is being written, **P%TS_TAG_H<31:17>** are used as outputs; when the tag store is being read, **P%TS_TAG_H<31:17>** are used as inputs.

Some of the tag lines are not used when the cache is bigger than 128 kilobytes, as shown in Table 13–8. When this is the case, the board designer does not need to connect the pin at all on the board. The pin is pulled low through a resistor in the pad so that internal to the Cbox, the unused tag lines are recognized as zeros when the tag is read.

Table 13–8: Usage of P%TS_TAG_H<20:17> based on cache size

Cache size	Unused P%TS_TAG_H pins
128 kilobytes	None
256 kilobytes	**P%TS_TAG_H<17>**
512 kilobytes	**P%TS_TAG_H<18:17>**
2 megabytes	**P%TS_TAG_H<20:17>**

All of the **P%TS_TAG_H<31:17>** pads are built with internal resistors, for chip layout consistency.

Each **P%TS_TAG_H** pin is connected to one RAM I/O pin. A system designer who intends to run NVAX only in 30-bit mode can leave **P%TS_TAG_H<31:29>** unconnected, and they will be pulled low internally so that the Cbox sees a zero value.

13.3.1.5 P%TS_ECC_H<5:0>

P%TS_ECC_H<5:0> are I/O pins which are used to transfer the ECC check bits to and from the tag store RAMs. When the tag store is being written, **P%TS_ECC_H<5:0>** are used as outputs; when the tag store is being read, **P%TS_ECC_H<5:0>** are used as inputs.

Each **P%TS_ECC_H** pin is connected to one RAM I/O pin.

13.3.1.6 P%TS_OWNED_H

P%TS_OWNED_H is an I/O pin which is used to transfer the ownership bit to and from the tag store RAMs. When the tag store is being written, **P%TS_OWNED_H** is used as an output; when the tag store is being read, **P%TS_OWNED_H** is used as an input.

P%TS_OWNED_H is connected to one RAM I/O pin.

13.3.1.7 P%TS_VALID_H

P%TS_VALID_H is an I/O pin which is used to transfer the valid bit to and from the tag store RAMs. When the tag store is being written, **P%TS_VALID_H** is used as an output; when the tag store is being read, **P%TS_VALID_H** is used as an input.

P%TS_VALID_H is connected to one RAM I/O pin.

13.3.1.8 P%DR_INDEX_H<20:3>

These pins drive the address lines of the data RAMs, thus indexing into one row of the data store. The value driven depends upon the corresponding bits in the address of the memory reference being done.

P%DR_INDEX_H<16:3> are used for every cache configuration. **P%DR_INDEX_H<20:17>** are used based on the cache size selected. When the cache size selected is smaller than 2 megabytes, some or all of these four bits are driven to 0 rather than to the value given in the address. This is shown in Table 13–9.

Table 13–9: Usage of P%DR_INDEX_H<20:5> based on cache size

Cache size	P%DR_INDEX_H bits driven unconditionally to 0	P%DR_INDEX_H bits used
128 kilobytes	**P%DR_INDEX_H<20:17>**	**P%DR_INDEX_H<16:5>**
256 kilobytes	**P%DR_INDEX_H<20:18>**	**P%DR_INDEX_H<17:5>**
512 kilobytes	**P%DR_INDEX_H<20:19>**	**P%DR_INDEX_H<18:5>**

Table 13–9 (Cont.): Usage of P%DR_INDEX_H<20:5> based on cache size

Cache size	P%DR_INDEX_H bits driven unconditionally to 0	P%DR_INDEX_H bits used
2 megabytes	None	P%DR_INDEX_H<20:5>

P%DR_INDEX_H<16:5> are driven by NVAX and received by 18 RAM chips.

13.3.1.9 P%DR_OE_L

P%DR_OE_L (Data RAM Output Enable) is an output pin which controls the data RAMs. It enables the RAMs to drive their outputs. It is asserted (driven low) when the data RAMs are being read, and allows the data RAMs to drive **P%DR_DATA_H<63:0>** and **P%DR_ECC_H<7:0>**. When the data RAMs are being written, **P%DR_OE_L** is deasserted (driven high).

P%DR_OE_L is driven by NVAX and received by 18 RAM chips.

13.3.1.10 P%DR_WE_L

P%DR_WE_L (Data RAM Write Enable) is an output pin which, when asserted, enables the data RAMs to be written. It is asserted (driven low) during writes of the data RAMs.

P%DR_WE_L is driven by NVAX and received by 18 RAM chips.

13.3.1.11 P%DR_DATA_H<63:0>

P%DR_DATA_H<63:0> are I/O pins which are used to transfer the cache data to and from the data RAMs. When the data RAMs are being written, **P%DR_DATA_H<63:0>** are used as outputs; when the data RAMs are being read, **P%DR_DATA_H<63:0>** are used as inputs.

Each one of **P%DR_DATA_H<63:0>** is connected to one RAM I/O pin.

13.3.1.12 P%DR_ECC_H<7:0>

P%DR_ECC_H<7:0> are I/O pins which are used to transfer the data ECC to and from the data store. When the data store is being written, **P%DR_ECC_H<7:0>** are used as outputs; when the data store is being read, **P%DR_ECC_H<7:0>** are used as inputs.

Each one of **P%DR_ECC_H<7:0>** is connected to one RAM I/O pin.

13.3.2 Backup Cache Block Diagrams

Figure 13–4 and Figure 13–5 show the connections to the tag store and data RAMs and the way the address is used for the 128-kilobyte cache.

Figure 13–4: Tags and Data for 128-Kilobyte Cache

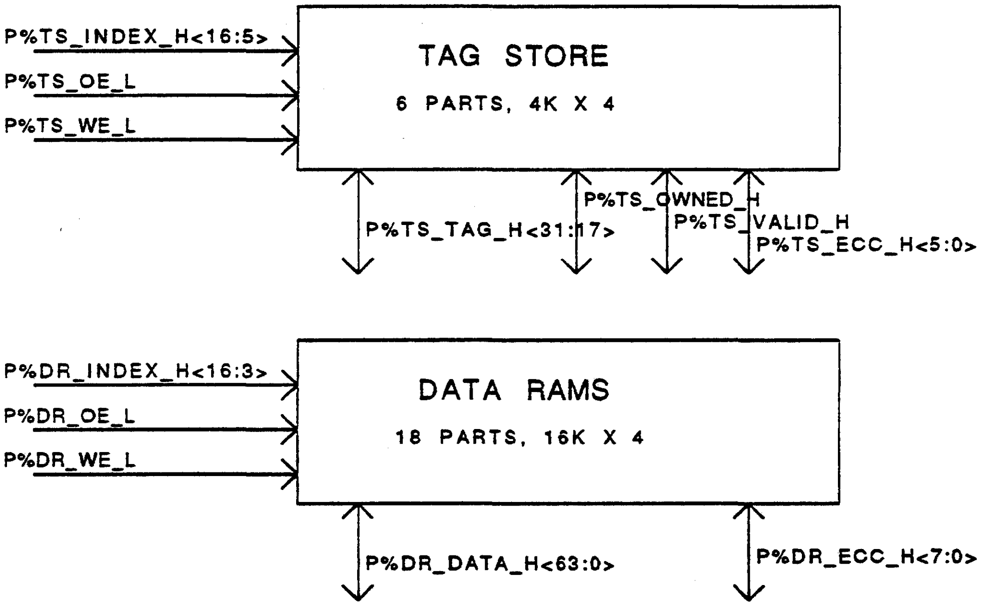


Figure 13–5: Address as used for 128-Kilobyte Cache

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|         tag - 15 bits                      | data and tag store index - 12 bits|     | UNUSED |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                                                    |
                                     used to address data quadword within hexaword--'
                                     unused for tag store
```

Figure 13–6 and Figure 13–7 show the connections to the tag store and data RAMs and the way the address is used for the 256-kilobyte cache.

Figure 13–6: Tags and Data for 256-Kilobyte Cache

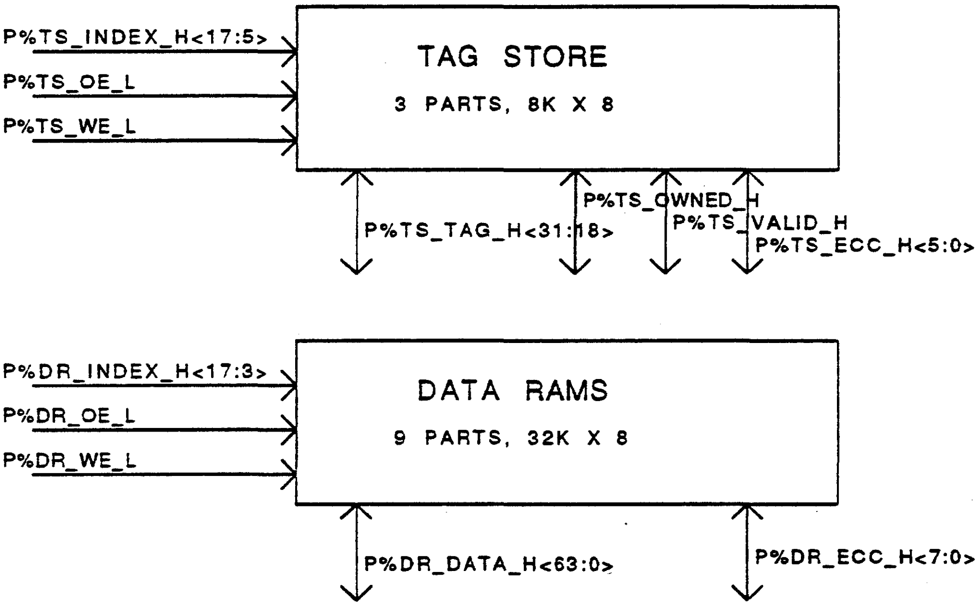


Figure 13–7: Address as used for 256-Kilobyte Cache

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|         tag - 14 bits                  | data and tag store index - 13 bits    |     | UNUSED |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                                                    |
                                     used to address data quadword within hexaword--'
                                     unused for tag store
```

Figure 13–8 and Figure 13–9 show the connections to the tag store and data RAMs and the way the address is used for the 512-kilobyte cache.

Figure 13–8: Tags and Data for 512-Kilobyte Cache

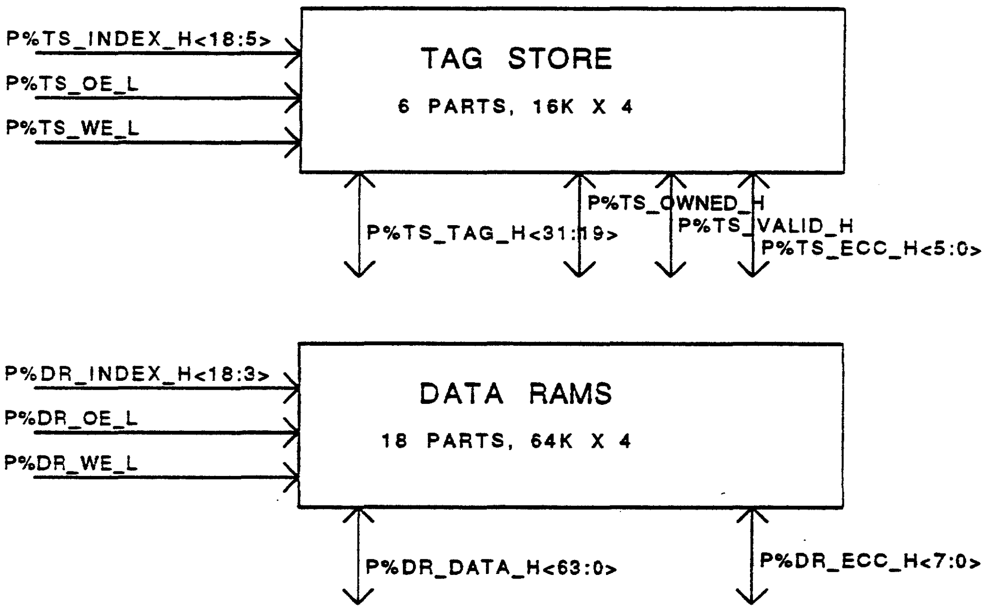


Figure 13–9: Address as used for 512-Kilobyte Cache

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|            tag - 13 bits                      |  data and tag store index - 14 bits                  |       | UNUSED  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
                                                                                                               |
                                                               used to address data quadword within hexaword--'
                                                               unused for tag store
```

Figure 13–10 and Figure 13–11 show the connections to the tags and data RAMs and the way the address is used for the 2-megabyte cache.

Figure 13–10: Tags and Data for 2-Megabyte Cache

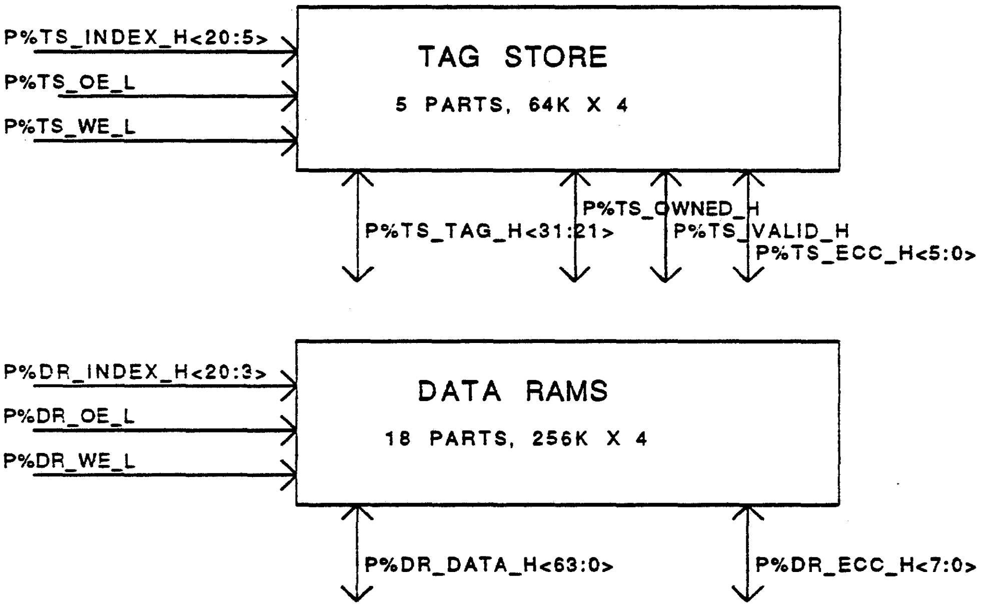


Figure 13–11: Address as used for 2-Megabyte Cache

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|          tag - 11 bits           |     data and tag store index - 16 bits          |     | UNUSED |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                                                        |
                                     used to address data quadword within hexaword--'
                                     unused for tag store
```

13.4 The Cbox Datapath

The Cbox includes datapath and control for interfacing to the Mbox, the cache RAMs, and to the NDAL. The portion of the Cbox which primarily interfaces to the Mbox and the cache RAMs will be referred to here as the Cbox proper, while the portion of the Cbox which primarily interfaces to the NDAL will be referred to as the BIU.

The Cbox datapath is organized around a number of queues and latches, an address bus and a data bus in the Cbox proper, and an address bus and a data bus in the BIU. Separate access is provided to the tag store and the data RAMs.

Table 13–10 lists the Cbox queues and the major latches. Each is covered in more detail later in the section. The IPRs are not covered here, as they are covered in Section 13.5.

Table 13–10: Cbox Queues and Major Latches

Queue/Latch	Entries	Address/Data	Function
CM_OUT_LATCH	1	Address<31:3> and data<63:0>	Holds fill data or an invalidate address being sent to the Mbox.
FILL_DATA_PIPEs	2	Data<63:0>	Pipeline data destined for the Mbox.
DREAD_LATCH	1	Address<31:0>	Holds a data-stream read request from the Mbox.
IREAD_LATCH	1	Address<31:0>	Holds an instruction-stream read request from the Mbox.
WRITE_PACKER	1	Address<31:0> and data<63:0>	Compresses sequential memory writes to the same quadword.
WRITE_QUEUE	8	Address<31:0> and data<63:0>	Queues write requests from the Mbox.
FILL_CAM	2	Address<31:3>	Holds addresses for read or write misses which have resulted in a read to memory; one may hold the address of an in-progress DREAD_LOCK which has no memory request outstanding.
NDAL_IN_QUEUE	10	Address<31:5> or data<63:0>	Holds up to 8 quadword fills and up to 2 coherence transactions from the NDAL.
WRITEBACK_QUEUE	2	Address<31:3> and data<63:0> times 4	Holds writeback addresses and data to be driven on the NDAL. The queue holds up to 2 hexaword writebacks. It is also used for quadword WDISOWNs.
NON_WRITEBACK_QUEUE	2	Address<31:0> and data<63:0>	The NON_WRITEBACK_QUEUE holds all non-WDISOWN transactions destined for the NDAL. This includes reads, I/O space transactions, and normal writes which are done when the cache is off or in ETM.

It can be seen from Table 13–10 that some of the queues contain address and data entries in parallel (CM_OUT_LATCH, WRITE_PACKER, WRITE_QUEUE, WRITEBACK_QUEUE, NON_WRITEBACK_QUEUE), some contain either addresses or data (NDAL_IN_QUEUE), some contain only data (FILL_DATA_PIPE), and some contain only addresses (DREAD_LATCH, IREAD_LATCH, FILL_CAM).

The Cbox is organized around an address datapath and a data datapath. A block diagram of the data datapath is given in Figure 13–12, and a block diagram of the address datapath is given in Figure 13–13.

There are five major busses in the Cbox: C_BUS%DBUS_H<63:0>, C_BUS%BIU_DATA_H<63:0>, C_ADC%ABUS_H<31:0>, C_ADC%BIU_ADDR_OUT_H<31:0> and C_BIU%ADC_ADDR_IN_H<31:0>. The first two transfer data, and the last three transfer addresses. From the block diagrams, it can be seen which of the latches and queues are connected to which busses. Transfers between address and data are connected only through the Abus/Dbus Xfer block, which is in the BIU.

The data flows may be understood by examining Figure 13–12. Write data enters the Cbox through the WRITE_QUEUE and is written into the data RAMs. When a writeback of a block occurs, data is read out of the data RAMs, transferred to the WRITEBACK_QUEUE in the BIU, and is driven onto the NDAL.

When read data is read from the backup cache, it is sent to the Mbox through the CM_OUT_LATCH. When read data returns from memory, it enters the Cbox through the NDAL_IN_QUEUE, is driven across C_BUS%BIU_DATA_H<63:0> to C_BUS%DBUS_H<63:0> and into the data RAMs, as well as to the Mbox through the CM_OUT_LATCH.

When the Bcache is off, write data is sent from the WRITE_QUEUE directly to the NON_WRITEBACK_QUEUE and to memory, bypassing the cache entirely.

The last data flow of significance has to do with the reading and writing of IPRs. The Dbus IPRs and the NDAL IPRs are read and written directly from the data datapath.

The address flows may be understood by examining Figure 13–13. Address bits <31:3> are used for memory space reads and writes, which always address a quadword boundary. Address bits <31:0> are used for I/O space reads and writes, which may address individual bytes.

Read addresses arrive through the IREAD_LATCH and the DREAD_LATCH, and write addresses arrive via the WRITE_QUEUE. Each address is driven across C_ADC%ABUS_H<31:0> to the tag RAMs, where it is looked up so that hit may be calculated. The index portion of the address is also driven to the data RAMs in case of a hit.

If a read or a write results in a hit, the data is sent back to the Mbox via the CM_OUT_LATCH. The requested quadword is always sent first on a Bcache hit. Bits <4:3> are driven onto C%MBOX_FILL_QW_H<4:3> to enable the Mbox to distinguish between quadwords within a hexaword. The most significant bits are not driven for fill data, as the Mbox knows from its miss latches and the fill command (D_CF or I_CF) which hexaword address the data corresponds to.

If the read or write does not result in a Bcache hit, the miss address is loaded into the FILL_CAM, which holds addresses of outstanding read and write misses; the address is also driven to the BIU, where it enters the NON_WRITEBACK_QUEUE to be driven onto the NDAL. When the fill data returns, the value of the NDAL signal P%ID_H<0> is used to locate the correct one of the two addresses in the FILL_CAM so that the data RAMs and the tag RAMs may be written. The address is driven out of the FILL_CAM to index the tag and data RAMs.

Another address-type operation occurs when a cache coherency transaction appears on the NDAL. In this case, the address comes in through the NDAL_IN_QUEUE and is driven from the BIU to the CBOX proper through the CBOX_BIU_INTERFACE. The address is looked up in the tag RAMs, and if it hits, the address is sent through the CM_OUT_LATCH to the Mbox for a Pcache invalidate. If necessary, the VALID and/or OWNED bit is cleared for the Bcache entry. Only address bits <31:5> are used for invalidates, as the invalidate is always to a hexaword.

If a writeback is required, the index is driven to the data RAMs so the data can be read out. The address is then driven to the WRITEBACK_QUEUE for the writeback; it is followed shortly by the writeback data on the data busses.

When Abus IPRs are read or written, the address busses and the data busses come into play. When an Abus IPR is read, the data is driven onto C_ADC%ABUS_H<31:0> and then to C_ADC%BIU_ADDR_OUT_H<31:0>. The BIU uses the Abus/Dbus XFER block to transfer the data to C_BUS%BIU_DATA_H<63:0>; it then goes to C_BUS%DBUS_H<63:0> and back to the Mbox through the CM_OUT_LATCH.

When an Abus IPR is written, the data is driven from the Mbox through the WRITE_QUEUE, to C_BUS%DBUS_H<63:0>, and to C_BUS%BIU_DATA_H<63:0>. The Abus/Dbus XFER block transfers the data to C_ADC%BIU_ADDR_OUT_H<31:0>, and it is then driven to C_ADC%ABUS_H<31:0> so that it can be written into the register.

The byte mask is received from the Mbox for writes and I/O space reads. It is passed through the Cbox and onto the NDAL for writes when the cache is off or in ETM, and it is passed through to the NDAL for all I/O space transactions.

Figure 13–12: Cbox block diagram with DATA_BUS

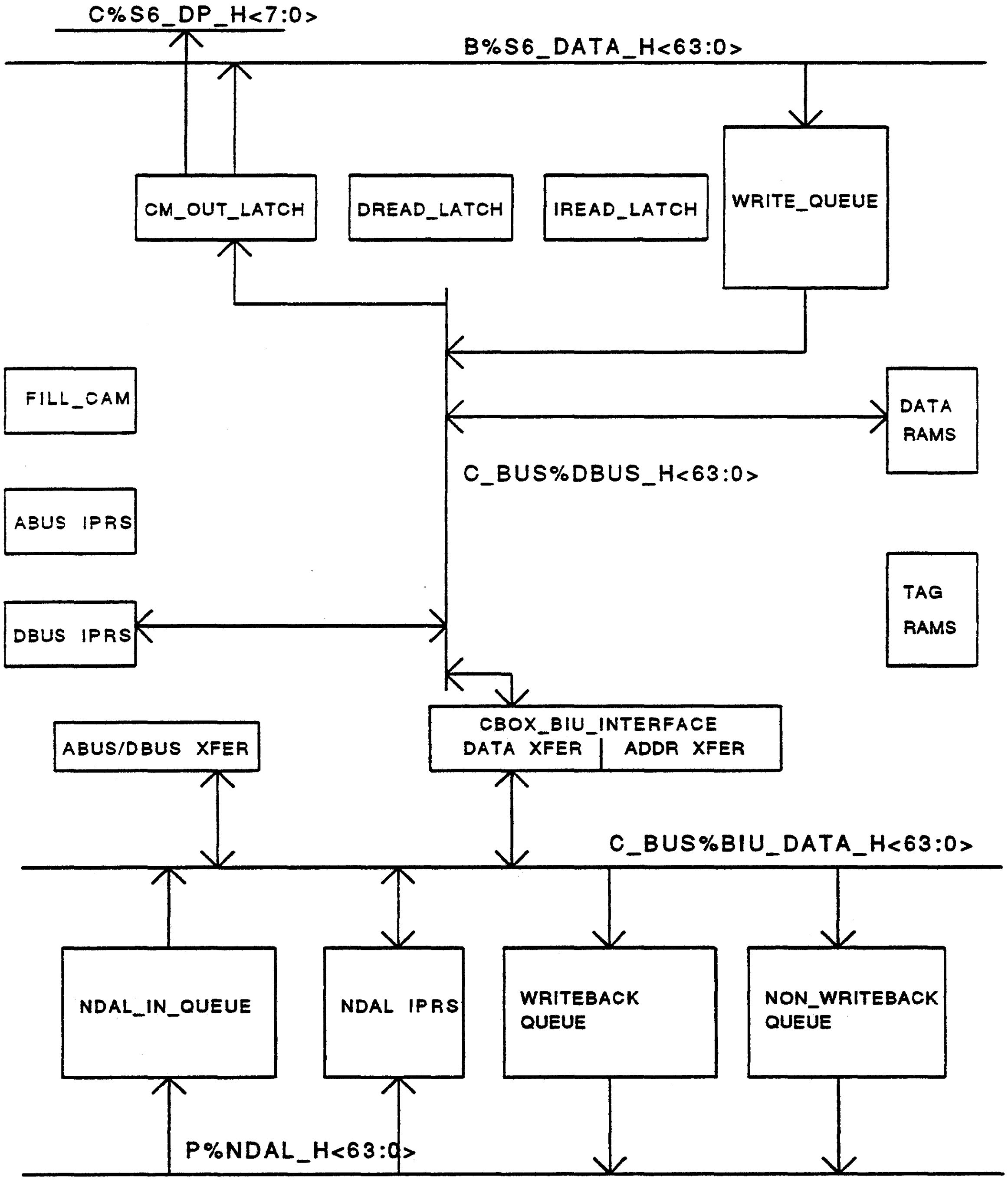
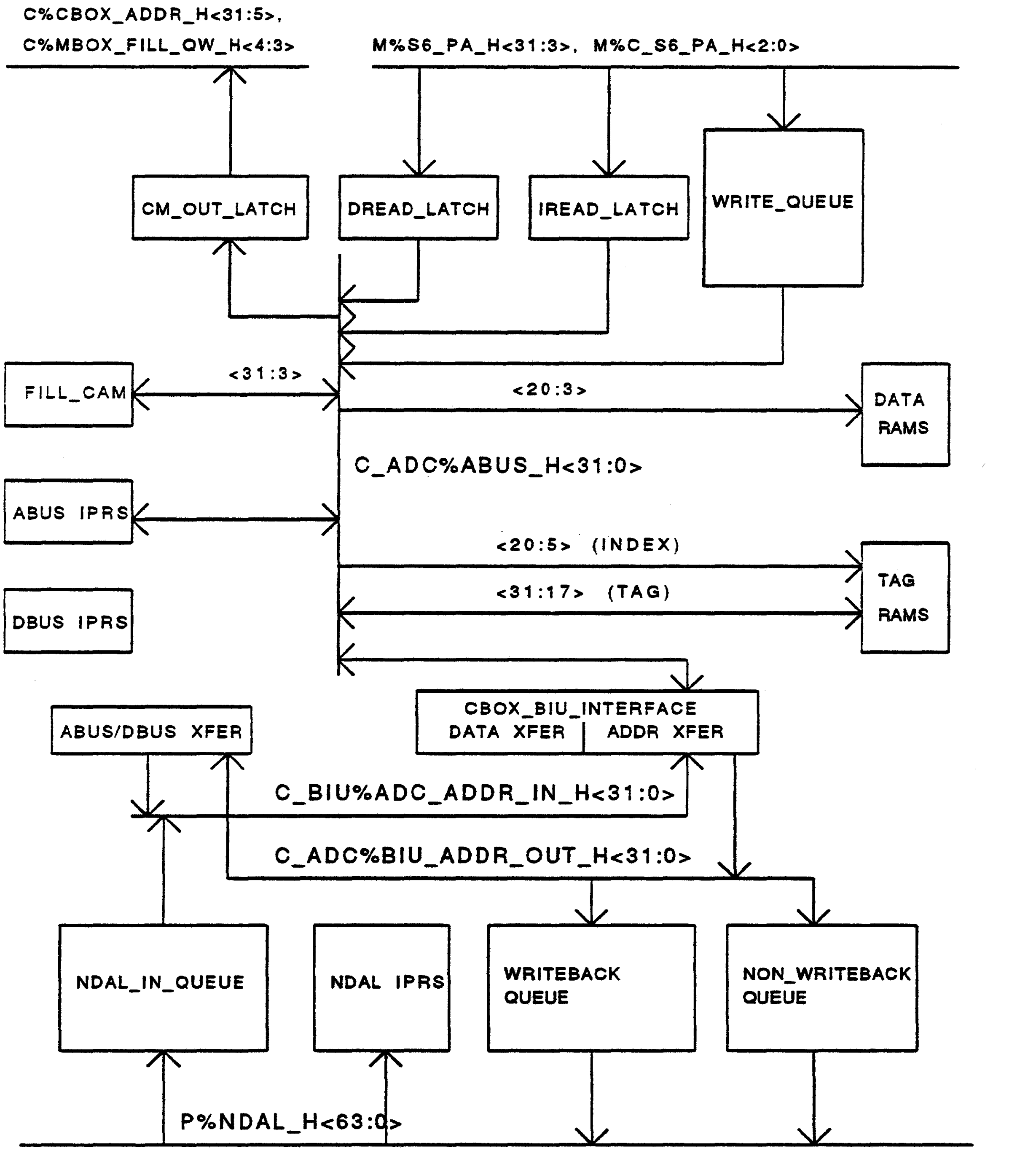


Figure 13–13: Cbox block diagram with ADDRESS_BUS

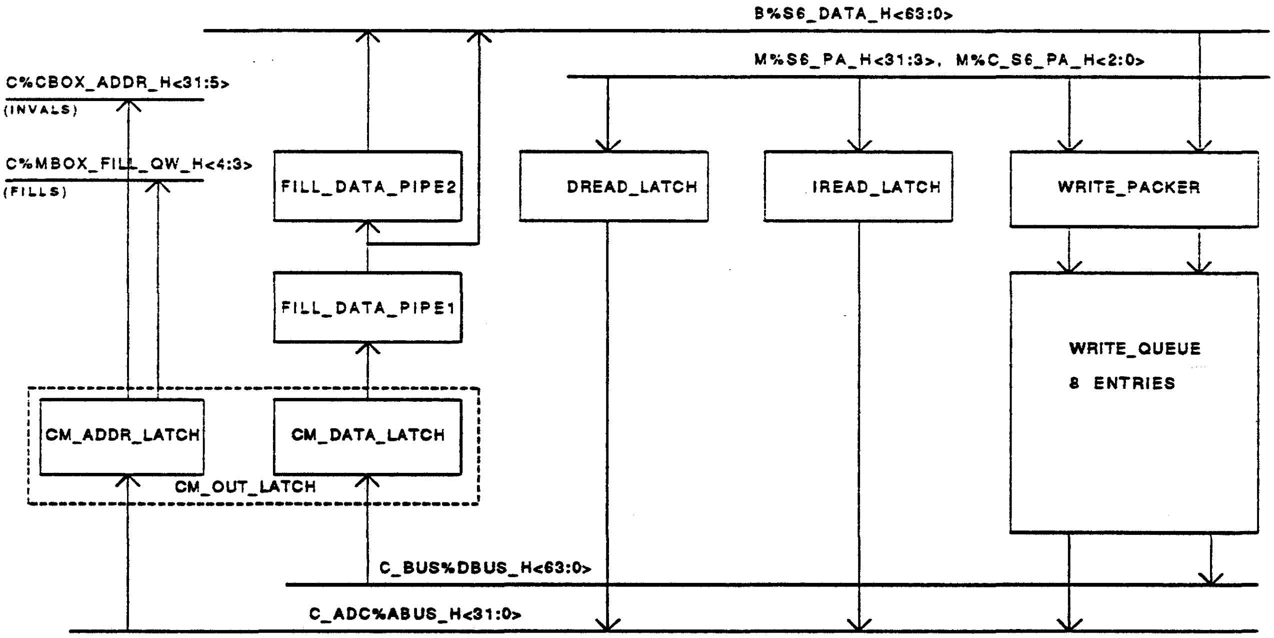


13.4.1 Mbox Interface

All NVAX CPU chip transactions for the Cbox arrive through the Cbox-Mbox interface. Reads come from the Mbox to the Cbox through the read latches. Writes arrive through the WRITE_PACKER and the WRITE_QUEUE. All fills returning from the Cbox to the Mbox go through the CM_OUT_LATCH.

A block diagram of the Mbox interface is shown in Figure 13–14.

Figure 13–14: Mbox Interface



When the Mbox has a command for the Cbox, the command appears on **M%S6_CMD_H<4:0>**. **M%CBOX_REF_ENABLE_L** is asserted for all reads, IPR_RDs, and IPR_WRs. It is not asserted for writes since the Cbox accepts all writes from the Mbox. The Cbox loads the address from **M%S6_PA_H<31:3>** and **M%C_S6_PA_H<2:0>** into either the IREAD_LATCH, the DREAD_LATCH, or the WRITE_PACKER. If the command is a write, the Cbox loads the data from **B%S6_DATA_H** and the byte enable from **M%S6_BYTE_MASK_H** into the WRITE_PACKER.

Table 13–11 shows the commands which pass between the Mbox and the Cbox.

Table 13–11: Mbox-Cbox Commands

Command	Description	Cbox datapath element involved
Mbox to Cbox commands driven on M%S6_CMD_H<4:0>		
IREAD[1]	Instruction stream read	IREAD_LATCH
DREAD[1]	Data stream read	DREAD_LATCH
DREAD_MODIFY[1]	Data stream read with modify intent	DREAD_LATCH
DREAD_LOCK[1]	Interlocked data stream read	DREAD_LATCH
WRITE_UNLOCK	Write which releases lock	WRITE_PACKER, WRITE_QUEUE
WRITE	Normal write	WRITE_PACKER, WRITE_QUEUE
IPR_RD[1]	Read of an internal or external processor register	DREAD_LATCH
IPR_WR[1]	Write of an internal or external processor register	WRITE_PACKER, WRITE_QUEUE
Cbox to Mbox commands driven on C%CBOX_CMD_H<1:0>		
D_CF	Data stream cache fill	CM_OUT_LATCH
I_CF	Instruction stream cache fill	CM_OUT_LATCH
INVAL	Hexaword invalidate	CM_OUT_LATCH
NOP	No operation.	

[1] Qualified by M%CBOX_REF_ENABLE_L.

13.4.1.1 Mbox to Cbox Transactions

The Mbox commands and accompanying control and data signals are shown in Table 13–12. **M%CBOX_REF_ENABLE_L** and **M%CBOX_LATE_EN_H** are used to enable certain transactions coming to the Cbox; **M%CBOX_LATE_EN_H** is only used for transactions which may hit in the Pcache. From the table, is may be seen that the assertion of **M%CBOX_REF_ENABLE_L** is not necessary for writes and write unlocks; and that **M%CBOX_LATE_EN_H** is only used for DREADs, IREADs, and READ MODIFYs. **M%S6_BYTE_MASK_H<7:0>** is valid for all transactions, although **B%S6_DATA_H<63:0>** is not valid for read transactions.

Table 13–12: Mbox to Cbox Command Matrix

	Mbox-driven Signal or Bus				
M%S6_CMD_H<4:0>	M%CBOX_REF_ENABLE_L	M%CBOX_LATE_EN_H	M%S6_PA_H<31:3> M%C_S6_PA_H<2:0>	M%S6_BYTE_MASK_H<7:0>	B%S6_DATA_H<63:0>
DREAD	valid[1]	valid	valid	valid	X[2]
READ MODIFY	valid	valid	valid	valid	X
IREAD	valid	valid	valid	valid	X
READ LOCK	valid	0[3]	valid	valid	X
IPR READ	valid	0	valid	valid	X
IPR WRITE	valid	0	valid	valid	valid
WRITE	X	X	valid	valid	valid
WRITE UNLOCK	X	X	valid	valid	valid
OTHER	X	X	X	X	X

[1] "valid" denotes that the signal is either asserted or deasserted by the Mbox, and the Cbox interprets it appropriately.

[2] "X" denotes that the Mbox may drive any value to the Cbox, and the Cbox does not care what value is driven.

[3] "0" denotes that the Mbox never asserts the signal in this case.

13.4.1.1.1 The IREAD_LATCH and the DREAD_LATCH

When the Mbox has a read command for the Cbox, the Cbox loads the address from from **M%S6_PA_H<31:3>** and **M%C_S6_PA_H<2:0>** into either the IREAD_LATCH or the DREAD_LATCH, depending on the command. Only IREADs are loaded into the IREAD_LATCH. The DREAD_LATCH is used for DREAD, DREAD_MODIFY, DREAD_LOCK, and IPR_READ.

The Mbox only has one outstanding IREAD and one outstanding DREAD at a time, so no backpressure for the latches is needed. When the DREAD_LATCH is valid, the Mbox does not start the next DREAD-type transaction until all fill data from the previous command is returned to the Mbox. When the IREAD_LATCH is valid, the Mbox does not start the next IREAD transaction until either the IREAD has been aborted or all fill data from the IREAD is returned to the Mbox.

The Cbox services a read hit from the read latch; a read miss is transferred to the FILL_CAM where it awaits the arrival of data from memory. Table 13–13 and Table 13–14 show the fields which are contained in the two read latches.

Table 13–13: IREAD_LATCH Fields

Field	Purpose
ADDRESS<31:0>	Physical address of the read request.
CMD<4:0>	Specific command being done (IREAD).

Table 13–14: DREAD_LATCH Fields

Field	Purpose
CMD<4:0>	Specific command being done (DREAD, DREAD_MODIFY, DREAD_LOCK, IPR_READ).
ADDRESS<31:0>	Physical address of the read request.

When the Mbox asserts M%ABORT_CBOX_IRD_H, the Cbox clears the IREAD_LATCH entry if the reference has not yet started. If the CBOX is in the middle of the tag store lookup or in the middle of a hit sequence and returning the Iread fill data, it aborts the lookup or the data sequence. If a miss has already been initiated, the CBOX continues with the fills to the backup cache but does not send any data to the Mbox.

13.4.1.1.2 WRITE_PACKER and WRITE_QUEUE

Writes from the Mbox go through the WRITE_PACKER and into the WRITE_QUEUE. The WRITE_PACKER holds one quadword of data; the WRITE_QUEUE consists of 8 entries, each of which contains a quadword of data. The purpose of the WRITE_PACKER is to accumulate memory-space writes to the same quadword which arrive sequentially, so that only one write has to be done into the cache. Performance modelling shows that this can reduce by 70% the number of writes done to the backup cache.

Only normal WRITE commands to the same quadword are packed together. Other writes pass immediately from the WRITE_PACKER into the WRITE_QUEUE. The WRITE_PACKER is flushed at the following times:

- When a memory-space WRITE to a different quadword arrives. The new quadword then remains in the write packer until a write packer flush condition is met.
- When a WRITE_UNLOCK arrives. The WRITE_UNLOCK is then passed immediately from the WRITE_PACKER to the WRITE_QUEUE.
- When an I/O space write arrives. The I/O space write is then passed immediately from the WRITE_PACKER to the WRITE_QUEUE.
- When an IPR_WRITE arrives. The IPR_WRITE is then passed immediately from the WRITE_PACKER to the WRITE_QUEUE.
- If an IREAD or a DREAD arrives to the same hexaword as that of the entry in the WRITE_PACKER.

- Whenever any condition for flushing the write queue is met on the entry in the WRITE_PACKER.
- If the DISABLE_PACK bit in the CCTL IPR is set. In this case, every write passes directly through the WRITE_PACKER without delay.

THREE-CYCLE LATENCY THROUGH THE WRITE_QUEUE

> If the WRITE_QUEUE and the WRITE_PACKER are empty, the latency of any write through them is 3 cycles. The implication of this is that if any reads which flush the WRITE_QUEUE are done alternately with writes, their execution will be greatly slowed. This applies to IPR reads and writes and may be an issue in testing the chip.

Table 13–15 describes the fields in the WRITE_QUEUE.

Table 13–15: WRITE_QUEUE Fields

Field	Purpose
VALID	Indicates that the entry contains valid information.
DWR_CONFLICT	Indicates that this write conflicts with a DREAD, giving the WRITE_QUEUE priority. Check is done using hexaword address.
IWR_CONFLICT	Indicates that this write conflicts with an IREAD, giving the WRITE_QUEUE priority. Check is done using hexaword address.
CMD<4:0>	Specific command being done.
ADDRESS<31:0>	Physical address of the write.
BYTE_EN<7:0>	Byte enable for the write.
DATA<63:0>	Data to be written.

When a quadword of data is moved into the WRITE_QUEUE, it is serviced by the Cbox arbiter as the lowest-priority task, unless special conditions exist.

Servicing writes separately from reads allows reads to take higher priority and gets read data back to the CPU faster. However, a read which follows a write to the same hexaword must not be allowed to complete before the write completes. To prevent this there are conflict bits, DWR_CONFLICT<8:0> and IWR_CONFLICT<8:0>, implemented in the WRITE_QUEUE and WRITE_PACKER, one for each entry. The conflict bits ensure correct ordering between writes and a DREAD or an IREAD to the same hexaword.

When a DREAD arrives, the hexaword address is checked against all entries in the WRITE_QUEUE and WRITE_PACKER. Any entry with a matching hexaword address has its corresponding DWR_CONFLICT bit set. The DWR_CONFLICT bit is also set if the WRITE_QUEUE entry is an IPR_WRITE, a WRITE_UNLOCK, or an I/O space write. If any DWR_CONFLICT bit is set, the WRITE_QUEUE takes priority over DREADs, allowing the writes up to the point of the conflicting write to complete first.

When an IREAD arrives, the hexaword address is checked against all entries in the WRITE_QUEUE and WRITE_PACKER. Any entry with a matching hexaword address has its corresponding IWR_CONFLICT bit set. The IWR_CONFLICT bit is also set if the WRITE_QUEUE entry is an IPR_WRITE, a WRITE_UNLOCK, or an I/O space write. If any IWR_CONFLICT bit is set, the WRITE_QUEUE takes priority over IREADs, allowing the writes up to the point of the conflicting write to complete first.

As each write is done, the conflict bits and valid bit of the entry are cleared. When the last write which conflicts with a DREAD finishes, there are no more DWR_CONFLICT bits set, and the DREAD takes priority again, even if other writes arrived after the DREAD. In this way a DREAD which conflicts with previous writes is not done until those writes are done, but once those writes are done, the DREAD proceeds.

The analogous statement is true for an IREAD which has a conflict. If IWR_CONFLICT is set and the IREAD is aborted before the conflicting write queue entry is processed, the WRITE_QUEUE continues to take precedence over the IREAD_LATCH until the conflicting entry is retired.

If both a DREAD and an IREAD have a conflict in the WRITE_QUEUE, writes take priority until one of the reads no longer has a conflict. If the DREAD no longer has a conflict, the DREAD is then done. Then the WRITE_QUEUE continues to have priority over the IREAD_LATCH since the IREAD has a conflict, and when the conflicting writes are done, the IREAD may proceed. If another DREAD arrives in the meantime, it may be allowed to bypass both the writes and the IREAD if it has no conflicts.

This mechanism is used for other cases to enforce read/write ordering. Cases where the WRITE_QUEUE (and the WRITE_PACKER) must be flushed before proceeding are listed below:

1. DREAD_LOCK and WRITE_UNLOCK.
2. All IPR_READs and IPR_WRITEs (includes Clear Write Buffer).
3. All I/O space reads and I/O space writes.
4. Dread or Iread conflict with a write (checked to hexaword granularity, on address bits <31:5>).

When a DREAD_LOCK arrives from the MBOX, DWR_CONFLICT bits for all valid writes in the WRITE_QUEUE and WRITE_PACKER are set so that all writes preceding the DREAD_LOCK are done before the DREAD_LOCK is done.

When any IPR_READ arrives, all DWR_CONFLICT bits for valid entries in the WRITE_QUEUE and WRITE_PACKER are set, forcing the writes to complete before the IPR_READ completes. This ensures that IPR reads and writes are executed in order.

When any D-stream I/O space read arrives, all DWR_CONFLICT bits for valid entries in the WRITE_QUEUE and WRITE_PACKER are set, so that previous writes complete first.

When any I-stream I/O space read arrives, all IWR_CONFLICT bits for valid entries in the WRITE_QUEUE and WRITE_PACKER are set, so that previous writes complete first.

Note that when a WRITE_UNLOCK arrives, the WRITE_QUEUE is always empty as it was previously flushed before the READ_LOCK was serviced.

When a new entry for the DREAD_LATCH arrives, it is checked for conflicts with the WRITE_QUEUE. At this time the DWR_CONFLICT bit is set on any WRITE_QUEUE entry which is an I/O space write, an IPR_WRITE, or a WRITE_UNLOCK. Similarly, when a new entry for the IREAD_LATCH arrives, it is checked for conflicts with the WRITE_QUEUE. At this time the IWR_CONFLICT bit is set on any WRITE_QUEUE entry which is an I/O space write, an IPR_WRITE, or a WRITE_UNLOCK.

Thus, all transactions from the Mbox except memory space reads and writes unconditionally force the flushing of the WRITE_QUEUE. Memory space reads cause a flush if they conflict with a previous write.

If the WRITE_QUEUE fills up, the Cbox asserts C%WR_BUF_BACK_PRES_H. The Mbox then stops sending more writes to the Cbox until C%WR_BUF_BACK_PRES_H is deasserted.

13.4.1.2 Cbox to Mbox Transactions

The Cbox sends fills and invalidates to the Mbox. The signals which the Cbox drives in doing this are shown in Table 13–16.

Table 13–16: Cbox to Mbox Interface signals

Field	Purpose
C%CBOX_CMD_H<1:0>	Specific command being done: either D_CF, I_CF, INVAL, or NOP.
C%CBOX_ADDR_H<31:5>	Hexaword address for invalidate sent to Mbox
C%REQ_DQW_H	Indicates that the quadword of fill data being returned was the requested quadword of data: the quadword to which the original address corresponded. It is also asserted if C%CBOX_HARD_ERR_H is asserted and the requested quadword has not yet been returned; the Mbox then notifies the Ibox and/or Ebox that the requested data has been returned so that the machine does not hang.
C%LAST_FILL_H	Indicates that this is the last data being sent for the read request.
C%CBOX_HARD_ERR_H	Indicates that an unrecoverable error is associated with the data. This bit only qualifies fills, not invalidates. When C%CBOX_HARD_ERR_H is asserted, the Cbox also asserts C%LAST_FILL_H as no more fills follow. C%CBOX_HARD_ERR_H may be asserted as the result of an uncorrectable error in the Bcache or as the result of RDE on the NDAL.
C%CBOX_ECC_ERR_H	Indicates that a correctable backup cache ECC error is associated with the current fill data and the data should be ignored. Valid for fills only, not invalidates. Corrected data will follow.
C%MBOX_FILL_QW_H<4:3>	Address bits to indicate to which quadword within the hexaword the current fill data belongs.
B%S6_DATA_H<63:0>	Bus used to receive data from the Mbox and to send fill data to the Mbox.
C%S6_DP_H<7:0>	Byte data parity for B%S6_DATA_H<63:0>.

Table 13–17 shows what signals are driven and valid for every Cbox-to-Mbox transaction.

If an error in the backup cache or on the NDAL happens while fill data is being retrieved, the Cbox notifies the Mbox using C%CBOX_HARD_ERR_H or C%CBOX_ECC_ERR_H. Table 13–18 shows how both normal cases and error cases are handled by the Mbox.

Table 13–17: Cbox to Mbox Command Matrix

	C%CBOX_CMD_H<1:0>			
Cbox-driven Signal or Bus	**NOP (00)**	**INVAL (01)**	**I_CF (10)**	**D_CF (11)**
C%CBOX_ADDR_H<31:5>	X[1]	valid[2]	X	X
C%REQ_DQW_H	0[3]	0	valid[4]	valid
C%LAST_FILL_H	0	0	valid	valid
C%CBOX_HARD_ERR_H	0	0	valid	valid
C%CBOX_ECC_ERR_H	X	X	valid	valid
C%MBOX_FILL_QW_H<4:3>	X	X	valid	valid
B%S6_DATA_H<63:0>	not driven	not driven	driven	driven
C%S6_DP_H<7:0>	not driven	not driven	driven	driven

[1] "X" denotes that the Cbox may drive any value to the Mbox, and the Mbox does not care what value is driven.

[2] "valid" denotes that the signal is either asserted or deasserted by the Cbox, and the Mbox interprets it appropriately.

[3] "0" denotes that the Cbox never asserts the signal in this case.

[4] The Mbox ignores the value driven by the Cbox in this case.

Table 13–18: Cbox to Mbox commands and resulting Mbox actions

C%CBOX_CMD_H<1:0>	Qualifiers [1]	Mbox Action
NOP	Qualifiers do not apply.	Take no action.
I_CF or D_CF	None asserted.	Accept fill data for outstanding IREAD or DREAD; expect more.
I_CF or D_CF	C%LAST_FILL_H asserted	Accept fill data for outstanding IREAD or DREAD; expect no more.
I_CF or D_CF	C%CBOX_HARD_ERR_H, C%LAST_FILL_H	Perform invalidate, expect no more fills for this read. (C%LAST_FILL_H is always asserted when C%CBOX_HARD_ERR_H is asserted.)
I_CF or D_CF	C%CBOX_ECC_ERR_H	Ignore this fill data, expect fill later.
I_CF or D_CF	C%CBOX_ECC_ERR_H and C%LAST_FILL_H	Ignore this fill data, expect fill later.
I_CF or D_CF	C%CBOX_ECC_ERR_H and C%CBOX_HARD_ERR_H	This case never happens, and is disallowed.
INVAL	Qualifiers do not apply.	Perform invalidate.
INVAL to outstanding fill	Qualifiers do not apply.	Perform invalidate, expect fill data. Do not validate the data in the Pcache when it returns.

[1] Qualifiers covered in this table are: C%CBOX_HARD_ERR_H, C%LAST_FILL_H, and C%CBOX_ECC_ERR_H.

13.4.1.2.1 CM_OUT_LATCH

The CM_OUT_LATCH holds fill data and invalidate addresses which are destined for the Mbox. The Mbox never backpressures the Cbox (it can always receive a command from the Cbox) so a queue is not needed. The latch has an address portion and a data portion. The fields are shown in Table 13–19.

Table 13–19: CM_OUT_LATCH Fields

Field	Purpose
CMD<1:0>	Specific command being done.
ADDR<31:5>	Physical address of the invalidate. This field is not used for fills.
FILL_QW<4:3>	Quadword alignment of the fill. This field is not used for invalidates.
DATA<63:0>	Fill data.

The CM_OUT_LATCH is loaded with an invalidate when the backup cache deallocates a valid block or when it performs an invalidate due to a cache coherency transaction on the NDAL. The CM_OUT_LATCH is loaded with cache fill data when the NDAL returns fill data which was requested by the Mbox or when a read request hits in the backup cache. Cbox control ensures that both events never happen in the same cycle.

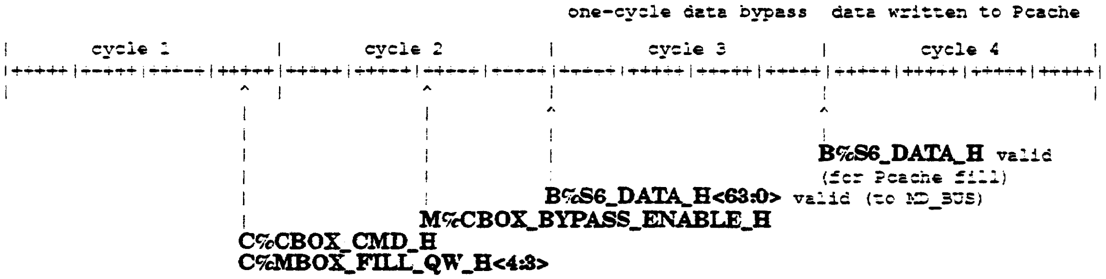
The command from the CM_OUT_LATCH is driven on C%CBOX_CMD_H<1:0>. If the command is an invalidate, the address is driven on C%CBOX_ADDR_H<31:5>, and no data is driven to the Mbox. If the command is a fill, the quadword alignment is driven on C%MBOX_FILL_QW_H<4:3>. (The Mbox has the hexaword address during these cycles.) Fill data is piped through the FILL_DATA_PIPEs and driven on B%S6_DATA_H<63:0>. The Cbox calculates byte parity on the fill data and drives it on C%S6_DP_H<7:0>.

If an IREAD is in progress in the Cbox and the MBOX asserts M%ABORT_CBOX_IRD_H, the Cbox prevents any further command, address, or data for that Iread from being driven to the Mbox, as described in Section 13.4.1.2.3.

13.4.1.2.2 FILL_DATA_PIPE1 and FILL_DATA_PIPE2

The FILL_DATA_PIPEs are used to pipeline the fill data for two cycles so that the Cbox drives B%S6_DATA_H<63:0> coincidentally with the write-enable of the Pcache. If there is a free cycle on B%S6_DATA_H<63:0>, the Cbox may bypass the fill data from the FILL_DATA_PIPE1 (to achieve a one-cycle bypass). This allows the Mbox to return data to the Ibox or the Ebox one cycle early. The cache fill to the Pcache is done in the normal cycle, driven from FILL_DATA_PIPE2, even if Ebox or Ibox data was bypassed in an earlier cycle. The timing relationships for one cache fill are shown in Figure 13–15.

Figure 13–15: B%S6_DATA_H<63:0> bypass timing



In this example, a fill is just arriving in cycle 1, so the Cbox drives C%CBOX_CMD_H and C%MBOX_FILL_QW_H<4:3>.

The Mbox drives M%CBOX_BYPASS_ENABLE_H to the Cbox in cycle 2 to indicate that B%S6_DATA_H is free during the current cycle. This causes the Cbox to bypass data from FILL_DATA_PIPE1 to B%S6_DATA_H to achieve a one-cycle bypass.

In cycle 3 the Cbox drives the data from FILL_DATA_PIPE2 to the Pcache for the write. It does this even though the bypass was done previously, because the Pcache is always written in the third cycle after C%CBOX_CMD_H is driven with the fill command.

The rules for the Cbox driving data on B%S6_DATA_H are as follows:

1. IF FILL_DATA_PIPE2 contains valid data, drive B%S6_DATA_H from FILL_DATA_PIPE2
2. ELSE IF M%CBOX_BYPASS_ENABLE_H is asserted and FILL_DATA_PIPE1 contains valid data, drive from FILL_DATA_PIPE1 to achieve a one-cycle bypass.

The Mbox keeps enough state to know what the Cbox will be bypassing in any given cycle.

When the Cbox drives B%S6_DATA_H, it also generates byte parity and drives C%S6_DP_H with the same timing.

The fields of the FILL_DATA_PIPEs are shown in Table 13–20.

Table 13–20: Fields of FILL_DATA_PIPE1 and FILL_DATA_PIPE2

Field	Purpose
IREAD	Indicates that fill data is for an IREAD.
DATA<63:0>	Fill data.

The IREAD field is necessary in case of an IREAD abort, as described in Section 13.4.1.2.3. If **M%ABORT_CBOX_IRD_H** is asserted and the data in either FILL_DATA_PIPE1 or FILL_DATA_PIPE2 is for an IREAD, that FILL_DATA_PIPE must be cleared so that data is not driven back to the Mbox.

13.4.1.2.3 IREAD Aborts

The Mbox asserts the signal **M%ABORT_CBOX_IRD_H** to notify the Cbox to abort any IREAD which it is currently processing. This may happen because of a branch mispredict where the Istream has been prefetching from one branch and has to change over to the other. The Mbox then aborts all outstanding IREADs so that a new IREAD can begin.

When the Cbox receives the abort signal, the read in question may be anywhere in the Cbox read sequence. The exact action taken depends on where the read is, as shown in Table 13–21.

Table 13–21: Cbox Action Upon Receiving M%ABORT_CBOX_IRD_H

State of the IREAD	Action Taken by the Cbox
No IREAD outstanding	No action taken.
IREAD_LATCH valid but not started	Clear the IREAD_LATCH so the request will not be started.
IREAD_LATCH valid and hit calculation in progress	Abort the hit calculation immediately. This frees the tag store and data RAMs for another request.
IREAD_LATCH valid and read hit in progress	Abort the data RAM sequence immediately. The tag store and data RAMs are freed up for another request.
IREAD valid in FILL_CAM	Clear the TO_MBOX bit in the FILL_CAM entry. When the fill data returns from memory, validate it in the backup cache but don't send the data to the Mbox.
IREAD fill data in CM_OUT_LATCH or FILL_DATA_PIPEs	Clear the entry containing IREAD data so that the data is not returned to the Mbox.

Figure 13–16 shows an example of timing for the Cbox abort response. In cycle 1, **M%ABORT_CBOX_IRD_H** is asserted during phase 2. The Cbox is ready to drive the I_CF command and **B%S6_DATA_H** during phase 4. The assertion of **M%ABORT_CBOX_IRD_H** prevents both of those actions.

The next IREAD may appear two cycles after the abort.

Figure 13–16: M%ABORT_CBOX_IRD_H Timing

```
|       cycle 1         |       cycle 2         |       cycle 3         |
|+++++|+++++|+++++|+++++|+++++|+++++|+++++|+++++|+++++|+++++|+++++|+++++|
|                       |                       |                       |
         ^           ^  ^                                              ^
         |           |  |                                              |
         |           |  |                                              Mbox may send next IREAD
         |           |  B%S6_DATA_H for I_CF not driven due to abort
         |           C%CBOX_CMD_H=I_CF not driven due to abort
        M%ABORT_CBOX_IRD_H
```

13.4.2 ECC Datapaths[1]

The backup cache tag store and data store are both protected by error-detect-and-correct codes (ECC). ECC was chosen for its capability to correct errors because the cache is writeback and may contain the only copy of data in the system.

The codes employed detect and/or correct the following errors:

1. Detect and correct single-bit errors.
2. Detect double-bit errors.
3. Detect three and four bit failures if within one nibble.
4. Detect some addressing failures.
5. Detect all-zero's failure on all protected bits.
6. Detect all-one's failure on all protected bits.

In general, ECC works as follows: Some number of check bits are generated. Each check bit is parity calculated over some subset of the data bits to be protected. The data bits and the check bits together are known as a code word.

When data is written, the check bits are calculated and stored with the data; when data is read the check bits are regenerated and compared against the stored check bits. The result of the comparison is called the syndrome; if it is all zeros there is no error. The syndrome is passed through the syndrome decoder, which decodes one of N states. Each of the N states corresponds to one of the data or check bits being protected by ECC.

If the syndrome does not decode successfully, the error is recognized as uncorrectable. If it does decode successfully, the output of the decoder indicates which bit is in error and that bit is inverted to achieve the data correction.

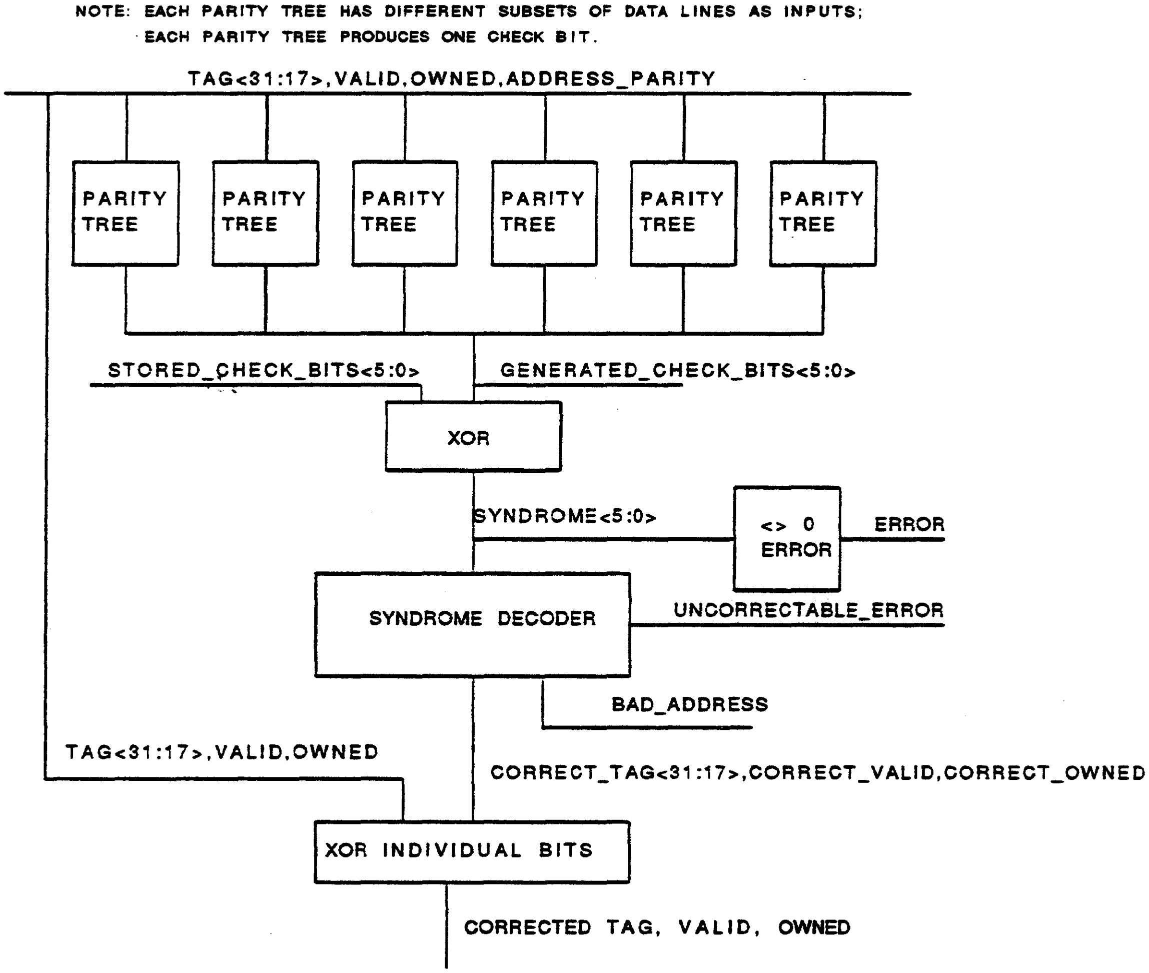
13.4.2.1 Backup Cache Tag Store ECC

Figure 13–17 shows a block diagram of the ECC datapath for tag store ECC. **P%TS_TAG_H<31:17>**, **P%TS_OWNED_H**, and **P%TS_VALID_H** are protected directly by ECC.

When the tag store is written, the generated check bits are written into the RAMs with the tag, valid and owned bits. When the tag store is read, the check bits are regenerated on the stored tag, valid and owned bits and compared with the stored check bits. The result of the comparison is the syndrome, which decodes to tell the hardware which bit is in error.

[1] see Steve Elkind's memo of 31 January 1989, ECC Codes for NVAX Bcache, for more detail about the codes chosen.

Figure 13–17: Tag Store ECC Block Diagram


NOTE: EACH PARITY TREE HAS DIFFERENT SUBSETS OF DATA LINES AS INPUTS;
EACH PARITY TREE PRODUCES ONE CHECK BIT.
TAG<31:17>,VALID,OWNED,ADDRESS_PARITY
PARITY TREE
PARITY TREE
PARITY TREE
PARITY TREE
PARITY TREE
PARITY TREE
STORED_CHECK_BITS<5:0>
GENERATED_CHECK_BITS<5:0>
XOR
SYNDROME<5:0>
<> 0 ERROR
ERROR
SYNDROME DECODER
UNCORRECTABLE_ERROR
BAD_ADDRESS
TAG<31:17>,VALID,OWNED
CORRECT_TAG<31:17>,CORRECT_VALID,CORRECT_OWNED
XOR INDIVIDUAL BITS
CORRECTED TAG, VALID, OWNED

A failure in addressing the RAMs is covered indirectly in the following way: When an entry is written into the tag store, even parity is generated on the on-chip version of P%TS_INDEX_H<20:5>. This is the address parity bit. (Those bits of P%TS_INDEX_H<20:17> which are not required to address the RAMs, based on cache size selection, are zero'd during parity generation.) The address parity bit, P%TS_TAG_H<20:5>, P%TS_OWNED_H, and P%TS_VALID_H are all used in generating the check bits to be stored. The address parity bit itself is not actually stored.

Sh'd be 31:17

When an entry is read from the tag store, parity on P%TS_INDEX_H<20:5> is recalculated and used in the regeneration of the check bits, which are then compared with the stored check bits. If there was an addressing failure in either reading or writing the RAMs, and the regenerated check bits do not match the stored check bits, the output of the syndrome decoder indicates that the address bit is in error. Addressing failures are only detected if the failure was such that incorrect parity is produced from the address.

ECC det tag

ECC tag pred exp

The ECC datapath makes a "predictive" ECC possible which is used in the hit calculation. While the tag RAMs are being accessed, the six predictive ECC check bits are calculated on the expected tag, valid, and owned bits. This predictive ECC is then compared with the actual ECC check bits read from the TAG RAMs during the hit calculation. In this way, an ECC error prevents a cache hit, so that a hit is never detected and then rescinded due to an error.

The code used for tag ECC is shown in Figure 13–18. The check bit which is marked with a "1" in each row is generated by a parity tree whose inputs are the Tag, Valid, Owned, and AP (address parity) bits which are marked with a "1" in that row.

Figure 13–18: Tag Store Error Correcting Code Matrix

even parity on TS INDEX<20:5>

	generated check bits								tag bits																
Syndrome	C0	C1	C2	C3	C4	C5	O	V	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	AP
S0	1	0	0	0	0	0	1	0	0	0	1	0	1	1	1	1	0	1	0	1	1	0	1	0
S1	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	1	1	1
S2	0	0	1	0	0	0	1	1	1	1	1	0	0	0	1	1	1	0	0	0	1	0	1	1
S3	0	0	0	1	0	0	1	0	1	0	0	1	0	1	0	1	1	1	1	0	0	1	1	1
S4	0	0	0	0	1	0	1	1	1	1	0	1	1	0	1	1	0	1	1	1	1	0	0	1
S5	0	0	0	0	0	1	1	0	0	1	1	0	1	1	0	0	1	0	1	0	1	1	1	1
	nibble 0				nibble 1				nibble 2				nibble 3				nibble 4				^			^

nibble 5, three bits only

not stored

Even parity - C0, C2, C3, C5
Odd parity - C1, C4
Sn = (generated Cn) XOR (stored Cn)

In a tag store read operation, a non-zero syndrome indicates an error. If the syndrome generated matches one of the columns in the matrix, the error is correctable and the matching column indicates the bit to be corrected. For example, if syndrome<5:0> equals 011100(BIN), then tag bit <31> must be inverted to correct the problem. Any syndrome value which is non-zero and does not match a column in the matrix indicates an uncorrectable error.

This code has the property that if any three or four bits in one nibble are in error, the syndrome produced will not match any matrix column. This means that an uncorrectable error will be flagged for a single 4-bit-wide RAM failure. It does not necessarily protect against single RAM failures if 8-bit-wide RAMs are used.

NOTE

Nibble protection only works if the bits in each nibble shown in the matrix are physically stored in the same RAM chip. The board designer must ensure that this is the case.

Odd parity is used for check bits 1 and 4 to protect against the all-zeros failure mode. Otherwise, all-zeros would be a valid code word. The choice of odd and even parity bits prevents all-ones from being a valid code word as well.

13.4.2.2 Backup Cache Data Store ECC

Figure 13–19 shows a block diagram of the ECC datapath for data ram ECC. **P%DR_DATA_H<63:0>** are protected directly by ECC. Address failure is covered indirectly in the same manner as it is covered on the tag store. When data is written into the data RAMs, parity is generated on the on-chip version of **P%DR_INDEX_H<20:3>** and used as an additional data bit in generating the check bits to be stored. The address parity bit is not actually stored. When an entry is read from the data RAMs, parity on **P%DR_INDEX_H<20:3>** is recalculated and used in the regeneration of the check bits, which are then compared (XOR'd) with the stored check bits to produce the syndrome for the transaction. (If a cache size is selected which does not use some or all of **P%DR_INDEX_H<20:17>**, those bits are zero'ed during the parity calculation.) In many cases an address failure is detected because the check bits will not match and an error is flagged.

The syndrome is used to calculate whether there was an error, and if so, and it was a correctable error, the syndrome tells which bit needs to be corrected.

The code used for data ECC is shown in Figure 13–20. The check bit (C) which is marked with a "1" in each row is generated by a parity tree whose inputs are the data bits marked with a "1" in that row.

As in tag store ECC, any syndrome value which is non-zero and does not match a column in the table indicates an uncorrectable error. A correctable error is indicated when the syndrome matches a column in the table. For example, data bit <44> must be inverted to correct the error if syndrome<7:0> equals 10000011(BIN).

This code has the property that if any three or four bits in one nibble are in error, the syndrome produced will not match any matrix column. This means that an uncorrectable error will be flagged for a single 4-bit-wide RAM failure.

NOTE

> Nibble protection only works if the bits in each 4-bit nibble shown in the matrix are physically stored in the same RAM chip. The board designer must ensure that this is the case. If x8 RAMs are used, the failure of an entire RAM chip is not protected by the code.

Odd parity is used in check bits 3 and 7 to prevent all-ones and all-zeros from being valid code words.

Figure 13–19: Data RAM ECC Block Diagram

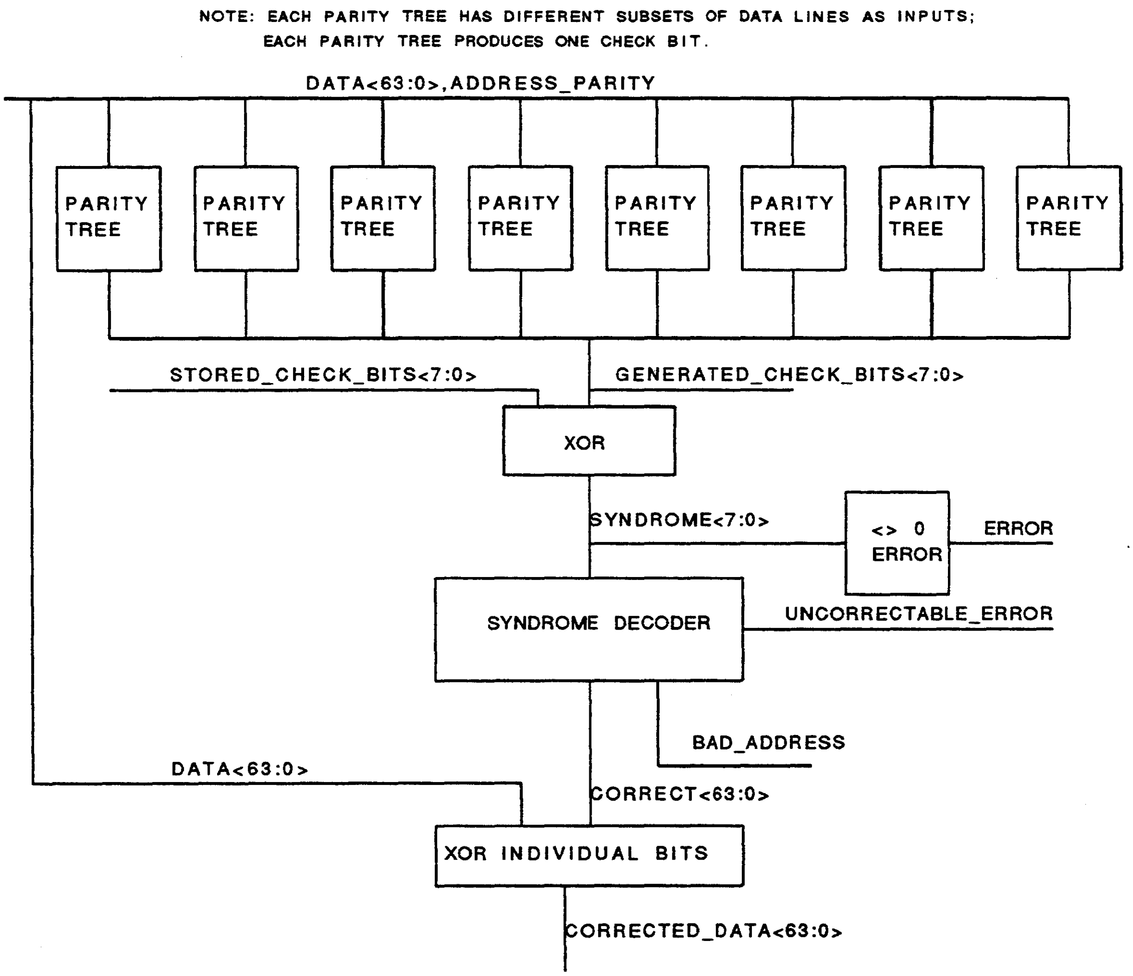


Figure 13–20: Backup Cache Data Store Error Correcting Code Matrix

```
DDDD DDDD DDDD DDDD DDDD DDDD DDDD DDDD DDDD DDDD DDDD DDDD DDDD DDDD DDDD D          DDD
0123 4567 8911 1111 1111 2222 2222 2233 3333 3333 4444 4444 4455 5555 5555 6CCC CCCC C666 A
            01 2345 6789 0123 4567 8901 2345 6789 0123 4567 8901 2345 6789 0012 3456 7123 P

1101 0001 0001 0011 1110 1101 1101 1100 0011 0010 0010 1101 1110 1010 1110 0100 0000 0000 1   S0
1010 0010 0010 0100 1011 1010 1010 1011 0100 0100 0100 1011 1011 1101 1101 1010 0000 0000 1   S1
0111 1100 1100 1000 0101 0111 0111 0111 1000 1001 1001 0110 0101 0111 0011 0001 0000 0000 1   S2
0100 1111 1111 0001 0010 0001 0001 0001 0001 0001 0001 0001 1111 0001 1111 1000 1000 0100 0   S3
1111 0101 1001 0111 0001 0100 0100 1000 0111 1101 1101 0010 0110 1000 0110 0000 0100 0010 1   S4
1111 1010 0110 1101 0100 1000 1000 0010 1101 1011 1011 0100 1001 0100 1001 0000 0010 0001 1   S5
1011 1111 0000 0100 1000 1101 0010 0100 1011 1000 0111 0111 1111 1101 0000 1000 0001 0111 0   S6
0000 1111 0000 1111 0000 1111 0000 0000 0000 1111 0000 1111 0000 1111 1111 0000 0000 1111 0   S7

AP is not stored in the RAMs.

Even parity - C0, C1, C2, C4, C5, C6
Odd parity  - C3, C7
Sn = (generated Cn) XOR (stored Cn)
```

13.4.3 The BIU

The BIU contains the NDAL pads, the NDAL_IN_QUEUE, the WRITEBACK_QUEUE, the NON_WRITEBACK_QUEUE, the BIU IPRs, and timeout counters for outstanding reads. The pads are run on the NDAL clocks, while the rest of the BIU is run on the NVAX internal clocks.

The BIU IPRs are described in Section 13.5; the rest of the BIU is described here.

13.4.3.1 NDAL_IN_QUEUE

The NDAL_IN_QUEUE receives fill data and cache coherency requests from the NDAL. It consists of 8 quadword entries for fill data and two entries for cache coherency addresses. Queue control ensures that each entry is processed in the order in which it was received, so that fills and coherency requests are always processed in order.

The BIU also uses the NDAL_IN_QUEUE mechanisms to inform the FILL_CAM that a read transaction was not acknowledged or timed out before the fill data returned.

The 8 fill data slots ensure that there is always room in the queue for CPU fill data being returned from memory.

The two cache coherency slots are managed through the assertion of **P%CPU_SUPPRESS_L**. The BIU asserts **P%CPU_SUPPRESS_L** on the NDAL to prevent the cache coherency slots from overflowing. When one slot fills, the BIU must assert **P%CPU_SUPPRESS_L** immediately because the next NDAL cycle may be another cache coherency cycle, which would fill both queue slots. This means that two cache coherency commands may be received only if they are on back-to-back cycles; if only one is received, **P%CPU_SUPPRESS_L** is asserted until that one is handled by the Cbox. This should happen quickly since the NDAL_IN_QUEUE is serviced by the Cbox as the highest priority task.

The BIU deasserts **P%CPU_SUPPRESS_L** when it is able to accept more cache coherency commands. Note that fill data may always return, whether or not **P%CPU_SUPPRESS_L** is asserted, as there is always room in the queue for fill data.

The NDAL_IN_QUEUE is loaded with a valid entry to be processed by the Cbox (1) whenever there is a valid memory address cycle on the NDAL, where **P%ID_H<2:1>** is not equal to the NVAX ID, and which is accompanied by one of the following commands: IREAD, DREAD, OREAD, or WRITE (cache coherency cycles); (2) whenever there is a Read Data Return or Read Data Error cycle on the NDAL and **P%ID_H<2:1>** indicates that it belongs to the CPU; (3) when the BIU detects NACK for an outgoing read; (4) when a read transaction times out before data is returned.

The fields of the two portions of the NDAL_IN_QUEUE are shown in Table 13–22.

Table 13–22: NDAL_IN_QUEUE Fields

Field	Purpose
Fill entries	
VALID	Indicates that the entry contains valid information.
DATA<63:0>	Fill data being returned.
Cache coherency entries	
VALID	Indicates that the entry contains valid information.
ADDRESS<31:5>	The address of the cache coherency request.

When the BIU sends a transaction from the NDAL_IN_QUEUE to the Cbox proper, it is accompanied by one of the commands shown in Table 13–23.

Table 13–23: BIU commands sent to Cbox proper

Command name	Meaning
C_BIU%%NOP_CMD	No operation.
C_BIU%%FILL_0_CMD	Fill for FILL_CAM entry 0.
C_BIU%%FILL_1_CMD	Fill for FILL_CAM entry 1.
C_BIU%%RDE_0_CMD	Read Data Error for FILL_CAM entry 0.
C_BIU%%RDE_1_CMD	Read Data Error for FILL_CAM entry 1.
C_BIU%%NACK_0_CMD	No NDAL acknowledgement received for read from FILL_CAM entry 0.
C_BIU%%NACK_1_CMD	No NDAL acknowledgement received for read from FILL_CAM entry 1.
C_BIU%%TIMO_0_CMD	Read from FILL_CAM entry 0 has timed out.
C_BIU%%TIMO_1_CMD	Read from FILL_CAM entry 1 has timed out.
C_BIU%%INVAL_R_CMD	Cache coherency request resulting from a DREAD or an IREAD on the NDAL.
C_BIU%%INVAL_O_CMD	Cache coherency request resulting from an OREAD or a WRITE on the NDAL.

No address is returned for fills, as the NDAL **P%ID_H<0>** which is returned tells the Cbox which FILL_CAM entry was used for the read address. This information is encoded in the commands in Table 13–23. The Cbox uses the backup cache index from the FILL_CAM to write the correct locations in the tag store and data RAMs.

There are four separate NDAL Read Data Return commands to allow the Cbox to identify the quadwords within the hexaword as they return. The lower two bits of the NDAL command are encoded to represent bits <4:3> of a quadword address. The BIU passes these bits to the CBOX_BIU_INTERFACE, which drives them onto **C_ADC%ABUS_H<4:3>** when the data is driven onto **C_BUS%DBUS_H<63:0>**. The information is then driven to the Bcache and to the Mbox. In this way the correct quadword cache entry is written in both caches.

13.4.3.2 NON_WRITEBACK_QUEUE

All outgoing commands except disown writes pass through the NON_WRITEBACK_QUEUE. When the backup cache is on, the NON_WRITEBACK_QUEUE contains read misses, OREADs due to write misses, and I/O space reads and writes. When the backup cache is off, all transactions except quadword disown writes (which result from WRITE_UNLOCKs) go out through the NON_WRITEBACK_QUEUE.

The NON_WRITEBACK_QUEUE has two entries. The fields of each entry in the queue are shown in Table 13–24.

Table 13–24: NON_WRITEBACK_QUEUE Fields

Field	Purpose
VALID	Indicates that the entry contains valid information.
CMD<3:0>	Specific command being done.
ID0	Identification, driven onto **P%ID_H<0>**, for outgoing reads only.
ADDRESS<31:0>	Address of the outgoing command.
LENGTH<63:62>	Length of the outgoing command.
BYTE_ENABLE<47:40>	Byte enable.
DATA<63:0>	Data, used if the outgoing command is a write.

The format of the address field corresponds to that of an address cycle on the NDAL, which is described in Section 3.3.4.1.

Writes from this queue are always byte-enabled quadword writes whether to memory space or I/O space.

The NON_WRITEBACK_QUEUE has a backpressure signal so that when it gets full, the Cbox stalls transactions from the Mbox until there is room in the queue to proceed. Fills and cache coherency transactions continue normally.

13.4.3.3 WRITEBACK_QUEUE

The WRITEBACK_QUEUE holds addresses and data for write disowns to memory. It contains two entries, each consisting of address and data for either a hexaword or a quadword disown write.

Table 13–25 shows the fields in the WRITEBACK_QUEUE.

Table 13–25: WRITEBACK_QUEUE Fields

Field	Purpose
VALID	Indicates that the entry contains valid information.
CMD<3:0>	Specific command being done.
ADDRESS<63:0>	Address cycle for the writeback.
DATA0<63:0>	First quadword of writeback data.

Table 13–25 (Cont.): WRITEBACK_QUEUE Fields

Field	Purpose
DATA1<63:0>	Second quadword of writeback data.
DATA2<63:0>	Third quadword of writeback data.
DATA3<63:0>	Fourth quadword of writeback data.
BYTE_ENABLE<7:0>	Byte enable for quadword disown writes.

The format of the address field corresponds to that of an address cycle on the NDAL, which is described in Section 3.3.4.1.

When a disown write is done, the ADDRESS field is first loaded. CMD<3:0> is loaded with the WDISOWN command. Four quadwords of write data are loaded if the transaction is hexaword length; if the transaction is quadword length, one quadword of data is loaded.

All writeback data is read from the data RAMs before the NDAL transaction is started, to simplify error handling. If a quadword of data is read out with an uncorrectable error, the command field sent with that data cycle is changed from WDATA to BADWDATA.

The WRITEBACK_QUEUE always takes priority over the NON_WRITEBACK_QUEUE in driving the NDAL.

The WRITEBACK_QUEUE backpressures the Cbox control when it gets full, causing the following:

1. All reads from the Mbox are prevented.
2. All writes from the Mbox are prevented.
3. All fills are prevented.
4. All cache coherency lookups are prevented.

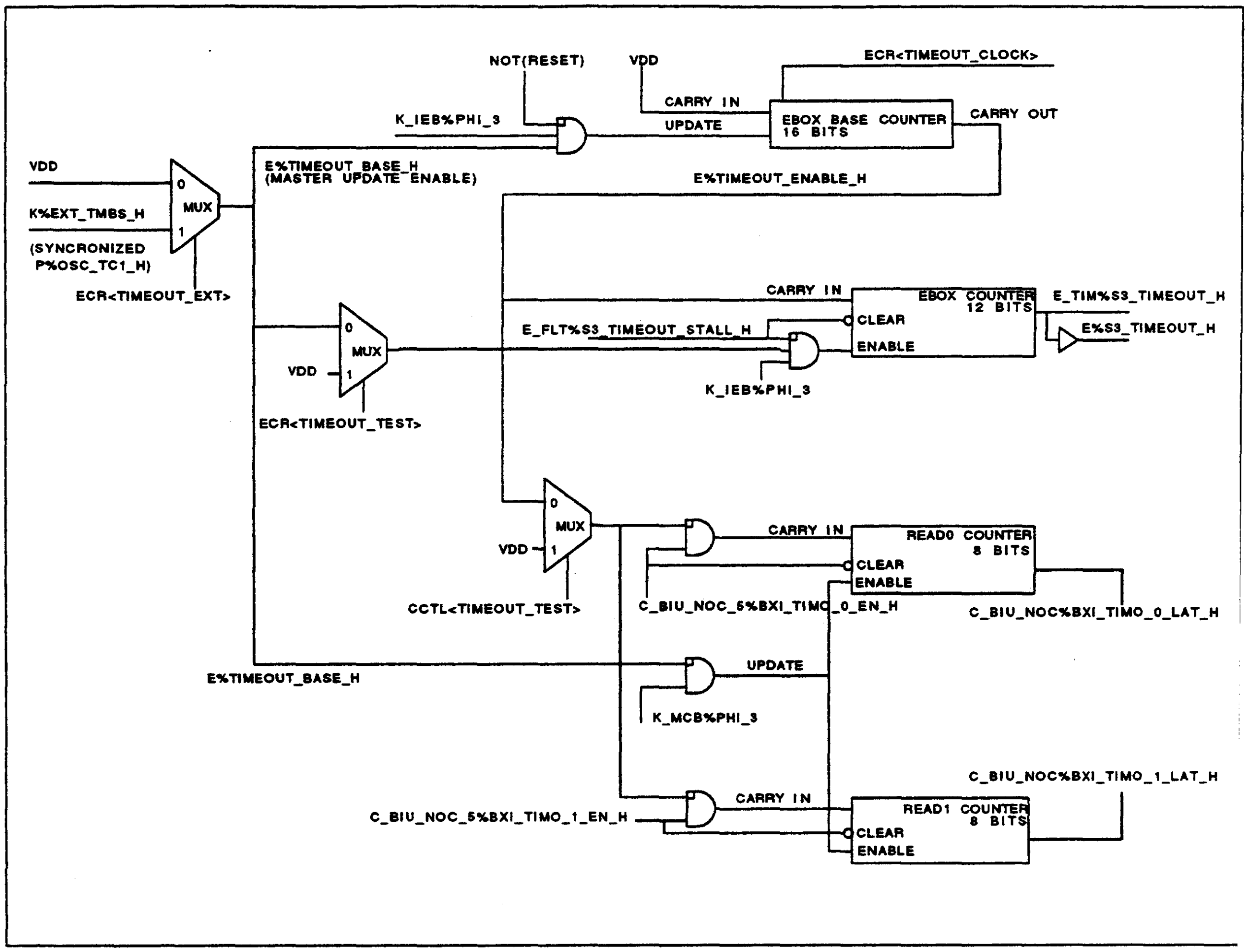
13.4.3.4 Timeout counters

The BIU has two timeout counters, one for each read request which may be outstanding. If all the fills for an outstanding read have not completed when the associated timeout counter expires, the BIU notifies the FILL_CAM of the error and it is handled as described in Chapter 3.

The NVAX timeout counters are shown in Figure 13–21. The Ebox contains the Ebox base counter and the Ebox counter, which counts Ebox stall cycles. The Cbox contains two read counters which, in normal mode, are driven from the Ebox base counter. The Ebox counters are described in detail in Chapter 8.

Three IPR bits control the operation of the timeout counters. When ECR<TIMEOUT_EXT>, ECR<S3_TIMEOUT_TEST>, and CCTL<TIMEOUT_TEST> are all cleared, the counters are in normal mode. When ECR<TIMEOUT_EXT> is set, an external timebase may be used to lengthen the timeout period; when CCTL<TIMEOUT_TEST> is set, the read timeout counters are placed in test mode, under which the read timeout values are shortened; and when ECR<S3_TIMEOUT_TEST> is set, the Ebox counter is put in test mode, under which the S3 timeout value is shortened.

Figure 13–21: NVAX Timeout Counters



In normal mode, the Cbox and the Ebox share the base counter, which is run from the internal NVAX clock. The 12-bit Ebox counter and the 8-bit Cbox read counters are clocked with the global signal, **E%TIMEOUT_ENABLE_H**, which is generated from the 16-bit base counter. In normal mode, **E%TIMEOUT_ENABLE_H** is asserted for one NVAX internal cycle when the Ebox base counter overflows; if an external timebase is used (if ECR<TIMEOUT_EXT> is asserted), **E%TIMEOUT_ENABLE_H** is asserted for one cycle of the external timebase when the counter overflows. **E%TIMEOUT_BASE_H** is always asserted when the timeout counter is in normal mode; if ECR<TIMEOUT_EXT> is asserted, **E%TIMEOUT_BASE_H** is asserted for one NVAX internal cycle when the input clock transitions high.

The timeout values for normal mode are shown in Table 13–26.

Table 13–26: NVAX Timeout Values in Normal Mode

Cycle time	Timeout Granularity	Read timeout[1]	Ebox timeout[1]
10-ns NVAX	655 microseconds	167.117 (minimum) to 167.772 (max) milliseconds	2.6837 (minimum) to 2.68345 (max) seconds
12-ns NVAX	786 microseconds	200.54 (minimum) to 201.327 (max) milliseconds	3.22044 (minimum) to 3.22123 (max) seconds
14-ns NVAX	917 microseconds	233.964 (minimum) to 234.881 (max) milliseconds	3.75718 (minimum) to 3.7581 (max) seconds

[1] The timeout logic is in normal mode when ECR<TIMEOUT_EXT>, CCTL<TIMEOUT_TEST>, and ECR<S3_TIMEOUT_TEST> are all cleared.

Each Cbox read counter is initialized to zero when it is not enabled with either **C_BIU_NOC_5%BXI_TIMO_0_EN_H** or **C_BIU_NOC_5%BXI_TIMO_1_EN_H**, and counts as long as the read is outstanding. If all the fills do not return within the timeout period, the counter overflows and **C_BIU_NOC%BXI_TIMO_0_LAT_H** or **C_BIU_NOC%BXI_TIMO_1_LAT_H** is asserted. As a result, the read is aborted, the timeout counter is reset to zero, and the error is handled as described in Chapter 3.

If a system designer needs to lengthen the timeout values, an external timebase, **K%EXT_TMBS_H**, can be selected by setting ECR<TIMEOUT_EXT> in the Ebox control register. In this case, the Ebox base counter is clocked with the external timebase, which enters the chip through P%OSC_TC1_H.

The counters are configurable for use at chip test and at power-up test. At chip test and/or during power-up diagnostics, the read counters can be tested in the following way: Set CCTL<TIMEOUT_TEST> so that the Cbox counters run off the internal NVAX clock. Clear ECR<S3_TIMEOUT_TEST>. Do a read of a memory or I/O space location which will not respond within the timeout period. A read timeout should occur. This must be done for each timeout counter.

The timeout values for the Cbox and Ebox counters in test mode are shown in Table 13–27.

Table 13–27: NVAX Timeout Values in Test Mode

Cycle time	Timeout Granularity	Read timeout[1]	Ebox timeout[2]
10-ns NVAX	10 nanoseconds	2.55 (minimum) to 2.56 (max) microseconds	40.95 (minimum) to 40.96 (max) microseconds
12-ns NVAX	12 nanoseconds	3.06 (minimum) to 3.072 (max) microseconds	49.14 (minimum) to 49.152 (max) microseconds
14-ns NVAX	14 nanoseconds	3.57 (minimum) to 3.584 (max) microseconds	57.33 (minimum) to 57.344 (max) microseconds

[1] Read timeout test is done under these conditions: ECR<TIMEOUT_EXT> and ECR<S3_TIMEOUT_TEST> cleared; CCTL<TIMEOUT_TEST> set.

[2] Ebox timeout test is done under these conditions: ECR<TIMEOUT_EXT> and CCTL<TIMEOUT_TEST> cleared; ECR<S3_TIMEOUT_TEST> set.

Forcing timeouts cannot be done by reading nonexistant memory or I/O: NDAL designers respond to nonexistant memory and I/O space with either NACK or RDE, which happen well before the timeout counters expire. A timeout can be accomplished in the following way:

1. Do a write or a read-modify-write which causes an OREAD to bring owned data into the backup cache.
2. Do an IPR WRITE to clear the owned bit in the backup cache tag store.
3. Perform another operation which requires ownership in the Bcache. This OREAD will timeout because it won't hit in the backup cache and memory won't respond because it believes the backup cache owns it.
4. Do an IPR WRITE to the Bcache tag store to put it back into the owned state.

The list which follows describes a scenario in which read data takes a long time to return to the Ebox. This case should not approach the Ebox timeout value; it is given to illustrate what can keep data from returning quickly to the Ebox.

1. The Cbox write queue is full.
2. A Dread, call it Dread A, enters the Cbox and has a conflict with the last write queue entry, Write A, which means that the whole write queue must be cleared out before Dread A can proceed.
3. The writes in the write queue all miss in the Bcache, and each one requires a writeback from another CPU which owns the block. As each writeback is done, the data is returned to the Bcache, ownership is passed to the Bcache, and the write queue is emptied of one write. In this scenario, eight writebacks are required before Read A can be processed.
4. After the Oread for Write A reaches the NDAL, an invalidate arrives for A. After the data is returned and Write A is processed, the block will be written back, due to the previous invalidate.
5. Now Dread A will miss in the Bcache, and it will have to wait for another writeback. Eventually this read data will return, and the Ebox gets its data.

DERIVATION OF TIMEOUT VALUES

The timeout values given on the previous pages were derived from NVAX cycles as follows:

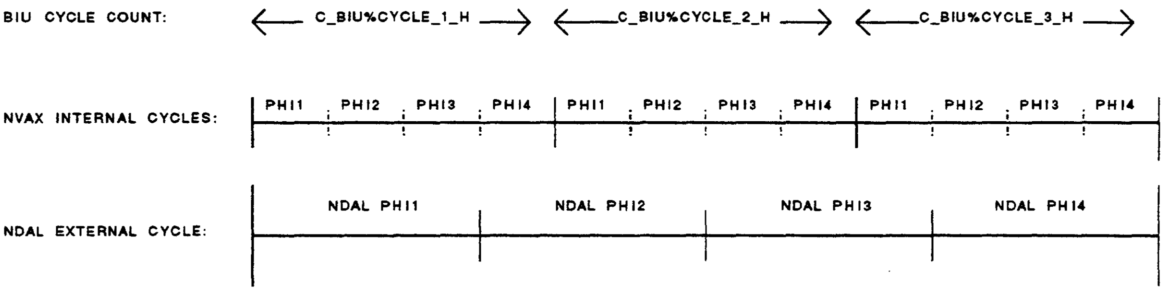
Table 13–28: Derivation of NVAX Timeout Values

NVAX mode	Timeout Granularity (in NVAX cycles)	Read timeout (in NVAX cycles)	Ebox timeout (in NVAX cycles)
Normal	2**16	2**24—2**16 (minimum) to 2**24 (max)	2**28—2**16 (minimum) to 2**28 (max)
Test	1	2**8—1 (minimum) to 2**8 (max)	2**12—1 (minimum) to 2**12 (max)

13.4.3.5 BIU clocking: Relating internal cycles to external cycles

Three NVAX internal cycles take place in the time of one NDAL cycle. The BIU relates internal cycles to external cycles by naming the internal cycles according to where they fall relative to the external cycle. This is shown in Figure 13–22.

Figure 13–22: BIU cycle counts

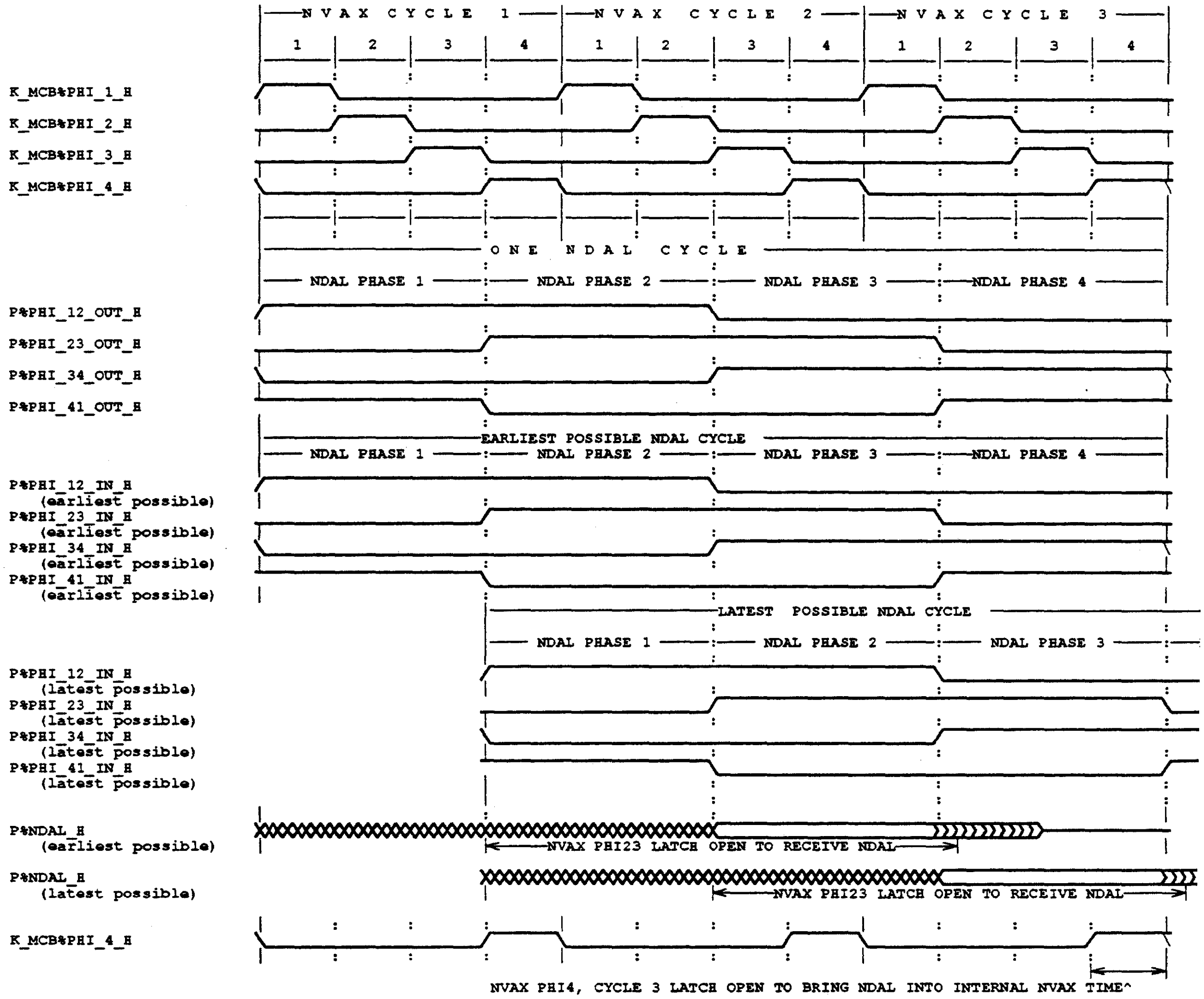


The BIU has a shift register which asserts only one of the signals C_BIU%CYCLE_1_H, C_BIU%CYCLE_2_H, and C_BIU%CYCLE_3_H during any given NVAX cycle. This shift register is initialized properly by K_CE%RESET_H, which comes from the clock section of the chip. During reset, the clock section asserts K_CE%RESET_H during every NDAL phase 4, allowing the BIU to initialize the shift register properly.

Only the NVAX internal clocks are used in the Cbox and BIU, while only the external clocks are used in the pad ring. Through the use of C_BIU%CYCLE_1_H, C_BIU%CYCLE_2_H, and C_BIU%CYCLE_3_H, the BIU is able to properly drive and receive the NDAL to and from the pad ring.

There is a delay in the NDAL clocks as they travel from NVAX to the other NDAL chips and also back to NVAX. The delay from the NVAX output pin, P%PHI12_OUT_H, to the NVAX input pin, P%PHI12_IN_H, may be as little as 0ns or as much as three internal NVAX phases (one NDAL phase). This delay is shown graphically in Figure 13–23.

Figure 13–23: NVAX time relative to NDAL time


NVAX CYCLE 1
NVAX CYCLE 2
NVAX CYCLE 3
1
2
3
4
K_MCB%PHI_1_H
K_MCB%PHI_2_H
K_MCB%PHI_3_H
K_MCB%PHI_4_H
ONE NDAL CYCLE
NDAL PHASE 1
NDAL PHASE 2
NDAL PHASE 3
NDAL PHASE 4
P%PHI_12_OUT_H
P%PHI_23_OUT_H
P%PHI_34_OUT_H
P%PHI_41_OUT_H
EARLIEST POSSIBLE NDAL CYCLE
P%PHI_12_IN_H (earliest possible)
P%PHI_23_IN_H (earliest possible)
P%PHI_34_IN_H (earliest possible)
P%PHI_41_IN_H (earliest possible)
LATEST POSSIBLE NDAL CYCLE
P%PHI_12_IN_H (latest possible)
P%PHI_23_IN_H (latest possible)
P%PHI_34_IN_H (latest possible)
P%PHI_41_IN_H (latest possible)
P%NDAL_H (earliest possible)
NVAX PHI23 LATCH OPEN TO RECEIVE NDAL
P%NDAL_H (latest possible)
NVAX PHI23 LATCH OPEN TO RECEIVE NDAL
K_MCB%PHI_4_H
NVAX PHI4, CYCLE 3 LATCH OPEN TO BRING NDAL INTO INTERNAL NVAX TIME^

K_MCB%PHI_1_H, **K_MCB%PHI_2_H**, **K_MCB%PHI_3_H**, and **K_MCB%PHI_4_H** are the internal NVAX clocks which are used in the Cbox. Figure 13–23 shows that the NDAL clocks at the input pins (**P%PHI12_IN_H**, **P%PHI23_IN_H**, **P%PHI34_IN_H**, and **P%PHI41_IN_H**) may be delayed by up to three internal NVAX phases. The NDAL always operates with respect to the clocks as received at each NDAL driver/receiver, so if the NDAL clocks are delayed, the entire operation of the NDAL is delayed.

The CBOX BIU is designed so that even if the NDAL is operating with three full phases of delay from the internal NVAX clocks, the BIU is able to drive and receive the NDAL properly. For example, **P%NDAL_H<63:0>** are valid at the beginning of NDAL phase 3. NVAX receives this bus using an NDAL latch which is open while **P%PHI23_IN_H** is asserted. The output of this latch is sent from the NVAX pad ring to a latch in the NVAX BIU which is open during NVAX phase 4 of BIU cycle 3. This timing allows 2 NVAX phases of delay to get the signal from the pad ring to the BIU. Thus, the NDAL is properly received for the entire range of possible NDAL delay. Once the NDAL is latched by the phase 4, cycle 3 latch, the BIU operates entirely using the internal NVAX clocks; the NDAL clocks are only used in the pad ring itself.

13.4.4 The FILL_CAM

The FILL_CAM has two entries, each of which is used for an outstanding read to memory or for a DREAD_LOCK in progress. Its depth limits the number of outstanding reads to memory at a time. The fields in each FILL_CAM entry are described in Table 13–29.

Table 13–29: FILL_CAM Fields

Field	Purpose
ADDRESS<31:3>	Quadword-aligned address of the read request.
RDLK	Indicates that a READ_LOCK is in progress.
IREAD	This is an Istream read from the Mbox which may be aborted.
OREAD	This is an outstanding OREAD; block ownership bit should be set when the fill returns.
WRITE	This read was done for a write; write is waiting to be merged with the fill.
TO_MBOX	Data is to be returned to the Mbox.
RIP	READ invalidate pending.
OIP	OREAD invalidate pending.
DNF	Do not fill - data is not to be written into the cache or validated when the fill returns.
RDLK_FL_DONE	Indicates that the last fill for a READ_LOCK arrived.
REQ_FILL_DONE	Indicates that the requested quadword of data was received from the NDAL.
COUNT<1:0>	Counts the number of fill quadwords that have been successfully returned.
VALID	Indicates that the entry contains valid information.

The FILL_CAM backpressures the Cbox control so that if it is full, any read or write request stalls until an entry is free.

When the read miss first occurs and the FILL_CAM entry is loaded, the following bits are cleared: RIP, OIP, RDLK_FL_DONE, and REQ_FILL_DONE. VALID is set and the ADDRESS field is loaded. IREAD, RDLK, OREAD, WRITE, and TO_MBOX are loaded with the correct information. If the cache is off, in ETM, or the miss is for an I/O reference, DNF is set; otherwise it is cleared. COUNT is set to 0 if four fill quadwords are expected; it is set to 3 if only one quadword is expected.

As each fill returns successfully, COUNT is incremented so that when the final fill returns and COUNT=3, the Cbox updates the tag store appropriately.

If an abort request arrives from the Mbox, and the entry is marked IREAD, the TO_MBOX bit is cleared. When the data returns, it will be written into the backup cache (if DNF is not set) but it will not be sent to the Mbox.

If a coherence request arrives from the NDAL which matches the address of a FILL_CAM entry, RIP or OIP may be set. Table 13–30 shows when each is set.

Table 13–30: Cbox Response to Coherence Transactions to FILL_CAM Entries

State of OREAD bit	Coherence Transaction	Cbox Action
OREAD set or clear	OREAD, READLK, any write	Set OIP. Send invalidate immediately to the Pcache.
OREAD set	DREAD, IREAD	Set RIP.
OREAD clear	DREAD, IREAD	Take no action.

When all the fills for an outstanding miss have completed, a cache coherence transaction is initiated if either RIP or OIP is set and DNF is not set. This is done immediately after the fill and the validate of the cache are done, and cannot be interrupted by any other transaction.

When a WRITE_UNLOCK completes successfully and RIP or OIP is set, the cache coherence transaction is initiated immediately.

There are several error cases where RIP or OIP may be set, indicating the need for a cache coherence transaction, but the Cbox will not perform the transaction, possibly causing the system element to time out. These cases are as follows:

1. The fill sequence fails by ending in RDE or timeout. If the fill was meant for the Pcache and ends in an error, the Pcache invalidates itself.
2. A READ_LOCK sequence does not conclude with a WRITE_UNLOCK but with a write-one-to-clear to the RDLK bit in CEFSTS.

As shown in the table above, when an ownership-type coherence transaction arrives, an invalidate is sent immediately to the Pcache and OIP is set. When the cache coherence transaction to the tag store is processed immediately after all the fills have arrived, a second invalidate will be issued to the Pcache, although it is not strictly necessary. The first invalidate is sent immediately so that the block in the Pcache is invalidated as soon as possible, to prevent the stale data from being accessed before the rest of the fills return.

13.4.4.1 Block-conflict in the FILL_CAM

Every new read or write from the Mbox is checked against valid FILL_CAM entries so that any transaction with a cache block conflict is stalled until all the fills return for the outstanding read, clearing the conflicting FILL_CAM entry. In this way, cache accesses to a block with an outstanding fill are prevented.

When the cache is off or in ETM, writes are not checked for block conflict but are sent immediately to memory.

13.4.4.2 The FILL_CAM and DREAD_LOCKs

Each DREAD_LOCK from the Mbox is held in the FILL_CAM until the associated WRITE_UNLOCK completes, regardless of whether the read hits or misses in the backup cache. Only one DREAD_LOCK/WRITE_UNLOCK transaction is in progress at a time. A DREAD_LOCK which does not produce an owned hit in the backup cache results in an OREAD on the NDAL to gain ownership of the block so that the write can be done.

By holding the DREAD_LOCK address in the FILL_CAM from the time the DREAD_LOCK starts until the WRITE_UNLOCK completes, the Cbox prevents the block from being written back to memory during that time. This guarantees that the DREAD_LOCK/WRITE_UNLOCK sequence will not be interrupted by another CPU requesting ownership of the block. The CPU depends on no other state in memory once the OREAD is done in order to complete the WRITE_UNLOCK, so no deadlock can arise.

Every new transaction is checked against the FILL_CAM to ensure that the block is not inaccessable due to an outstanding fill or DREAD_LOCK.

If either RDLK bit is set in the FILL_CAM, IREADs and DREADs are not processed. Incoming fills and coherency transactions continue normally; and the WRITE_QUEUE is serviced normally. The only transaction which should appear in the WRITE_QUEUE (when either RDLK bit is set) is the WRITE_UNLOCK corresponding to the READ_LOCK.

The one exception to this is when the READ_LOCK terminates in an error. In this case an IPR_WRITE to CEFSTS is the next transaction which appears in the WRITE_QUEUE. Specifically, a write-one-to-clear of the RDLK bit in CEFSTS has the side effect of clearing any RDLK bit in the FILL_CAM which is set. If one of the RDLK bits is cleared in the FILL_CAM, hardware also clears the corresponding valid bit, freeing the entry for a new transaction.

When the RDLK bit is cleared by a normal WRITE_UNLOCK, a cache coherency transaction is initiated if RIP or OIP was set on the entry. RIP and OIP are ignored when the RDLK bit is cleared by the "IPR write unlock" method.

13.5 Cbox Internal Processor Registers

The processor registers that are implemented by the NVAX Cbox are logically divided into three groups, as follows:

- Normal—Those IPRs that address individual registers in the NVAX CPU chip or system environment.
- Bcache tag IPRs—The read-write block of IPRs that allow direct access to the Bcache tags.
- Bcache deallocate IPRs—The write-only block of IPRs by which a Bcache block may be deallocated.

Each group of IPRs is distinguished by a particular pattern of bits in the IPR address, as shown in Figure 13–24.

Figure 13–24: IPR Address Space Decoding as seen by Software

```
Normal IPR Address

 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|       SBZ          | 0|                      SBZ                      |       IPR Number      |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

Bcache Tag IPR Address

 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|       SBZ          | 1| 0| 0| x|                Bcache Tag Index               |     SBZ      |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

Bcache Deallocate IPR Address

 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|        SBZ         | 1| 0| 1| x|          Bcache Tag Deallocate Index          |     SBZ      |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

The numeric range for each of the three groups is shown in Table 13–31.

Table 13–31: IPR Address Space Decoding

IPR Group	Mnemonic[1]	IPR Address Range (hex)	Contents
Normal		00000000..000000FF	256 individual IPRs.
Bcache Tag	BCTAG	01000000..011FFFE0[2]	64k Bcache tag IPRs, each separated by 20(hex) from the previous one.
Bcache Deallocate	BCFLUSH	01400000..015FFFE0[2]	64k Bcache tag deallocate IPRs, each separated by 20(hex) from the previous one.

[1] The mnemonic is for the first IPR in the block

[2] Unused fields in the IPR addresses for these groups should be zero. Neither hardware nor microcode detects and faults on an address in which these bits are non-zero. Although non-contiguous address ranges are shown for these groups, the entire IPR address space maps into one of the these groups. If these fields are non-zero, the operation of the CPU is UNDEFINED.

NOTE

> The address ranges shown above are those used by the programmer. When processing normal IPRs, the microcode shifts the IPR number left by 2 bits for use as an IPR command address. This positions the IPR number to bits <9:2> and modifies the address range as seen by the hardware to 0..3FC, with bits <1:0>=00. No shifting is performed for the other groups of IPR addresses.

Because of the sparse addressing used for IPRs in groups other than the normal group, valid IPR addresses are not separated by one. Rather, valid IPR addresses are separated by 20(hex). For example, the IPR address for Bcache tag 0 is 01000000 (hex), and the IPR address for Bcache tag 1 is 01000020 (hex). In this group, bits <4:0> of the IPR address are ignored, so IPR numbers 01000001 through 0100001F all address Bcache tag 0.

Processor registers in all groups except the normal group are processed entirely by the NVAX CPU chip and will never appear on the NDAL. This is also true for a number of the IPRs in the normal group. IPRs in the normal group that are not processed by the NVAX CPU chip are converted into I/O space references and passed to the system environment via a read or write command on the NDAL.

The processor registers implemented by the NVAX Cbox are are shown in Table 13–32.

Table 13–32: Cbox Processor Registers

		Number			Cbox	Cbox
Register Name	Mnemonic	(Dec)	(Hex)	Type	Loc[1]	Addr[2]
Cbox Control Register	CCTL	160	A0	RW	Abus	280
Reserved for Cbox		161	A1			
Bcache Data ECC	BCDECC	162	A2	W	Dbus	288
Bcache Error Tag Status	BCETSTS	163	A3	RW	Abus	28C
Bcache Error Tag Index	BCETIDX	164	A4	R	Abus	290
Bcache Error Tag	BCETAG	165	A5	R	Abus	294
Bcache Error Data Status	BCEDSTS	166	A6	RW	Dbus	298
Bcache Error Data Index	BCEDIDX	167	A7	R	Abus	29C
Bcache Error Data ECC	BCEDECC	168	A8	R	Dbus	2A0
Reserved for Cbox		169	A9			
Reserved for Cbox		170	AA			
Fill Error Address	CEFADR	171	AB	R	Abus	2AC
Fill Error Status	CEFSTS	172	AC	RW	Abus	2B0
Reserved for Cbox		173	AD			
NDAL Error Status	NESTS	174	AE	RW	BIU	2B8
Reserved for Cbox		175	AF			
NDAL Error Output Address	NEOADR	176	B0	R	BIU	2C0
Reserved for Cbox		177	B1			
NDAL Error Output Command	NEOCMD	178	B2	R	BIU	2C8
Reserved for Cbox		179	B3			
NDAL Error Data High	NEDATHI	180	B4	R	BIU	2D0
Reserved for Cbox		181	B5			
NDAL Error Data Low	NEDATLO	182	B6	R	BIU	2D8
Reserved for Cbox		183	B7			
NDAL Error Input Command	NEICMD	184	B8	R	BIU	2E0
Reserved for Cbox		185	B9			
Reserved for Cbox		186	BA			
Reserved for Cbox		187	BB			
Reserved for Cbox		188	BC			
Reserved for Cbox		189	BD			
Reserved for Cbox		190	BE			
Reserved for Cbox		191	BF			
Bcache Tag (01000000 - 011FFFE0 HEX)	BCTAG			RW	Abus	

[1] Each Cbox IPR is located in the Cbox Abus datapath, the Cbox Dbus datapath, or the Cbox BIU datapath.

[2] The address given is as it is seen in the Cbox, after microcode has shifted the software address left by two bits.

Table 13–32 (Cont.): Cbox Processor Registers

Register Name	Mnemonic	Number (Dec)	(Hex)	Type	Cbox Loc[1]	Cbox Addr[2]
Bcache Deallocate (01400000 - 015FFFE0 HEX)	BCFLUSH			W	Abus	

[1] Each Cbox IPR is located in the Cbox Abus datapath, the Cbox Dbus datapath, or the Cbox BIU datapath.

[2] The address given is as it is seen in the Cbox, after microcode has shifted the software address left by two bits.

IPRs in the system and in the Cbox are accessed through IPR_READs and IPR_WRITEs from the Mbox to the Cbox. When the Cbox recognizes a valid IPR_READ on M%S6_CMD_H<4:0>, it loads the read into the DREAD_LATCH to be processed. The Mbox guarantees that only one DREAD or IPR_READ may be outstanding at a time, so that the DREAD_LATCH will not be overwritten. A valid IPR_WRITE is loaded into the WRITE_PACKER and proceeds immediately to the WRITE_QUEUE.

All IPR reads and writes to the Cbox flush the WRITE_QUEUE before completing. Any IPR_READ sets DWR_CONFLICT bits in all valid entries in the WRITE_QUEUE so that any preceding writes of any kind will complete before the IPR_READ. An IPR_WRITE is placed in the WRITE_QUEUE after the preceding writes so that the ordering takes place naturally. If a read arrives after the IPR_WRITE and before it has been processed, the WRITE_QUEUE conflict bits are set so that the WRITE_QUEUE takes priority over the read.

If the IPR_READ addresses one of the Cbox registers, the Cbox returns the data from the register through the CM_OUT_LATCH, in the usual way that it would return data for a read hit. The only difference is that it returns just one quadword or less of data, rather than the usual 4 quadwords. The Cbox asserts C%LAST_FILL_H so the Mbox does not expect any more fills.

If a write-only Cbox register is read, the Cbox returns UNPREDICTABLE data. Reading an unimplemented Cbox register returns UNPREDICTABLE data; if an unimplemented register is written, the write is discarded by the Cbox and normal operation continues.

If the Cbox receives an IPR access to a legal IPR address which is not within the Cbox block of IPR addresses, it converts it to an I/O space read or write. The Cbox merges the IPR address with E1000000 hex, effectively adding the base I/O space address of the IPR block to the IPR address. This is done in hardware by forcing bits <31:29> and bit <24> to 1's. (The other upper bits are expected to be received as zero's.)

From this point on, the transaction is treated as an I/O space transaction by the Cbox. It sends the request off-chip to the NDAL through the NON_WRITEBACK_QUEUE. When the fill data returns, the data is returned to the Mbox but is not cached by the Cbox. I/O space reads and writes are never cached in the primary cache or the backup cache.

NVAX BCACHE -
2 misses can be
outstanding to
memory.

13.5.1 Cbox Control IPR (CCTL)

CCTL is a read/write register which contains bits controlling the behavior of the Cbox. The bits are detailed in Figure 13–25 and Table 13–33.

Figure 13–25: IPR A0 (hex), CCTL

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  | x| x| x| x| x| x| x| x| x| x| x| x| x|  |     |        |  |  |  |  |  |     |     |  |  | :CCTL
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
  |  |                                         |   |       |     |  |  |  |  |   |     |    |  |
  |  |                                         |   |       |     |  |  |  |  |   |     |    |  \-ENABLE
  |  \-SW_ETM                                  |   |       |     |  |  |  |  |   |     |    \-TAG_SPEED
  \-HW_ETM                                     |   |       |     |  |  |  |  |   |     |
                                               |   |       |     |  |  |  |  |   |     \-DATA_SPEED
                                               |   |       |     |  |  |  |  |   \-SIZE
                                               |   |       |     |  |  |  |  |
                                               |   |       |     |  |  |  |  \-FORCE_HIT
                                               |   |       |     |  |  |  \-DISABLE_ERRORS
                                               |   |       |     |  |  \-SW_ECC
                                               |   |       |     |  \-TIMEOUT_TEST
                                               |   |       |     \-DISABLE_PACK
                                               |   |       \-PM_ACCESS_TYPE
                                               |   \-PM_HIT_TYPE
                                               \-FORCE_NDAL_PERR
```

Table 13–33: CCTL Field Descriptions

Name	Extent	Type	Description
ENABLE	0	RW,0	Turns the bcache on and off.
TAG_SPEED	1	RW,0	Controls time to access the tag RAMs.
DATA_SPEED	3:2	RW,0	Controls time to access the data RAMs.
SIZE	5:4	RW,0	Selects one of four backup cache sizes.
FORCE_HIT	6	RW,0	Forces memory reads and writes to hit in the backup cache.
DISABLE_ERRORS	7	RW,0	Disables all backup cache ECC errors.
SW_ECC	8	RW,0	Enables use of ECC check bits as given by software for the tag store and data RAMs.
TIMEOUT_TEST	9	RW,0	Puts the Cbox read timeout counters into test mode.
DISABLE_PACK	10	RW,0	Disables the Cbox write packer.
PM_ACCESS_TYPE	13:11	RW,0	Selects type of Bcache access for the performance monitoring hardware.
PM_HIT_TYPE	15:14	RW,0	Selects type of Bcache hit for the performance monitoring hardware.
FORCE_NDAL_PERR	16	RW,0	Forces a parity error in the command field of the next outgoing NDAL transaction.
SW_ETM	30	RW,0	Used by software to put the backup cache into ETM.
HW_ETM	31	WC	Used by hardware to put the backup cache into ETM.

13.5.1.1 ENABLE

When ENABLE = 1, the backup cache is enabled for operation. When ENABLE=0, the backup cache is off and all references are treated as misses and are not looked up in the backup cache. When the backup cache is off, FORCE_HIT, SW_ETM and HW_ETM are ignored. Reset clears this bit so that the Bcache is off when the chip is reset.

13.5.1.2 TAG_SPEED

The Cbox provides this bit to select the speed of the tag rams. Table 13–34 shows the relationship of the value of TAG_SPEED and the access time of the tag RAMs, given in NVAX cycles. This is the total RAM access time including internal Cbox processing time. For information on the actual cache ram access times required, see Section 13.3.1. Reset clears this bit so that the tag access repetition rate is 3 cycles when the chip is reset.

Table 13–34: TAG_SPEED

	tag read	tag write	
TAG_SPEED	rep rate	rep rate	comments
0	3 cycles	3 cycles	
1	4 cycles	4 cycles	may not be used when DATA_SPEED=00

13.5.1.3 DATA_SPEED

The Cbox provides this bit to select the speed of the data rams. Table 13–35 shows the relationship of the value of DATA_SPEED and the access time of the data RAMs, given in NVAX cycles. This is the total RAM access time including internal Cbox processing time. For information on the actual cache ram access times required, see Section 13.3.1. Reset clears these bits so that the data read rep rate is 2 cycles when the chip is reset.

Table 13–35: DATA_SPEED

	data read	data write	
DATA_SPEED<1:0>	rep rate	rep rate	comments
00	2 cycles	3 cycles	may not be used when TAG_SPEED=1
01	3 cycles	4 cycles	
10	4 cycles	5 cycles	
11	unused[1]	unused[1]	

[1] Cbox response in this mode is UNDEFINED.

The fastest DATA_SPEED may not be selected with the slowest TAG_SPEED, for in this configuration the result of the cache hit calculation is not known in time for the Cbox state machines to operate correctly.

13.5.1.4 SIZE

Four backup cache sizes are selectable by using the SIZE bits, as shown in Table 13–36. These bits are cleared on reset so that when the chip is reset, the 128-kilobyte cache is selected by default.

Table 13–36: SIZE

SIZE<1:0>	Backup cache size
00	128 kilobytes
01	256 kilobytes
10	512 kilobytes
11	2 megabytes

13.5.1.5 FORCE_HIT

When FORCE_HIT is set, all memory references, both Dstream and Istream reads and writes, are forced to hit in the backup cache. The tag store state is not changed but data is always read or written. Reset clears this bit.

The backup cache must be enabled when the cache is used in FORCE_HIT mode.

This mode is expected to be used for testing purposes only.

13.5.1.6 DISABLE_ERRORS

When DISABLE_ERRORS is set, all ECC errors from the backup cache are ignored. Neither C%CBOX_H_ERR_H nor C%CBOX_S_ERR_H is asserted. C%CBOX_HARD_ERR_H is not asserted for data returning to the Mbox. The backup cache data syndrome is loaded into BCEDECC on every cache access; the behavior of BCETSTS, BCETIDX, BCETAG, BCEDSTS, and BCEDIDX is unpredictable. This feature allows operation of the backup cache even if the error detection and correction logic is faulty. It also allows access to the backup cache syndrome for the purposes of testing the ECC logic. Reset clears this bit.

13.5.1.7 SW_ECC

When SW_ECC is clear, the Cbox generates correct ECC check bits for all writes to the tag store and data RAMs. When SW_ECC is set, the Cbox does not generate the check bits when the backup cache is written with data, but uses the check bit values as specified by software and written in the BCDECC register. Note that if a read or write reference misses in the Bcache when SW_ECC is set, all four fills will be written with the ECC given in BCDECC when they return.

When SW_ECC is set and the tag store is written using an IPR write to BCTAG, the Cbox uses the check bits for the tag store as given through the IPR write. The value of SW_ECC does not affect tag store transactions other than IPR writes.

Reset clears this bit.

13.5.1.8 TIMEOUT_TEST

When TIMEOUT_TEST is set, the Cbox uses the internal clock to clock its read timeout counter. When TIMEOUT_TEST is clear, the Cbox uses **E%TIMEOUT_BASE_H** to clock its timeout counters. Reset clears this bit.

13.5.1.9 DISABLE_PACK

When DISABLE_PACK is set, the Cbox does not pack quadword writes together. Instead, the write packer passes every write it receives directly into the write queue. When the bit is clear, the Cbox write packer operates normally. DISABLE_PACK is intended for testing purposes only. Reset clears this bit.

13.5.1.10 PM_ACCESS_TYPE

PM_ACCESS_TYPE selects the type of Bcache access for the performance monitoring hardware.. The function of these three bits is fully described in Section 13.11. Reset clears these bits.

13.5.1.11 PM_HIT_TYPE

PM_HIT_TYPE selects the type of Bcache hit for the performance monitoring hardware. The function of these two bits is fully described in Section 13.11. Reset clears these bits.

13.5.1.12 FORCE_NDAL_PERR

When a 1 is written to FORCE_NDAL_PERR, a parity error is caused in the command field of the next outgoing NDAL transaction. The parity error is caused by inverting the value of **P%PARITY_H<2>**.

Setting this bit causes only one parity error. The parity error does not occur until NVAX is granted the NDAL for its next outgoing transaction. If software sets FORCE_NDAL_PERR and clears it before NVAX is granted the bus, NVAX will still force a parity error on the next transaction. In order to produce a second parity error on the bus, FORCE_NDAL_PERR must be cleared and set again by software.

Reset clears this bit.

13.5.1.13 SW_ETM

This is a software-writeable bit to put the backup cache into Error Transition Mode. When the cache is on and software ascertains that the cache is producing errors, it can set this bit in order to turn off the cache while ensuring cache coherency. Software can then flush owned data through use of the Bcache Deallocate IPR, BCFLUSH. In this manner, the unique data can be extracted from the cache before it is turned off completely. Reset clears this bit.

13.5.1.14 HW_ETM

Hardware sets this bit when an uncorrectable error is detected in the backup cache tag store or data rams, unless DISABLE_ERRORS is set. Hardware sets the bit to put the backup cache into Error Transition Mode.

Software clears HW_ETM by writing a one to it.

13.5.2 IPR A2 (hex), BCDECC

Figure 13–26: Format of the BCDECC

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| x| x| x| x| x| x|   ECCHI   | x| x| x| x| x| x| x| x| x| x| x| x|   ECCLO   | x| x| x| x| x| x| :BCDECC
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

The ECCHI field corresponds to data check bits <7:4>. The ECCLO field corresponds to data check bits <3:0>.

This register is written by software. It is a write only register.

Software writes BCDECC using an IPR_WRITE. The value in the register is then used to explicitly write ECC into the data RAMs during any write of the data RAMs, but only if SW_ECC is set in the control register. If SW_ECC is not set, hardware ignores the value in BCDECC and generates the check bits to be written using the ECC syndrome generator.

BCDECC is expected to be used during testing only. It allows software to explicitly write bad ECC into the data RAMs in order to test Cbox error detection logic. Note that BCDECC will be used as the source of the ECC check bits during any write of the backup cache data RAMs, including those done for fills. Cache transactions must be carefully controlled while this register is being used in order to obtain the expected results. BCDECC will probably be most useful when used in FORCE_HIT mode, so that no fills are generated.

Reset does not affect this register.

13.5.3 Backup Cache Tag Store Error Registers (BCETSTS, BCETIDX, BCETAG)

On some tag store errors, hardware overwrites the corrupted values so that they cannot be diagnosed by reading the tag store directly. For this reason there are tag store error registers which hold the relevant data, so that software can understand the problem.

The tag store error registers are loaded when any tag store error occurs. Their contents are not changed during reset. The status bits in BCETSTS indicate what sort of error happened. Correctable errors are indicated by the CORR bit; the UNCORR and BAD_ADDR errors are both uncorrectable-type errors.

If no error is yet logged in the registers, the registers are loaded when either a correctable or an uncorrectable error occurs. Once the registers are loaded with information from a correctable error, they are locked against further correctable errors, and are only loaded again if an uncorrectable error happens. At this time either UNCORR or BAD_ADDR is set. The LOCK bit in BCETSTS is set as well. In this way, information from the first correctable error is held in the registers, and is only overwritten if an uncorrectable error happens later.

The error registers are cleared and unlocked by software. If the error registers hold data from a non-correctable error and yet another non-correctable error happens before the error registers are unlocked, the LOST_ERR bit is set. This indicates to software that it does not have sufficient information in the error registers to recover from all uncorrectable errors which have occurred.

13.5.3.1 Bcache Error Tag Status (BCETSTS)

The BCETSTS register gives the general status of an error in the tag store, indicating the transaction which was taking place at the time and the type of error. The register is written by hardware and read by software. Hardware does not clear the error bits in this register; this must be done by software using write-one-to-clear to the bottom 5 bits of the register. The contents of the register are not changed during reset.

Figure 13–27: IPR A3 (hex), BCETSTS

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x|     TS_CMD     |  |  |  |  |  | :BCETSTS
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                                                  |  |  |  |  |
                                                                                  |  |  |  |  `-LOCK
                                                                                  |  |  |  `-CORR
                                                                                  |  |  `-UNCORR
                                                                                  |  `-BAD_ADDR
                                                                                  `-LOST_ERR
```

Table 13–37: BCETSTS Field Descriptions

Name	Extent	Type	Description
LOCK	0	WC	Lock bit. Indicates that BCETSTS (except LOST_ERR), BCETIDX, and BCETAG are locked.
CORR	1	WC	Indicates that a correctable ECC error was encountered.

Table 13–37 (Cont.): BCETSTS Field Descriptions

Name	Extent	Type	Description
UNCORR	2	WC	Indicates that an uncorrectable ECC error was encountered.
BAD_ADDR	3	WC	Indicates that an addressing error was detected. This is an uncorrectable error.
LOST_ERR	4	WC	Indicates that more than one uncorrectable error occurred which was not recorded in the error registers.
TS_CMD	9:5	R	Indicates what tag store command was being processed at the time the error occurred.

13.5.3.1.1 LOCK

Whenever the tag store error registers are locked due to an uncorrectable error, the LOCK bit is set. At this time either UNCORR or BAD_ADDR is also set to indicate the type of uncorrectable error. When the LOCK bit is set, the BCETSTS, BCETIDX, and BCETAG registers are all locked. Clearing the lock bit unlocks all three registers. The LOCK bit is set by hardware and it is cleared by software. It is a write-one-to-clear bit.

13.5.3.1.2 CORR

CORR is set when the tag store ECC decoder detects a correctable error. When this occurs, the Bcache Tag Store Error registers are loaded and are locked against further correctable errors. They are not locked against an uncorrectable error which follows. BCETSTS<LOCK> is not set.

If a correctable error is followed by an uncorrectable error, the CORR bit remains set.

The CORR bit is set by hardware and it is cleared by software. It is a write-one-to-clear bit.

13.5.3.1.3 UNCORR

UNCORR is set when the tag store ECC decoder detects an uncorrectable error. When this occurs, the Bcache Tag Store Error registers are loaded and locked.

The UNCORR bit and the BAD_ADDR bit are exclusive: only one of them is set for a given error which sets the LOCK bit. If the other type of error occurs later, the related bit is not set since the register is already locked. In this case, LOST_ERR is set instead.

The UNCORR bit is set by hardware and it is cleared by software. It is a write-one-to-clear bit.

13.5.3.1.4 BAD_ADDR

BAD_ADDR is set when the tag store ECC decoder detects an error in the address bit, indicating some problem with the address lines going to the tag rams. This is an uncorrectable error, thus, when it occurs, the Bcache Tag Store Error registers are loaded and locked.

The UNCORR bit and the BAD_ADDR bit are exclusive: only one of them is set for a given error which sets the LOCK bit. If the other type of error occurs later, the related bit is not set since the register is already locked. In this case, LOST_ERR is set instead.

The BAD_ADDR bit is set by hardware and it is cleared by software. It is a write-one-to-clear bit.

13.5.3.1.5 LOST_ERR

LOST_ERR indicates that after the first uncorrectable error was recorded in the tag store error registers, an additional uncorrectable error occurred for which state was not saved. LOST_ERR is set by hardware and is cleared by software. It is a write-one-to-clear bit.

13.5.3.1.6 TS_CMD

The TS_CMD field indicates what the tag store was doing when the error was detected. Its values are listed in Table 13–38.

Table 13–38: Interpretation of TS_CMD

TS_CMD	NAME	Tag Store Operation
00111	DREAD	Data-stream (DREAD) or DREAD_MODIFY tag lookup
00011	IREAD	Instruction-stream tag lookup
00010	OREAD	Ownership-read tag lookup for a write or a READ_LOCK
01000	WUNLOCK	Ownership-read tag lookup for a WRITE_UNLOCK (lookup done only in ETM)
01101	R_INVAL	Cache coherency tag lookup as the result of NDAL DREAD or IREAD
01001	O_INVAL	Cache coherency tag lookup as the result of NDAL OREAD or WRITE
01010	IPR_DEALLOC	Tag lookup for an explicit IPR deallocate operation

There are three tag store operations which do not cause any sort of errors: tag store update after a fill, ipr write of the tag store, ipr read of the tag store. Thus, these commands will not appear in BCETSTS.

13.5.3.2 Bcache Error Tag Index (BCETIDX)

This register is loaded and locked when a tag store error occurs. If a correctable error is followed by a second error which is not correctable, the register is loaded with information from the second, more serious error. Except for this case, once it is locked, it is not changed until software explicitly unlocks the register. This register is written by hardware and read by software. Its contents are not changed during reset.

Figure 13–28: IPR A4 (hex), BCETIDX

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                             Backup Cache Tag Store Address                     | 0| 0| 0| 0| 0| :BCETIDX
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

BCETIDX contains the complete hexaword address corresponding to a tag store request which resulted in an error. Since the full address is saved, both the cache index and the cache tag of the request are known. Thus, this register shows what index was being accessed when the error occurred as well as showing what the tag of the request was. Software can compare this tag with the actual tag read from the RAMs, which is saved in BCETAG.

On a BCFLUSH which incurs an error, the address used to flush the cache is captured in BCETIDX, not the memory address of the block.

13.5.3.3 Bcache Error Tag (BCETAG)

This register is loaded when a tag store error occurs. It is locked when an uncorrectable error occurs on a tag store access. Once the register is locked, it is not overwritten until it is unlocked by software. BCETAG is written by hardware and read by software. It is a read-only register from the software point of view. The contents of BCETAG are not changed during reset.

The register holds the data which was read from the tag store and produced the error, as shown in Figure 13–29.

Figure 13–29: IPR A5 (hex), BCETAG

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                   TAG          |           |    ECC          |  |  | 0| 0| 0| 0| 0| 0| 0| 0| 0| :BCETAG
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                       |                        |  |
                                       `-TAG or 0, based        |  `-VALID
                                         on cache size          `-OWNED
```

Table 13–39: BCETAG Field Descriptions

Name	Extent	Type	Description
VALID	9	RO	Valid bit
OWNED	10	RO	Ownership bit
ECC	16:11	RO	ECC check bits
TAG	31:17	RO	Backup cache tag

13.5.3.3.1 VALID

VALID is the bit read from the tag RAMs which indicates whether the block is valid in the Bcache.

13.5.3.3.2 OWNED

OWNED is the bit read from the tag RAMs which indicates whether the Bcache owns the block in question.

13.5.3.3.3 ECC

The ECC field contains the check bits as read from the tag RAMs during the tag access which produced the error.

13.5.3.3.4 TAG

The TAG field of BCETAG is the cache tag as read from the tag RAMs. It must be interpreted based on the cache size being used, as shown in Table 13–40. When certain address bits are not used as tag bits for the cache size given, their value in BCETAG is 0.

Table 13–40: TAG Interpretation

Cache size	Tag bits used	Unused tag bits
128 kilobytes	TAG<31:17>	None
256 kilobytes	TAG<31:18>	TAG<17>
512 kilobytes	TAG<31:19>	TAG<18:17>
2 megabytes	TAG<31:21>	TAG<20:17>

13.5.4 Backup Cache Data RAM Error Registers (BCEDSTS, BCEDIDX, BCEDECC)

The data RAM error registers hold data relevant to errors in the backup cache data RAMs, so that software can understand the problem.

BCEDSTS holds the general status of the problem. BCEDIDX holds the data RAM index being used when the problem occurred. BCEDECC holds the syndrome bits as calculated on the data which was read from the RAMs when the problem occurred.

If no error is yet logged in the data RAM error registers, the registers are loaded when either a correctable or an uncorrectable error occurs. Once the registers are loaded with information from a correctable error, they are locked against further correctable errors, and are only loaded again if an uncorrectable error happens. If an uncorrectable error happens, the LOCK bit in BCEDSTS is set and the registers are not overwritten until software clears the error bits. In this way, information from the first correctable error is held in the registers, and is only overwritten if an uncorrectable error happens later.

If the registers are locked, any subsequent non-correctable error causes the LOST_ERR bit to be set, but does not modify any other information in the registers. LOST_ERR indicates to software that it does not have sufficient information in the error registers to recover from all uncorrectable errors which have occurred.

Of the backup cache data RAM error registers, only BCEDSTS is writable by software. Software clears the error and lock bits which reenables all the Data RAM error registers to record the next error which occurs.

The contents of BCEDSTS, BCEDIDX, and BCEDECC are not affected by reset.

13.5.4.1 Bcache Error Data Status (BCEDSTS)

Figure 13–30: IPR A6 (hex), BCEDSTS

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x|   DR_CMD  | 0| 0| 0|  |  |  |  |  | :BCEDSTS
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                                                   |  |  |  |  |
                                                                                   |  |  |  |  `-LOCK
                                                                                   |  |  |  `-CORR
                                                                                   |  |  `-UNCORR
                                                                                   |  `-BAD_ADDR
                                                                                   `-LOST_ERR
```

Table 13–41: BCEDSTS Field Descriptions

Name	Extent	Type	Description
LOCK	0	WC	Lock bit. Indicates that the BCEDSTS, BCEDIDX, and BCEDECC registers are locked.
CORR	1	WC	Indicates that a correctable ECC error was encountered.

Table 13–41 (Cont.): BCEDSTS Field Descriptions

Name	Extent	Type	Description
UNCORR	2	WC	Indicates that an uncorrectable ECC error was encountered.
BAD_ADDR	3	WC	Indicates that an addressing error was detected.
LOST_ERR	4	WC	Indicates that a second uncorrectable error occurred; it was not recorded in the error registers.
DR_CMD	11:8	R	Indicates what command was being processed by the data RAMs at the time the error occurred.

The LOCK bit is set when an error which was not correctable has occurred. If the CORR bit is set, the data ram error registers are locked unless an uncorrectable error occurs. On an uncorrectable error, the LOCK bit is set and the registers are permanently locked until unlocked by software.

The contents of BCEDSTS are not affected by reset.

13.5.4.1.1 LOCK

Whenever the data RAM error registers are loaded with an uncorrectable error, the LOCK bit is set. At this time either UNCORR or BAD_ADDR is also set to indicate the type of uncorrectable error. (A correctable error does not set BCEDSTS<LOCK>.) When the LOCK bit is set, the BCEDSTS, BCEDIDX, and BCEDECC registers are all locked. Clearing the lock bit unlocks all three registers. The LOCK bit is set by hardware and it is cleared by software. It is a write-one-to-clear bit.

13.5.4.1.2 CORR

CORR is set when the data ECC decoder detects a correctable error. When this occurs, the Bcache Data Error registers are loaded and locked against further correctable errors; BCEDSTS<LOCK> is not set. The CORR bit is set by hardware and it is cleared by software. It is a write-one-to-clear bit.

13.5.4.1.3 UNCORR

UNCORR is set when the data ECC decoder detects an uncorrectable error. When this occurs, the Bcache Data Error registers are loaded and locked. The UNCORR bit is set by hardware and it is cleared by software. It is a write-one-to-clear bit.

13.5.4.1.4 BAD_ADDR

BAD_ADDR is set when the data ECC decoder detects an error in the address bit, indicating some problem with the address lines going to the data rams. This is an uncorrectable error, thus, when it occurs, the Bcache Data Error registers are loaded and locked. The BAD_ADDR bit is set by hardware and it is cleared by software. It is a write-one-to-clear bit.

13.5.4.1.5 LOST_ERR

LOST_ERR indicates that after the first uncorrectable error was recorded in the data error registers, an additional uncorrectable error occurred for which state was not saved. LOST_ERR is set by hardware and is cleared by software. It is a write-one-to-clear bit.

13.5.4.1.6 DR_CMD

The DR_CMD field indicates what the data RAMs were doing when the error was detected. Its values are listed in Table 13–42.

Table 13–42: Interpretation of DR_CMD

DR_CMD<11:8>	Name	Data RAM operation
0111	DREAD	Data lookup for a Dstream read
0011	IREAD	Data lookup for an Istream read
0100	WBACK	Data lookup for a writeback
0010	RMW	Data lookup for a read-modify-write (done for normal writes and WRITE_UNLOCKs.)

There are two data RAM operations which do not cause any sort of errors: full quadword writes and fills. Thus, these commands will not appear in BCEDSTS.

DR_CMD is only written by hardware. It is read-only for software.

13.5.4.2 Bcache Error Data Index (BCEDIDX)

This register holds the index of a data RAM transaction; it is loaded when an error is detected on a data RAM access. The index loaded due to a correctable error is not overwritten unless an uncorrectable error occurs afterwards. If an uncorrectable error occurs, BCEDIDX is loaded and locked. BCEDIDS is unlocked by software; the lock bit is in the BCEDSTS register.

BCEDIDX is read-only from software's point of view. Its contents are not affected by reset.

Figure 13–31: IPR A7 (hex), BCEDIDX

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0| 0|           |     Backup cache data RAM index         | 0| 0| 0| :BCEDIDX
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                       |
                                       `-index or undefined, based on cache size
```

BCEDIDX must be interpreted based on the cache size being used, as shown in Table 13–43. When certain address bits are not used as index bits for the cache size given, their value in BCEDIDX is undefined.

Table 13–43: BCEDIDX Interpretation

Cache size	Index bits used	undefined index bits
128 kilobytes	BCEDIDX<16:3>	BCEDIDX<20:17>
256 kilobytes	BCEDIDX<17:3>	BCEDIDX<20:18>
512 kilobytes	BCEDIDX<18:3>	BCEDIDX<20:19>
2 megabytes	BCEDIDX<20:3>	None

13.5.4.3 Bcache Error Data ECC (BCEDECC)

This register holds the syndrome as calculated on the backup cache data and check bits. It is loaded when an error occurs on a data RAM access. Then it follows the same lock rules that the other Bcache Data Error registers follow. It is unlocked by software. The lock bit is in the BCEDSTS register. The contents of BCEDECC are not affected by reset.

When DISABLE_ERRORS is set, BCEDECC is loaded on every quadword read from the cache. This provides a way of testing the ECC logic by reading the results of the syndrome calculation. Note that because 4 quadwords are read from the Bcache at a time, BCEDECC will contain the syndrome from the LAST quadword read after the 4-qw transaction is complete. Software can control which quadword is read last by varying the requested quadword of a transaction; the Bcache controller always returns the requested quadword first, then returns the remaining 3 quadwords in wraparound order. For example, if the programmer wants to see the contents of BCEDECC after quadword 2, she would do a read to quadword 3 of the block, and the quadwords would be read out in the order 3-0-1-2.

Software can use BCDECC to write known check bits to the data RAMs; when the RAMs are read, the syndrome is captured by BCEDECC. Once the syndrome is known, the check bits which were calculated by the ECC hardware can be deduced, because the check bits read from the RAMs were known. The syndrome is simply the XOR of the calculated check bits and the check bits which were read from the RAMs.

If the programmer wants to learn what the correct checkbits for a particular data pattern should be, she can write data to the cache while BCDECC contains all zero's and CCTL<SW_ECC> is set. This forces checkbits of zero to be written to the cache with the data. When the data is read back, BCEDECC will contain the correct checkbits for the data (the XOR of the checkbits read and the checkbits calculated by hardware).

BCEDECC is read-only from software's point of view.

Figure 13–32: IPR A8 (hex), BCEDECC

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| x| x| x| x| x| x|     ECCHI     | x| x| x| x| x| x| x| x| x| x| x| x|     ECCLO     | x| x| x| x| x| x| :BCEDECC
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

The ECCHI field corresponds to syndrome bits <7:4>. The ECCLO field corresponds to syndrome bits <3:0>.

13.5.5 Fill Error Registers (CEFADR, CEFSTS)

Some errors are related to outstanding reads to memory. These errors may be diagnosed using the CEFSTS and CEFADR registers. CEFSTS holds general information about the type of read outstanding; CEFADR holds the address of the outstanding read. The contents of these these registers are not changed during reset.

13.5.5.1 Cbox Error Fill Status (CEFSTS)

The CEFSTS register holds information related to a problem on a read which was sent to memory. If a read request to memory times out or is terminated with RDE, the CEFSTS register and the CEFADR register are loaded and locked.

The register is read-write. Only the lowest five bits and the UNEXPECTED_FILL bit may be written, and then only to clear them after an error. CEFSTS is not affected by reset.

Figure 13–33: IPR AC (hex), CEFSTS

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| x| x| x| x| x| x| x| x| x| x|  | x| x| x| x|COUNT|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | :CEFSTS
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                |                    |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
                                |                    |  |  |  |  |  |  |  |  |  |  |  |  |  |  `-RDLK
                                |                    |  |  |  |  |  |  |  |  |  |  |  |  |  `-LOCK
                                |                    |  |  |  |  |  |  |  |  |  |  |  |  `-TIMEOUT
                                |                    |  |  |  |  |  |  |  |  |  |  |  `-RDE
                                |                    |  |  |  |  |  |  |  |  |  |  `-LOST_ERR
                                |                    |  |  |  |  |  |  |  |  |  `-ID0
                                |                    |  |  |  |  |  |  |  |  `-IREAD
                                |                    |  |  |  |  |  |  |  `-OREAD
                                |                    |  |  |  |  |  |  `-WRITE
                                |                    |  |  |  |  |  `-TO_MBOX
                                |                    |  |  |  |  `-RIP
                                |                    |  |  |  `-OIP
                                |                    |  |  `-DNF
                                |                    |  `-RDLK_FL_DONE
                                `-UNEXPECTED_FILL    `-REQ_FILL_DONE
```

Table 13–44: CEFSTS Field Descriptions

Name	Extent	Type	Description
RDLK	0	WC	Indicates that a READ_LOCK was in progress.
LOCK	1	WC	Indicates that an error occurred and the register is locked.
TIMEOUT	2	WC	FILL failed due to transaction timeout.
RDE	3	WC	FILL failed due to Read Data Error.
LOST_ERR	4	WC	Indicates that more than one error related to fills occurred.
ID0	5	RO	NDAL identification bit for the read request.
IREAD	6	RO	This is an Istream read from the Mbox which may be aborted.
OREAD	7	RO	This is an outstanding OREAD.
WRITE	8	RO	This read was done for a write.

Table 13–44 (Cont.): CEFSTS Field Descriptions

Name	Extent	Type	Description
TO_MBOX	9	RO	Data is to be returned to the Mbox.
RIP	10	RO	READ invalidate pending.
OIP	11	RO	OREAD invalidate pending.
DNF	12	RO	Do not fill - data not to be written into the cache or validated when the fill returns.
RDLK_FL_DONE	13	RO	Indicates that the last fill for a READ_LOCK arrived.
REQ_FILL_DONE	14	RO	Indicates that the requested quadword was successfully returned from the NDAL.
COUNT	16:15	RO	For a memory space transaction, indicates how many of the fill quadwords have been successfully returned. For I/O space, is set to 11(BIN) when the transaction starts as only one quadword will be returned.
UNEXPECTED_FILL	21	WC	Set to indicate that an unexpected fill was received from the NDAL.

13.5.5.1.1 RDLK

RDLK is set to show that a READ_LOCK is in progress. This bit is write-one-to-clear. The side effect of performing a write-one-to-clear to this bit is to clear the VALID bit for an entry which had its RDLK bit set; this has the effect of clearing out the FILL_CAM entry. This is the same action which is taken when a WRITE_UNLOCK is received. Microcode uses this functionality during certain error sequences; the bit is implemented in the zero position to make the microcoding as efficient as possible.

This bit is normally not read as a one by software, because the microcode ensures that the READ_LOCK-WRITE_UNLOCK sequence is an indivisible operation. If, however, the first quadword of a READ_LOCK is returned successfully and then the transaction either times out or is terminated in RDE, CEFSTS is loaded with the RDLK bit set.

13.5.5.1.2 LOCK

The LOCK bit is set when a read transaction which has been sent to memory terminates in Read Data Error or in Timeout. At the same time, all information corresponding to the read is loaded from the FILL_CAM into the CEFSTS register. When the LOCK bit is set, one of TIMEOUT, RDE, or UNEXPECTED_FILL is also set to indicate the type of error. Once the LOCK bit is set, none of the information in CEFSTS or CEFADR changes, with the possible exception of LOST_ERR, until the LOCK bit is cleared.

Hardware sets the LOCK bit and software clears it by writing a one to that location.

13.5.5.1.3 TIMEOUT

TIMEOUT is set when a read transaction which was sent to the NDAL times out for some reason. When TIMEOUT is set, the LOCK bit is also set.

Hardware sets the TIMEOUT bit and software clears it by writing a one to that location.

13.5.5.1.4 RDE

RDE (Read Data Error) is set when a read transaction which was sent to the NDAL terminates in RDE. When the RDE bit is set, the LOCK bit is also set. The UNEXPECTED_FILL bit will be set as well, if the RDE was actually unexpected (no read corresponding to the RDE was outstanding when that RDE was received).

Hardware sets the RDE bit and software clears it by writing a one to that location.

13.5.5.1.5 LOST_ERR

The LOST_ERR bit is set when CEFSTS is already locked and another RDE, timeout, or unexpected fill error occurs. This indicates to software that multiple errors have happened and state has not been saved for every error.

Hardware sets the LOST_ERR bit and software clears it by writing a one to that location.

13.5.5.1.6 ID0

ID0 corresponds to the NDAL signal, **P%ID_H<0>**, which was issued with the read that failed. It also indicates which one of the two FILL_CAM entries was used to save information about the transaction while it was outstanding.

13.5.5.1.7 IREAD

IREAD indicates that the transaction in error was an IREAD.

13.5.5.1.8 OREAD

OREAD indicates that the transaction in error was an OREAD; the OREAD may have been done for a write, a READ_LOCK, or a read modify.

13.5.5.1.9 WRITE

WRITE indicates that the transaction in error was an OREAD done because of a write request.

13.5.5.1.10 TO_MBOX

TO_MBOX indicates that data returning for the read was to be sent to the MBOX.

13.5.5.1.11 RIP

RIP (Read Invalidate Pending) is set when a cache coherency transaction due to a read on the NDAL is requested for a block which has Oread fills outstanding at the time. This triggers a writeback of the block when the fill data arrives; a valid copy of the data is kept in the cache.

13.5.5.1.12 OIP

OIP (Oread Invalidate Pending) is set when a cache coherency transaction due to an OREAD or a WRITE on the NDAL is requested for a block which has OREAD fills outstanding at the time. This triggers a writeback and invalidate of the block when the fill data arrives.

13.5.5.1.13 DNF

DNF (Do Not Fill) is set when data for a read is not to be written into the Bcache. This is the case when the cache is off, in ETM, or when the read is to I/O space. The assertion of this bit prevents the block from being validated in the cache.

13.5.5.1.14 RDLK_FL_DONE

This bit is set in the fill cam when a READ_LOCK hits in the Bcache or the last fill arrives from the BIU for a READ_LOCK. Once this is set, the corresponding WRITE_UNLOCK is allowed to proceed. This overrides the FILL_CAM block conflict on the WRITE_UNLOCK which is inevitable since the READ_LOCK is held in the FILL_CAM until the WRITE_UNLOCK is done.

13.5.5.1.15 REQ_FILL_DONE

REQ_FILL_DONE is set when the requested quadword of data was successfully received from the NDAL. This is used to allow error handling software to differentiate between an error which occurred before the requested data was received, and an error which occurred after the requested data was received.

If the error occurs while the requested data is being returned, such as the requested data being returned with RDE, it is as if the requested data was not received. REQ_FILL_DONE will not be set because the requested data was not successfully received.

13.5.5.1.16 COUNT

These two bits indicate how many of the expected four quadwords have been returned successfully from memory for this read. If they are 00(BIN), no quadwords have returned, if they are 01(BIN), one quadword has returned, etc. If the entry was for a quadword read, the count bits are set to 11(BIN) when the reference is sent out.

As an example, if RDE is returned before any other RDR returns for a hexaword request, COUNT will be 00(BIN), to indicate that no quadwords of data were successfully returned.

13.5.5.1.17 UNEXPECTED_FILL

UNEXPECTED_FILL is set to indicate that an RDE or an RDR cycle was received from the NDAL with an ID for which the FILL_CAM entry was not valid. When UNEXPECTED_FILL is set, CEFSTS and CEFADR are loaded and locked. RDE will also be set if the unexpected fill was an RDE rather than an RDR.

UNEXPECTED_FILL is a write-one-to-clear bit which is set by hardware and cleared by software.

13.5.5.2 Fill Error Address (CEFADR)

The CEFADR register holds the original quadword read address of a fill which ended in an error condition. It is loaded when an error is detected on a fill. It is a read-only register.

CEFADR is locked when CEFSTS is locked. Its contents are not changed during reset.

Figure 13–34: IPR AB (hex), CEFADR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                 Fill error address                                   | 0| 0| 0| :CEFADR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

13.5.6 NDAL Error Registers (NESTS, NEOADR, NEOCMD, NEDATHI, NEDATLO, NEICMD)

The NDAL error registers hold information related to NDAL errors. NESTS, NDAL Error Status, holds error bits relating to any problems encountered.

NEOADR, NDAL Error Output Address, holds the address corresponding to the cycle which was in error. NEOCMD, NDAL Error Output Command, holds the command bits corresponding to the cycle in error.

NEDATHI, NDAL Error Data High Longword, and NEDATLO, NDAL Error Data Low Longword, hold the data from an NDAL cycle where NVAX detected a parity error on the bus. NEICMD, NDAL Error Input Command, holds the command bits corresponding to a cycle with a parity error.

The NDAL error registers are not affected by reset: their contents are not changed during reset.

13.5.6.1 NDAL Error Status IPR (NESTS)

The NESTS register holds information about any errors which happened on the NDAL. All six bits in this register are write-one-to-clear. Reset does not affect this register. Power-up does not initialize the register.

Figure 13–35: IPR AE (hex), NESTS

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x|  |  |  |  |  |  | :NESTS
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                                                |  |  |  |  |  |
                                                                                |  |  |  |  |  |
                                                                                |  |  |  |  |  `-NOACK
                                                                                |  |  |  |  `-BADWDATA
                                                                                |  |  |  `-LOST_OERR
                                                                                |  |  `-PERR
                                                                                |  `-INCON_PERR
                                                                                `-LOST_PERR
```

Table 13–45: NESTS Field Descriptions

Name	Extent	Type	Description
NOACK	0	WC	Indicates that **P%ACK_L** was not asserted for an outgoing NVAX cycle. This bit locks NEOADR and NEOCMD.
BADWDATA	1	WC	Indicates that an outgoing data cycle was accompanied by the BADWDATA command. This bit locks NEOADR and NEOCMD.
LOST_OERR	2	WC	Indicates that multiple outgoing errors, either NOACK or BADWDATA, were detected.
PERR	3	WC	Indicates that a parity error was detected on the NDAL. This bit locks NEDATHI, NEDATLO, AND NEICMD.

Table 13–45 (Cont.): NESTS Field Descriptions

Name	Extent	Type	Description
INCON_PERR	4	WC	Inconsistent parity error.
LOST_PERR	5	WC	Indicates that multiple NDAL parity errors were detected.

13.5.6.1.1 NOACK

NOACK is set when NVAX detects that **P%ACK_L** was not asserted on the NDAL for an outgoing NVAX cycle. When NOACK is set, NEOADR and NEOCMD are locked so that software can read them to see what transaction was being attempted when the error occurred.

NOACK is set on any outgoing NVAX cycle which is not acknowledged, whether it was an address cycle or a data cycle. The information which is locked in NEOADR and NEOCMD corresponds to the address cycle of the transaction. For example, if an outgoing write data cycle is not acknowledged, the address cycle for that write operation is saved in NEOADR and NEOCMD.

NOACK is not set if there was a previous BADWDATA. If a BADWDATA cycle is NOACK'd, both BADWDATA and NOACK are set.

NOACK is cleared by a write-one-to-clear.

13.5.6.1.2 BADWDATA

BADWDATA is set when the BIU receives data for a writeback from the cache which had an uncorrectable ECC error, and thus is being issued on the NDAL with the BADWDATA command. When BADWDATA is set, NEOADR and NEOCMD are locked so that software can read them to retrieve the information about the failure.

The address for the write operation is captured in NEOADR, and the command information for the cycle is captured in NEOCMD.

BADWDATA is not set if there was a previous NOACK. If a BADWDATA cycle is NOACK'd, both BADWDATA and NOACK are set.

13.5.6.1.3 LOST_OERR

LOST_OERR is set when NOACK or BADWDATA is already set and another one of those errors occurs. It notifies software that state was saved only for the first outgoing error.

LOST_OERR is cleared by a write-one-to-clear.

13.5.6.1.4 PERR

PERR is set when NVAX detects a parity error on the NDAL. When PERR is set, NEDATHI, NEDATLO, and NEICMD are locked so that software can read them to see what was on the NDAL when the error occurred.

Since NVAX calculates parity on every cycle, PERR will be set on both its own transfers and the transfers of other devices which fail the parity check.

PERR is cleared by a write-one-to-clear.

13.5.6.1.5 INCON_PERR

INCON_PERR (Inconsistent parity error) is set when an NDAL parity error is detected on a cycle which is also acknowledged with **P%ACK_L**. This means that NVAX detected a parity error but some other device acknowledged the transfer.

INCON_PERR is only set in conjunction with PERR. It is not set unless PERR is set. If one NDAL parity error has already occurred, setting PERR, but INCON_PERR was not set for that cycle, a subsequent cycle with an inconsistent parity error will not cause INCON_PERR to be set.

INCON_PERR is cleared by a write-one-to-clear.

13.5.6.1.6 LOST_PERR

LOST_PERR is set when PERR is already set and another NVAX transfer fails the parity check. LOST_PERR notifies software that multiple NVAX transfers have failed the parity check; state was saved only for the first.

LOST_PERR is cleared by a write-one-to-clear.

13.5.6.2 NDAL Error Output Address IPR (NEOADR)

The NEOADR register is loaded for every address cycle which the Cbox drives onto the NDAL, unless it is locked. It is loaded during the cycle when the corresponding **P%ACK_L** should be asserted on the NDAL. It is locked when the NOACK bit in the NESTS register is set.

When NEOADR is locked, it contains the address information for the first transaction which failed. If it is read when it is not locked, it contains information from the last address cycle which was acknowledged on the NDAL.

The format of NEOADR matches the low longword of the NDAL during an address cycle.

NEOADR is read-only to software. Its contents are not changed during reset.

Figure 13–36: IPR B0 (hex), NEOADR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                         NDAL address                                          | :NEOADR
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

13.5.6.3 NDAL Error Output Command (NEOCMD)

The NEOCMD register is loaded and locked exactly as NEOADR is loaded and locked. The format of NEOCMD is similar to that of the high longword of the NDAL during an address cycle. The high quadword byte enable positions are NOT included, since NVAX only uses quadword byte-enabled transactions; and the NDAL ID and command are added in the lower four bits of the longword.

The contents of NEOCMD are not affected by reset.

Figure 13–37: IPR B2 (hex), NEOCMD

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| LEN   |  x|  x|  x|  x|  x|  x|  x|  x|  x|  x|  x|  x|  x|  x|           BYTE_EN             |  0|    ID     |      CMD      | :NEOCMD
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Table 13–46: NEOCMD Field Descriptions

Name	Extent	Type	Description
CMD	3:0	RO	NDAL command as driven by NVAX during the transaction. For specific values, see Section 3.3.4.2.
ID	6:4	RO	Commander ID as driven by NVAX during the transaction. For specific values, see Section 3.3.4.3.
BYTE_EN	15:8	RO	Byte enable as driven by NVAX during the transaction. For specific values, see Section 3.3.4.1.
LEN	31:30	RO	Length of the NDAL transaction. For specific values, see Section 3.3.4.1.

The meanings of these fields are described in Chapter 3.

13.5.6.4 NDAL Error Input Command (NEICMD)

NEICMD, NEDATHI, and NEDATLO are loaded at the same time and they are locked at the same time. They are all loaded when a parity error occurs; at this time the PERR bit is set in NESTS, which locks the three registers. If a second NDAL parity error happens, the registers are not loaded again; they are not loaded again until after they are unlocked when software clears PERR.

NEICMD contains the **P%CMD_H<3:0>**, **P%ID_H<2:0>**, and **P%PARITY_H<2:0>** bits from the failed transfer.

NEICMD is a read-only register. Its contents are not changed during reset.

Figure 13–38: IPR B8 (hex), NEICMD

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| x| PARITY |   ID    |   CMD    | :NEICMD
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

13.5.6.4.1 PARITY

The PARITY field corresponds to the NDAL lines **P%PARITY_H<2:0>**.

13.5.6.4.2 ID

The ID field corresponds to the NDAL lines **P%ID_H<2:0>**.

13.5.6.4.3 CMD

The CMD field corresponds to the NDAL lines **P%CMD_H<2:0>**.

13.5.6.5 NDAL Error Data High and NDAL Error Data Low (NEDATHI and NEDATLO)

NEDATHI and NEDATLO behave analogously to NEICMD. They capture **P%NDAL_H<63:0>** during a cycle with a parity error. NEDATHI contains the high longword of data from the NDAL (**P%NDAL_H<63:32>**); NEDATLO contains the low longword of data from the NDAL (**P%NDAL_H<31:0>**).

The format of NEDATHI and NEDATLO must be interpreted based on the CMD found in NEICMD. If the CMD field shows that the cycle was a data cycle, the registers contain two longwords of data. If the CMD field shows that the cycle was an address cycle, the registers are in the format of an NDAL address cycle, as shown in Figure 13–39 and Figure 13–40.

The contents of NEDATHI and NEDATLO are not affected by reset.

Figure 13–39: IPR B4 (hex), NEDATHI

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| LEN |   UNDEFINED     |                    BYTE_EN                    |       UNDEFINED       | :NEDATHI
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 13–40: IPR B6 (hex), NEDATLO

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                            address                                            | :NEDATLO
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

13.5.7 Backup Cache Tag Store Access Through IPR Reads and Writes (BCTAG)

Direct access to the backup cache tag store is provided to aid in error recovery and diagnosis and to assist testing. These accesses work whether the cache is on or off, in ETM or in force hit mode.

If there is a valid FILL_CAM entry for the same cache block which is being accessed through an IPR read or write, the IPR read or write is stalled until the fills return and the FILL_CAM entry is no longer valid.

When the backup cache tag store is being accessed through IPR reads and writes, address bits <24:22> = 100 (BINARY). Address bits <20:5> are used as the index into the tag store RAMs; these indicate which backup cache location is to be written or read.

Figure 13–41: Backup Cache Tag Store IPR Addressing Format

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|        SBZ         | 1  0  0| x|           |       BCTAG Index                 |    SBZ       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                       |
                                       `-BCTAG Index or SBZ, based on cache size.
```

Some or all of bits <20:17> are not actually used as the index if the cache is smaller than 2 megabytes. This is set out explicitly in Table 13–48.

The format for reading and writing the backup cache tag store as an IPR is described in Figure 13–42 and Table 13–47.

Figure 13–42: IPRs 01000000 thru 011FFFE0 (hex), BCTAG

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|           TAG                  |           |     ECC         |  |  | x| x| x| x| x| x| x| x| x| :BCTAG
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                       |                         |  |
                                       `-TAG or 0, based         |  `-VALID
                                         on cache size           `-OWNED
```

Table 13–47: BCTAG Field Descriptions

Name	Extent	Type	Description
VALID	9	RW	Valid bit
OWNED	10	RW	Ownership bit
ECC	16:11	RW[1]	ECC check bits
TAG	31:17	RW	Tag data

[1] The ECC bits are written from the value given in the IPR_WRITE only if the SW_ECC bit of the CCTL IPR is set. Otherwise, the Cbox generates and writes correct ECC for the tag, owned and valid values being written.

Some or all of TAG<20:17> are not actually used as tag if the cache is larger than 128 kilobytes. This is set out in Table 13–48.

Table 13–48: Tag and Index Interpretation for BCTAG IPR

Cache size	Tag bits used	Index bits used
128 kilobytes	TAG<31:17>	Index<16:5>
256 kilobytes	TAG<31:18>	Index<17:5>
512 kilobytes	TAG<31:19>	Index<18:5>
2 megabytes	TAG<31:21>	Index<20:5>

The tag store must be initialized to a known state when the chip is powered up. This is done through IPR_WRITEs to BCTAG.

When the tag store is read, the ECC check bits are read out directly from the tag store in the format shown. ECC is not checked on IPR accesses to the tag store; no errors can occur during these accesses.

Some care must be taken if IPR reads of the tag store are done while other transactions are in progress. The tag information read out may not be what the programmer expects if cache misses or cache coherency transactions are in progress on the block which is being read. For example, if a cache miss is in progress, the new tag will be in the tag store but the valid and owned bits will be clear.

13.5.8 Backup cache deallocates through IPR access (BCFLUSH)

The Backup Cache Deallocate IPR is a write-only register which software uses to explicitly request the deallocation of a cache block. For example, this register may be used when hardware has put the cache into ETM and software wants to request writeback of the owned blocks to memory.

If there is a valid FILL_CAM entry for the same cache block which is being flushed, the flush is stalled until the fills return and the FILL_CAM entry is no longer valid.

Figure 13–43: IPRs 01400000 thru 015FFFE0 (hex), BCFLUSH

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|         SBZ           | 1| 0| 1| x|         Bcache Tag Deallocate Index        |     SBZ      | :BCFLUSH
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

When BCFLUSH is written, the Cbox accesses the tag store. If the block is invalid, no further action is taken. If the block is valid but not owned, the Cbox sends a block invalidate to the Mbox and invalidates the entry in the Bcache tag store. If the block is valid and owned, it sends a block invalidate to the Mbox, performs a writeback of the data, and invalidates the entry in the tag store.

This behavior takes place whether the cache is on, off, in ETM, or in FORCE_HIT mode. In FORCE_HIT mode, BCFLUSH does a real lookup of the tag store and does not force the access to hit. Software must take care not to force deallocates when cache state is not consistent with the state of memory. For example, when the cache is off, valid and owned bits may be set for blocks which are no longer up-to-date with respect to memory.

When a deallocate is done, the VALID and OWNED bits will be cleared as necessary, and the value of the stored TAG is modified. Its value is UNPREDICTABLE. Correct ECC is stored on the tag store entry.

A BCFLUSH operation never changes the data stored in the data RAMs.

Errors are detected and reported during BCFLUSH operations.

The index given is interpreted as in Table 13–48, based on the size of the cache.

BCFLUSH may be used when the Bcache is on, as the Pcache is kept a subset of the Bcache during these operations. However, new blocks may be allocated due to memory reads and writes as the cache is being flushed.

13.6 Cbox Control Description

The Cbox control consists of the following sections:

- Mbox Interface. Controls receiving commands from the Mbox including checking for read/write conflicts, and sending data and invalidates back to the Mbox.
- Cbox Arbiter. Decides which Cbox request should be serviced next.
- Tag Store Control. Controls access to the tag store RAMs, hit calculation, ECC generation and checking for tag RAMs, tag RAM error handling.
- Data Ram Control. Controls access to the data RAMs, ECC generation and checking for data RAMs, data RAM error handling.
- NDAL interface. Controls access to the NDAL queues and implements the NDAL protocol described in Chapter 3.

The tag store controller is a state machine which executes any of the following tasks, upon instruction from the arbiter:

- C_TAG%%DREAD_CMD. Performs a lookup for a data-stream read. Hits if tag matches and is valid.
- C_TAG%%IREAD_CMD. Performs a lookup for an instruction-stream read. Hits if tag matches and is valid. The operation may be cancelled midstream if the IREAD is aborted.
- C_TAG%%OREAD_CMD. Performs a lookup which requires ownership. Hits if tag matches and is valid and owned.
- C_TAG%%R_INVAL_CMD. Performs a cache coherency lookup as the result of an NDAL DREAD or IREAD; clears OWNED if necessary.
- C_TAG%%O_INVAL_CMD. Performs a cache coherency lookup as the result of an NDAL OREAD or WRITE; clears VALID and/or OWNED if necessary.
- C_TAG%%FILL_CMD. Sets the VALID and/or OWNED bit for a fill which has completed.
- C_TAG%%IPR_DEALLOC_WRITE_CMD. Performs a lookup for a deallocate; clears VALID and OWNED bits if the block was owned.
- C_TAG%%IPR_TAG_WRITE_CMD. Writes the tag store with given data.
- C_TAG%%IPR_TAG_READ_CMD. Reads the tag store from the location requested.

When the command given has been executed, the tag store controller notifies the arbiter that it has finished.

The data RAM controller is a state machine which executes any of the following tasks, upon instruction from the arbiter:

- C_DAT%%DREAD_CMD. Reads four quadwords of data-stream data from the Bcache and sends them to the Mbox interface.
- C_DAT%%IREAD_CMD. Reads four quadwords of instruction-stream data from the Bcache and sends them to the Mbox interface. The operation may be cancelled midstream if the Iread is aborted.
- C_DAT%%WB_CMD. Reads four quadwords of data from the Bcache and sends them to the WRITEBACK_QUEUE.
- C_DAT%%RM_WRITE_CMD. Performs a read-modify-write operation on the Bcache quadword.

- C_DAT%%WRITE_BM0_CMD. Performs a full quadword write on the Bcache.
- C_DAT%%FILL_CMD. Writes fill data into the Bcache; merges write data with the fill if necessary.

When the command given has been executed, the data RAM controller notifies the arbiter that it has finished.

The arbiter looks at the DREAD_LATCH, the IREAD_LATCH, the WRITE_QUEUE, and incoming transactions from the CBOX_BIU_INTERFACE to decide which to service next. It notifies the tag store controller and data RAM controller of which command to execute next.

Fills and cache coherency requests both arrive in the NDAL_IN_QUEUE and are sent to the Cbox proper through the CBOX_BIU_INTERFACE. They are processed in order; therefore, one does not have priority over the other.

When a transaction such as a read miss causes a cache block to be deallocated, the deallocate always takes place as the next data RAM transaction. Transactions in the CBOX_BIU_INTERFACE take next-highest priority. In the normal case, the DREAD_LATCH takes next priority, the IREAD_LATCH next, and the WRITE_QUEUE takes lowest priority. These priorities change if there are special circumstances, as shown in the tables which follow.

Table 13–49: Cbox Task Priority Under Normal Conditions.

Priority	Source of Transaction
1	Deallocate caused by previous transaction.
2	CBOX_BIU_INTERFACE (Fills and cache coherency requests)
3	DREAD_LATCH
4	IREAD_LATCH
5	WRITE_QUEUE

Table 13–50: Cbox Task Priority When DWR_CONFLICT Bits are Set in the WRITE_QUEUE.

Priority	Source of Transaction
1	Deallocate caused by previous transaction.
2	CBOX_BIU_INTERFACE (Fills and cache coherency requests)
3	IREAD_LATCH
4	WRITE_QUEUE
5	DREAD_LATCH - not serviced until DWR_CONFLICT bits are clear

Table 13–51: Cbox Task Priority When IWR_CONFLICT Bits are Set in the WRITE_QUEUE.

Priority	Source of Transaction
1	Deallocate caused by previous transaction.
2	CBOX_BIU_INTERFACE (Fills and cache coherency requests)
3	DREAD_LATCH
4	WRITE_QUEUE
5	IREAD_LATCH - not serviced until IWR_CONFLICT bits are clear

Table 13–52: Cbox Task Priority When a DREAD_LOCK is in progress until the WRITE_UNLOCK is done.

Priority	Source of Transaction
1	Deallocate caused by previous transaction.
2	CBOX_BIU_INTERFACE (Fills and cache coherency requests)
3	WRITE_QUEUE - the WRITE_UNLOCK corresponding to the DREAD_LOCK is the only write which will arrive unless an error occurs; in this case the IPR_WRITE clearing the RDLK bit in the FILL_CAM is the next write to arrive.
4	DREAD_LATCH - not serviced until the WRITE_UNLOCK completes or the FILL_CAM RDLK bit is cleared.
5	IREAD_LATCH - not serviced until the WRITE_UNLOCK completes or the FILL_CAM RDLK bit is cleared.

There are various resources in the Cbox which must be available for the start of a transaction. The necessary conditions vary, depending on the transaction in question.

Necessary conditions before servicing a fill from the CBOX_BIU_INTERFACE are as follows:

1. The data RAMs and the tag store must be free. The tag store is only strictly necessary for the last fill but for implementation simplicity, both are required for all fills.
2. The WRITEBACK_QUEUE must not be full. A writeback may be necessary at the completion of the fill.

Necessary conditions before servicing a cache coherency request from the CBOX_BIU_INTERFACE are as follows:

1. The tag store must be free.
2. The WRITEBACK_QUEUE must not be full.

Necessary conditions before servicing a transaction from the DREAD_LATCH or the IREAD_LATCH are as follows:

1. The data RAMs and the tag store must be free.
2. A FILL_CAM entry must be available, in case the read misses.
3. There must be an available entry in the NON_WRITEBACK_QUEUE, in case the read misses.
4. There must be no valid entry in the FILL_CAM for the same cache block as that of the new request.

5. There must be no RDLK bit set in the FILL_CAM, indicating that a READ_LOCK - WRITE_UNLOCK sequence is in progress.
6. There must be no block conflict with any WRITE_QUEUE entry.
7. The WRITEBACK_QUEUE must not be full.

Necessary conditions before servicing a full quadword write from the WRITE_QUEUE are as follows:

1. The tag store must be free.
2. If a read lock is not outstanding, a FILL_CAM entry must be available, in case the write misses and requires an OREAD.
3. If a read lock is not outstanding, there must be an available entry in the NON_WRITEBACK_QUEUE, in case the write misses.
4. There must be no valid entry in the FILL_CAM for the same cache block as that of the new request, unless the new request is a WRITE_UNLOCK.
5. If there is a READ_LOCK in the FILL_CAM, the fills for the READ_LOCK must have completed.
6. The WRITEBACK_QUEUE must not be full.

The tag store lookup for a full quadword write may be done while the data RAMs are busy with another transaction. When the data RAMs free up, the full quadword write is done. If full quadword writes are streaming through the WRITE_QUEUE, this effectively pipelines the tag store accesses and the data RAM accesses so that the writes take place at the maximum write rep rate of the data RAMs. This would not be the case if the arbiter required both the data RAMs AND the tag store to be free before starting the full quadword write.

Necessary conditions before servicing any WRITE_QUEUE entry other than a full quadword write are as follows:

1. The tag store and the data RAMs must be free.
2. If a read lock is not outstanding, A FILL_CAM entry must be available, in case the write misses and requires an OREAD.
3. If a read lock is not outstanding, there must be an available entry in the NON_WRITEBACK_QUEUE, in case the write misses.
4. There must be no valid entry in the FILL_CAM for the same cache block as that of the new request, unless the new request is a WRITE_UNLOCK.
5. If there is a READ_LOCK in the FILL_CAM, the fills for the READ_LOCK must have completed.
6. The WRITEBACK_QUEUE must not be full.

From the above lists, the following is true:

1. When the data RAMs are busy, the only tag store operations which may proceed are cache coherency requests and full quadword write requests.
2. No transaction from the Mbox which produces a block conflict with the FILL_CAM may proceed, except a WRITE_UNLOCK. This includes I/O space transactions and IPR transactions, for implementation simplicity.

13.7 Transaction Descriptions

13.7.1 IPR Reads and IPR Writes

These transactions are described in Section 13.5.

13.7.2 I/O Space

I/O space references are recognized when address bits <31:29> are equal to all ones. Address bits <31:0> are used for I/O space reads and writes, which may reference bytes. All bits of the address are driven onto the NDAL.

In addition, the byte enable field is valid for all I/O space reads and writes, as described in Chapter 3. When the Cbox receives an I/O space read or write, it passes the byte enable from the Mbox out through the BIU to the NDAL.

I/O space references are never cached in the Bcache. All such references are passed directly to the NDAL. I/O space fill data which returns is passed directly to the Mbox.

I/O space references are always quadword length. When the quadword returns on the NDAL, the Cbox returns it directly to the Mbox and asserts C%LAST_FILL_H so the Mbox does not expect any more fills.

I/O space references also result from IPR_READs and IPR_WRITEs to the Cbox which are not in Cbox register space. The Cbox converts these to I/O space reads and writes, as described in Section 13.5.

Before an I/O space read is allowed to proceed, the WRITE_QUEUE is flushed. I/O space writes are naturally ordered with respect to previous I/O space writes since they go into the WRITE_QUEUE behind any previous I/O space writes. They are also ordered with respect to previous reads and subsequent reads through the write conflict bit mechanism.

There are situations where I/O space writes will appear out of order with respect to memory space writes. See Section 13.14 for an explanation of when this may happen.

READ_LOCKs and WRITE_UNLOCKs to I/O space are not supported by the Cbox. If software issues these transactions through the Mbox, the Cbox converts them to normal DREADs and WRITEs on the NDAL.

13.7.3 Clear Write Buffer

In previous systems, Clear Write Buffer (CWB) was implemented as a separate command. NVAX implements this as an IPR read or write which the Cbox converts into an I/O space read or write on the NDAL. As this transaction passes through the Cbox, it has the effect of clearing previous entries in the WRITE_QUEUE, the NON_WRITEBACK_QUEUE, and the WRITEBACK_QUEUE.

An IPR_READ to clear the write buffers causes all the DWR_CONFLICT and IWR_CONFLICT bits in the WRITE_QUEUE to be set. All writes are flushed as top priority, and then the I/O space read is issued to the NDAL and system. Which device responds to the read is system-dependent.

An IPR_WRITE to clear the write buffers goes into the WRITE_QUEUE. If any reads are outstanding, they complete first due to their higher priority and then the writes complete. If a new read arrives while the IPR_WRITE is still in the WRITE_QUEUE, the conflict bit is set for that entry so the read does not complete until after the IPR_WRITE to clear the write buffer. After that IPR_WRITE completes, read/write priority goes back to the default behavior.

The Clear Write Buffer has the effect of clearing both the WRITEBACK_QUEUE and the NON_WRITEBACK_QUEUE, as follows: the CWB, whether issued as an IPR_READ or an IPR_WRITE, enters the NON_WRITEBACK_QUEUE. Since the WRITEBACK_QUEUE takes priority over the NON_WRITEBACK_QUEUE, any previous writebacks will be issued to the NDAL before the CWB is issued from the NON_WRITEBACK_QUEUE. Any entries which were already in the NON_WRITEBACK_QUEUE will be issued before the CWB as transactions in the queue are always issued in order. Thus, before the CWB completes, both outgoing NDAL queues are flushed of all previous transactions. If the CWB is issued as an IPR_READ, software receives positive acknowledgement that the queues were cleared when the fill returns.

The IPR_WRITE is issued to the NDAL as an I/O space write. As with the I/O space read to clear the write buffers, the device which responds is system-dependent.

13.7.4 Memory Read Hit

Several different kinds of memory reads may arrive from the Mbox, as shown in the following table.

Read	Cbox action
IREAD	hits if tag matches and valid bit is set
DREAD	hits if tag matches and valid bit is set
DREAD_MODIFY	hits if tag matches and valid bit is set
DREAD_LOCK	hits if tag matches, valid bit is set, and ownership bit is set

When the Mbox asserts **M%CBOX_REF_ENABLE_L**, the Cbox takes the command from **M%S6_CMD_H**. If the backup cache is occupied with another transaction, the Cbox puts an IREAD into the IREAD_LATCH or a DREAD into the DREAD_LATCH for later processing. Otherwise, the read bypasses the read latches and is started immediately.

When both the tag store and the data RAMs are free, the transaction starts. The tag lookup is done in parallel with the data lookup. If the read hits, data is driven from the backup cache RAMs back through the CM_OUT_LATCH. The fill command is sent to the Mbox on **C%CBOX_CMD_H<1:0>**. Two cycles later, the Pcache fill is done while the Cbox drives data onto **B%S6_DATA_H<63:0>**.

Using the fastest RAM speed configuration, the backup cache access incurs an additional 4-cycle latency penalty beyond the Pcache access. Each subsequent quadword in the block takes an extra two cycles from the previous quadword.

On a read hit in the backup cache, the requested quadword is always returned first to the Mbox. The subsequent quadwords are sent in wrapped order as shown in Table 13–53.

Table 13–53: Order of quadwords read from the Bcache

Requested QW	2nd QW returned	3rd QW returned	4th QW returned
QW0	QW1	QW2	QW3
QW1	QW2	QW3	QW0
QW2	QW3	QW0	QW1
QW3	QW0	QW1	QW2

13.7.5 Read Miss and Fill

At the same time the tag store access is done for a read, the address is put in the FILL_CAM. If the read misses, that entry is validated and the address is sent to the NON_WRITEBACK_QUEUE.

If the read command was DREAD_MODIFY and missed, it is converted to an OREAD on the NDAL. All other reads are sent as either IREADs or DREADs on the NDAL.

From the NON_WRITEBACK_QUEUE the request goes across the NDAL to the memory interface. When the memory interface returns the fill, the Cbox puts the fill into the NDAL_IN_QUEUE. Since the block size is 32 bytes and the NDAL is 8 bytes wide, four fill transactions on the NDAL result from the read request.

The arbiter services the CBOX_BIU_INTERFACE, and thus the fill, as highest priority. At this time, Cbox control takes the fill from the CBOX_BIU_INTERFACE and puts the data in the CM_OUT_LATCH. At the same time it starts writing the backup cache RAMs with the data, which takes at least three cycles, depending on RAM access time. The fill data is driven to the Mbox from the CM_OUT_LATCH as described in the cache hit section preceding.

As fill data returns, the Cbox keeps track of how many quadwords have been received with a two-bit counter in the FILL_CAM. If two read misses are outstanding, fills from the two misses may return interleaved, so each entry in the FILL_CAM has a separate counter. When the last quadword of a read miss arrives, the new tag is written and the valid bit is set in the cache. The owned bit is set if the fill was for an Ownership Read. The FILL_CAM is made available for the next cache miss.

If the RIP or OIP bit is set (and DNF is not set) in the FILL_CAM when the last fill returns, the arbiter immediately notifies the tag store control to start a cache coherency transaction on that block; nothing intervenes between the last fill and the cache coherency transaction.

13.7.6 Write Hit

A write from the WRITE_QUEUE is begun by accessing the tag store. It is a write hit if the tag matches, the valid bit is set, and the ownership bit is set. In this case the write data may be written into the data RAMs. The data RAMs are not accessed for the write until it is determined that the write hit.

The write is somewhat complicated because we have ECC across 8 bytes in the data RAMs. If all bytes in the quadword are not to be written with new data, the old data is read out of the data RAMs during the tag store lookup and before the write is done. The new data is merged with the old so that ECC can be calculated across the new quadword. This action is known as read-modify-write.

If byte enable indicates that the write is a full quadword write, the read-modify-write is not necessary. In this case, the tag store lookup may proceed even if the data RAMs are not available; when the RAMs then become available, the write is done (assuming the tag store access resulted in hit-owned). This allows sequential full quadword writes to be effectively pipelined, as the tag store lookup for the next write may proceed while the current write is being done into the data RAMs. If the fastest RAM configuration is used, this achieves a three-cycle repetition rate for full quadword writes.

When the write is complete, the entry is removed from the WRITE_QUEUE.

13.7.7 Write Miss

If the tag store lookup for a write is done and the ownership bit is not set or the tag does not match, an ownership read is issued to the memory subsystem through the NON_WRITEBACK_QUEUE. At the same time, the new tag is written to the backup cache tag store with cleared VALID and OWNED bits. When the requested quadword returns through the NDAL_IN_QUEUE, the write data is merged with the fill data, ECC is calculated, and the new data is written to the cache RAMs. At this time the write is removed from the WRITE_QUEUE. When the fourth quadword returns, the valid bit and the ownership bit are set in the tag store. None of the fill data is sent to the Mbox, since the request originated from a write rather than from an Mbox read.

13.7.8 Deallocates Due to CPU Reads and Writes

When any tag lookup for a read or a write results in a miss, the cache block is deallocated to allow the fill data to take its place. If the block is not valid, no action is taken for the deallocate. If the block is valid but not owned, the block is invalidated in the backup cache tag store and an invalidate is sent to the Pcache. If the block is valid and owned, the block is written back to memory, invalidated in the tag store, and an invalidate is sent to the Pcache. The Hexaword Disown Write command is used to write the data back.

If a writeback is necessary, it is done immediately after the read or write miss occurs. The miss and the deallocate are contiguous events and are not interrupted for any other transaction.

When the block is invalidated or deallocated at the time of the miss, the VALID and OWNED bits are cleared. The TAG is written with a value corresponding to the address of the read or write which just missed. When the fill returns, the VALID and OWNED bits are written appropriately.

The four quadwords for the deallocate are read out from the bcache in the order shown in Table 13–53. They are driven on the NDAL in order from QW0 to QW3, however, as required by the NDAL protocol for hexaword writes.

13.7.9 DREAD_LOCK and WRITE_UNLOCK

The Cbox receives DREAD_LOCK/WRITE_UNLOCK pairs from the Mbox. It never issues those commands on the NDAL. The Cbox always uses Ownership Read-Disown Write on the NDAL and depends on use of the ownership bit in memory to accomplish interlocks.

When the cache is on, a DREAD_LOCK which produces an owned hit in the backup cache causes no memory access. All four quadwords are read out of the Bcache and sent to the Mbox. The address is placed in the FILL_CAM to prevent any access of the block until the WRITE_UNLOCK is done.

A DREAD_LOCK which does not produce an owned hit in the backup cache results in an OREAD on the NDAL, whether the cache is on or off. When the cache is on, the WRITE_UNLOCK is written into the backup cache and is only written to memory if requested through a coherence transaction or due to a deallocate. When the cache is off, the WRITE_UNLOCK becomes a Quadword Disown Write on the NDAL.

When a DREAD_LOCK arrives in the DREAD_LATCH, the WRITE_QUEUE is flushed before the DREAD_LOCK is started. All transactions from the IREAD_LATCH or the DREAD_LATCH are prevented until the WRITE_UNLOCK takes place or until the RDLK bit in the FILL_CAM is cleared through an IPR_WRITE to the CEFSTS IPR.

During READ_LOCK/WRITE_UNLOCK processing, the NDAL_IN_QUEUE is serviced normally, so if the cache is on, the NDAL may see some writebacks while the DREAD_LOCK/WRITE_UNLOCK is in progress.

When the Bcache is running in normal mode, a WRITE_UNLOCK is not looked up in the tag store as it is guaranteed to be owned in the cache. The arbiter initiates a read-modify-write directly to the data RAMs without any tag store access at all. If the Bcache is in ETM, the WRITE_UNLOCK is looked up, as the block may or may not be owned in the cache.

When the Bcache is off, a WRITE_UNLOCK which is done without a preceding READ_LOCK will be sent directly to the NDAL. In any other mode of Bcache operation, the WRITE_UNLOCK is expected to be preceded by a READ_LOCK. When the cache is off, a WRITE_UNLOCK without a preceding READ_LOCK may be useful for error handling (this is not currently implemented in the microcode).

13.8 Cache Coherency

Since NVAX is used in multiprocessor systems, cache coherency requests requiring invalidates and/or writebacks arrive on the NDAL. These may require action in the Bcache and/or the Pcache. Under normal conditions, the Cbox ensures that the Pcache is a subset of the Bcache, as explained below. Thus, it is able to filter invalidate requests so that not all are sent to the Pcache.

Table 13–54 shows the actions taken in the Bcache, based on the NDAL command which arrives and matches a cache block.

Table 13–54: NVAX Backup Cache Invalidates and Writebacks

NDAL Command	Invalid block	Valid & Unowned	Valid & Owned
IREAD,DREAD	-	-	Writeback, set Bcache to valid-unowned state
OREAD	-	Invalidate	Writeback, Invalidate
WRITE	-	Invalidate	Writeback, Invalidate
WDISOWN	-	-	-

Whenever an invalidate is necessary in the Bcache, according to Table 13–54, an invalidate is also sent to the Pcache. .

Invalidates are sent to the Pcache under the following circumstances:

1. When an invalidate is necessary in the Bcache, due to a cache coherency request, the invalidate is also forwarded to the Pcache.
2. When a cache miss causes a Bcache deallocate, a corresponding invalidate is forwarded to the Pcache.
3. When a write to BCFLUSH causes a bcache deallocate, a corresponding invalidate is forwarded to the Pcache.
4. When a OREAD or WRITE cache coherency request matches an entry in the FILL_CAM, the invalidate is forwarded immediately to the Pcache. When the last fill returns, a second invalidate is forwarded to the Pcache.

5. When the Bcache is off or in FORCE_HIT mode, ALL cache coherency requests result in invalidates to the Pcache. It is not strictly necessary to send invalidates for IREAD and DREAD cache coherency requests, as multiple caches may contain read-only copies of data, but for implementation reasons they ARE sent as invalidates to the Pcache.
6. When the Bcache is in ETM, all OREAD and WRITE cache coherency requests result in invalidates to the Pcache. (IREAD and DREAD cache coherency requests do not result in invalidates to the Pcache.) A second invalidate is passed to the Pcache if the normal Bcache lookup conditions are met.

NOTE

When a cache coherency request hits in the cache and either VALID or OWNED is modified, the tag which is written to the cache is the same as the tag which was there originally.

13.9 Abnormal conditions

This section describes the various modes of Bcache behavior as well as Cbox response when it detects an error.

The Bcache has four operating states which are controlled by the following bits in the CCTL register: ENABLE, FORCE_HIT, SW_ETM, and HW_ETM. The four states are ON, OFF, ETM, and FORCE_HIT. The four states are determined and prioritized as follows:

1. OFF. If the ENABLE bit is cleared in CCTL, the Bcache is OFF and those conditions take precedence.
2. FORCE_HIT. If the ENABLE bit is set and FORCE_HIT is set, the Bcache is in FORCE_HIT mode and those conditions take precedence.
3. ETM. If the ENABLE bit is set, FORCE_HIT is cleared, and either SW_ETM or HW_ETM is set, the cache is in ETM mode and those conditions take precedence.
4. ON. If the ENABLE bit is set and FORCE_HIT, SW_ETM, and HW_ETM are cleared, the cache is ON.

The ON state is the normal operating condition of the cache. OFF, FORCE_HIT, and ETM modes are described in the sections which follow. A summary of the backup cache behavior when it is ON and incurring no errors is given in Table 13–55.

Table 13–55: Backup cache behavior while it is ON

Cache	Cache Response				
Transaction	Miss Invalid	Miss Valid	Miss Owned	Hit Valid	Hit Owned
CPU IREAD, DREAD	READ memory	Read memory, Pcache inval	Read memory, Pcache inval, Bcache dealloc	Read cache	Read cache
CPU Read Modify	OREAD memory	ORead memory, Pcache inval	OREAD memory, Pcache inval, Bcache dealloc	Read cache	Read cache
CPU READ_LOCK	OREAD memory	ORead memory, Pcache inval	OREAD memory, Pcache inval, Bcache dealloc	OREAD memory, Pcache inval	Read cache
CPU Write	OREAD memory	OREAD memory, Pcache inval	OREAD memory, Pcache inval, Bcache dealloc	OREAD memory, Pcache inval	Write cache
CPU WRITE_UNLOCK	No tag store lookup; write Bcache unconditionally				
Fill for OREAD for Write	Write cache with fill data and write data; set TS valid-owned				
Fill for OREAD	Write cache with fill data; set TS valid-owned				
Fill for READ	Write cache with fill data; set TS valid				
NDAL IREAD, DREAD	No action for a miss			No Action	Writeback, set Bcache valid-unowned
NDAL OREAD, WRITE	No action for a miss			Bcache inval, Pcache inval	Writeback, Bcache inval,Pcache inval

13.9.1 Cbox Behavior When the Backup Cache is OFF

The backup cache may be off for three reasons: the chip has just powered up, the system contains no backup cache, or software has disabled the cache by clearing the ENABLE bit in the Cbox control register.

When the cache is off, no accesses to the backup cache are done. Errors are not detected and cache state is UNCHANGED unless explicitly changed by software through IPR reads and writes.

When the backup cache is off, all Ownership-Invalidate cache coherency requests (as the result of OREADs or WRITEs) which arrive are forwarded as invalidates to the Mbox, as the data may be valid in the Pcache. All reads from the Mbox go directly to the NON_WRITEBACK_QUEUE, and an entry in the FILL_CAM is allocated. Fills which return are sent directly to the Mbox without accessing the Bcache, and when the last fill for a block arrives, the FILL_CAM entry is cleared. All writes except WRITE_UNLOCKs go directly to the NON_WRITEBACK_QUEUE. [1]

When the cache is off, a DREAD_LOCK/WRITE_UNLOCK pair from the Mbox becomes Hexaword Ownership Read/Quadword Disown Write on the NDAL.

All writes issued from NVAX when it is operating without a backup cache are of quadword length. Memory reads are of hexaword length since the Pcache block size is a hexaword. Even if the Pcache is off, a hexaword of data is returned to the Mbox.

A DREAD_MODIFY command from the Mbox normally becomes an OREAD on the NDAL when it misses in the cache. However, when the cache is off, a normal DREAD is used on the NDAL.

[1] If **P%CPU_WB_ONLY_L** is asserted, the WRITE_UNLOCK must be allowed to proceed. Only the WRITEBACK_QUEUE continues when **P%CPU_WB_ONLY_L** is asserted, so the WRITE_UNLOCK must go through the WRITEBACK_QUEUE.

13.9.2 Cbox Behavior When the Backup Cache is in FORCE_HIT Mode

FORCE_HIT mode is intended to be used for testing purposes only. It is used when the cache is enabled.

When FORCE_HIT is set, all memory space reads and writes to the Bcache, both Istream and Dstream, are forced to hit. Tag store state is not changed at all; the data RAMs are accessed as if the tag store access produced an owned-valid hit. Cache coherency transactions are treated as they are when the cache is off: they are not looked up in the backup cache, they are all forwarded to the Mbox, and cache state is not changed as the result of the cache coherency requests.

When the Bcache is in FORCE_HIT mode, deallocates are not done. Even if the tag matches and the VALID and OWNED bits are set, the block is not written back. The implication of this is that if FORCE_HIT mode is being used while running in a multiprocessor environment, the Bcache must be flushed of all owned blocks beforehand.

Tag store and data RAM ECC errors are detected in FORCE_HIT mode if DISABLE_ERRORS in the CCTL register is not set, resulting in the usual error handling.

Suppose the ECC logic for the data RAMs is to be tested. Put the cache in FORCE_HIT mode. Set SW_ECC in the Cbox control register. Write the desired ECC into BCDECC. Do a Dstream write to the desired location, and the location will be written using ECC from BCDECC rather than from Cbox-generated ECC. Suppose the ECC written is such that when the data is read, an ECC error will be flagged.

Now perform a read of the location while FORCE_HIT is still set. The read will result in an ECC error, showing that the logic is working correctly. The data ram error registers may be read and will correspond to the induced error.

13.9.3 Cbox Behavior When the Backup Cache is in Error Transition Mode

When the Cbox detects certain errors, as described in Chapter 3 and Section 13.4.2, it puts itself into Error Transition Mode.

The goals of the Cbox design during ETM are the following:

1. Preserve the state of the cache as much as possible for diagnostic software.
2. Honor Mbox references which hit owned blocks in the backup cache since this is the only source of data in the system.
3. Respond to NDAL cache coherency requests normally.

Once the Cbox enters Error Transition Mode, it remains in ETM until software explicitly disables or enables the cache. To ensure cache coherency, the cache must be completely flushed of valid blocks before it is re-enabled because some data can become stale while the cache is in ETM.

Table 13–56 describes how the backup cache behaves while it is in ETM.

Table 13–56: Backup cache behavior during ETM

Cache	Cache Response		
Transaction	Miss	Valid hit	Owned hit
CPU IREAD,DREAD	Read memory	Read memory	Read cache
CPU Read Modify	Read memory	Read memory	Read cache
CPU READ_LOCK	OREAD memory	OREAD memory	Oread memory, Bcache dealloc[1]
CPU Write	Write memory	Write memory	Write memory, Bcache dealloc[1]
CPU WRITE_UNLOCK	Write memory	Write memory	Write cache[1]
Fill (from read started before ETM)	Normal cache behavior		
Fill (from read started during ETM)	Do not update backup cache; return data to Mbox		
NDAL cache coherency request	Normal cache behavior except that o-inval always goes to Pcache[2]		

[1] Done to preserve write ordering; no invalidate is sent to the Pcache. For the READ_LOCK (or WRITE), the block writeback may be done before OR after the OREAD (or WRITE).

[2] The tag store controller looks up the invalidate request normally; if the lookup was an o-inval (due to an OREAD or a WRITE on the NDAL), the Cbox arbiter unconditionally forwards an invalidate to the Pcache. If the hit conditions are met in the cache, a second invalidate for the same block is forwarded to the Pcache (the tag store controller behaves as it does in normal mode.)

Any reads or writes which do not hit valid-owned during ETM are sent to memory: read data is retrieved from memory, and writes are written to memory, bypassing the cache entirely.

The cache supplies data for Ireads, Dreads, and Dread Modifys which hit valid-owned; this is normal cache behavior.

If a write hits a valid-owned block in the cache, the block is written back to memory and the write is also sent to memory. The write leaves the Cbox through the NON_WRITEBACK_QUEUE, enforcing write ordering with previous writes which may have missed in the cache.

If a READ_LOCK hits valid-owned in the cache, a writeback of the block is forced and the READ_LOCK is sent to memory (as an OREAD on the NDAL). This behavior enforces write ordering between previous writes which may have missed in the cache and the WRITE_UNLOCK which will follow the READ_LOCK.

The write ordering problem to which the previous two paragraphs allude is as follows: Suppose the cache is in ETM. Also suppose that under ETM, writes which hit owned in the cache are written to the cache while writes which miss are sent to memory. Write A misses in the cache and is sent to the non-writeback queue, on its way to memory. Write B hits owned in the cache and is written to the cache. A cache coherency request arrives for block B and that block is placed in the writeback queue. If Write A has not yet reached the NDAL, Writeback B can pass it since the writeback queue has priority over the non-writeback queue. If that happens, the system sees write B while it is still reading old data in block A, because write A has not yet reached memory.

Referring again to Table 13–56, note that a WRITE_UNLOCK that hits owned during ETM is written directly to the cache. There is only one case where a WRITE_UNLOCK will hit owned during ETM: if the READ_LOCK which preceded it was performed before the cache entered ETM. (Either the READ_LOCK itself or an invalidate performed between the READ_LOCK and the WRITE_UNLOCK caused the entry into ETM.) In this case, we know that no previous writes are in the non-writeback queue because writes are not put into the non-writeback queue when we are not in ETM. (There may be I/O space writes in the non-writeback queue but ordering with I/O space writes is not a constraint.) Therefore there is not a write ordering problem as in the previous paragraph.

Table 13–56 shows that during ETM, cache coherency requests are treated as they are during normal operation.

Fills as the result of any type of read originated before the cache entered ETM are processed in the usual fashion. If the fill is as a result of a write miss, the write data is merged, as usual, as the requested fill returns. Fills caused by any type of read originated during ETM are not written into the cache or validated in the tag store.

During ETM, the state of the cache is modified as little as possible. Table 13–57 shows how each transaction modifies the state of the cache.

Table 13–57: Backup cache state changes during ETM

Cache	Cache State Modified		
Transaction	**Miss**	**Valid hit**	**Owned hit**
CPU IREAD,DREAD, Read Modify	None.	None.	None.
CPU READ_LOCK	None.	None.	Clear VALID & OWNED; change TS_ECC accordingly.
CPU Write	None.	None.	Clear VALID & OWNED; change TS_ECC accordingly.
CPU WRITE_UNLOCK	None.	None.	Write new data, change DR_ECC accordingly.
Fill (from read started before ETM)	-Write new TS_TAG, TS_VALID, TS_OWNED, TS_ECC, DR_DATA, DR_ECC-		
Fill (from read started during ETM)	None.		
NDAL cache coherency request	Clear VALID & OWNED; change TS_ECC accordingly		

13.9.4 Cbox transition into Error Transition Mode

When the BIU encounters an error which induces ETM, it sends an explicit transaction to the arbiter requesting that the Cbox enter ETM. When the arbiter services this transaction, CCTL<HW_ETM> is set. The next transaction serviced by the arbiter will be under ETM.

When the backup cache tag store or data RAM controller encounters an ETM-inducing error, it sets CCTL<HW_ETM> immediately.

The arbiter picks up the new value of CCTL<HW_ETM> whenever it starts a new transaction. The tag store controller picks up the new value whenever the arbiter instructs it to start a new transaction. For a given transaction, the arbiter and the tag store always see the same value of ETM. Since they pick up the state of ETM at the beginning of every transaction, the Cbox always enters ETM in a predictable way.

Although the data ram controller may cause the assertion of HW_ETM, it does not use ETM in processing its transactions.

In general, if a transaction starts when the Bcache is operating normally, and it encounters an ETM-inducing error, the next transaction is handled in ETM. There is one exception: If a read is looked up in the tag store and hits, the data RAM controller looks up the data in the backup cache. While the data is being read out of the RAMs, the tag store controller may start a lookup for a quadword write. If the quadword write hits, the write WILL be done to the backup cache even if the read data encounters an ETM-inducing error before the write is done to the Bcache. This sequence would be as follows:

1. Tag store lookup and Data RAM lookup for READ A start.
2. Tag store lookup for READ A completes.
3. Tag store lookup for Quadword Write B starts.
4. Data RAM lookup for Read A encounters an ETM-inducing error.
5. Tag store lookup for Quadword Write B completes; it was a hit.
6. Data RAM lookup for Read A completes.
7. Data RAM write for Quadword Write B is carried out to the Bcache.

Quadword Write B completed as if the Bcache were operating normally. If the tag store lookup for the Quadword Write had not started until after the ETM-inducing error had been encountered, then the Quadword Write would have been carried out under ETM, and the write would have been done directly to memory.

13.9.5 How to turn the Bcache off

Because the Bcache is a writeback cache, care must be taken to maintain cache coherency when turning it off.

If the cache is running normally and software wishes to turn it off, it must do the following:

1. Write CCTL to set SW_ETM. In this mode, the Bcache will not allocate any new blocks and will send all cache coherency requests to the Mbox as invalidates.
2. Use the BCFLUSH register to flush all owned blocks out of the cache.
3. Turn off the Bcache by writing CCTL to clear ENABLE and SW_ETM simultaneously. If an error was encountered during the deallocate process, HW_ETM may be set and if so, should be cleared as well.

If the Bcache encounters an uncorrectable ECC error, the Cbox sets HW_ETM in the CCTL register. If software wishes to turn off the cache, it must do the following:

1. Use the BCFLUSH register to flush all owned blocks out of the cache.
2. Write CCTL to clear ENABLE and clear HW_ETM simultaneously. This turns off the Bcache.

If Bcache errors are happening, but only in part of the cache, software may be able to avoid the errored portion of the cache by disabling it through use of the SIZE field in CCTL. If part of the cache is failing, a smaller cache size may be selected so that only part of the cache RAMs are being used. The cache must be flushed before changing the cache size so that the tags are correct.

This only works if the smallest cache size is not being used to begin with, and if the failing areas of cache do not fall within the range of the smaller cache size selected.

13.9.6 How to turn the Bcache on

When NVAX powers up, garbage data is stored in the Bcache tags and data. This would result in ECC errors if the cache were turned on immediately.

Through IPR writes, every Bcache tag store entry must be written with cleared OWNED and VALID bits. The value written to the TAG is irrelevant, as long as correct ECC is written to the TAG store.

The Bcache data RAMs must also be initialized with correct ECC on powerup. FORCE_HIT mode may be used to initialize the Bcache data RAMs with correct ECC. If full quadword writes are used, no data RAM errors will be detected during this process, since the RAMs are written without being read first. If partial quadword writes are used, errors will be detected because of the read-modify-write which is necessary. If the programmer sets the DISABLE_ERRORS bit in the CCTL register, the Cbox will ignore these errors.

Once the tag store and data RAMS have been initialized, the cache may be enabled by setting ENABLE in the CCTL register.

If the Bcache is in ETM, it may be incoherent with respect to other CPUs and memory because of how it treats writes which hit valid but not owned in the cache (see Table 13–56). In addition, the Pcache, if enabled, is no longer a subset of the backup cache. The procedure for turning on the Pcache and the Bcache described in Chapter 16 must be followed.

If the Bcache is operating normally and is turned off for some reason, the programmer must ensure that when it is reenabled, all the OWNED and VALID bits are cleared.

13.9.7 Assertion of P%CPU_WB_ONLY_L

When **P%CPU_WB_ONLY_L** is asserted, NVAX may only arbitrate in order to issue Disown Writes on the NDAL. When **P%CPU_WB_ONLY_L** is asserted, the Cbox continues to process transactions from the NDAL_IN_QUEUE normally, performing writebacks as necessary. With one exception described below, the Cbox arbiter prevents all new reads and writes from the Mbox while **P%CPU_WB_ONLY_L** is asserted. Therefore, if **P%CPU_WB_ONLY_L** is asserted for long periods of time, CPU performance could be adversely impacted.

The exception to the rule is the following: If a READ_LOCK from the Mbox is in progress when **P%CPU_WB_ONLY_L** is asserted, the WRITE_UNLOCK from the write queue must be allowed to complete. Otherwise, deadlock could occur if the system asserted **P%CPU_WB_ONLY_L** until it received data from the WRITE_UNLOCK.

Therefore, while **P%CPU_WB_ONLY_L** is asserted, the write queue is permitted to continue if a READ_LOCK is in progress. The READ_LOCK is completed when either the WRITE_UNLOCK is issued and completed, or an "IPR WRITE_UNLOCK" to CEFSTS is issued and completed.

During the cycle in which **P%CPU_WB_ONLY_L** is asserted, the Cbox may issue a non-writeback command on the NDAL. It is up to the NDAL arbiter not to grant to NVAX again during that cycle, so that the Cbox does not issue another non-writeback command in the following cycle. If the NDAL arbiter does assert **P%CPU_GRANT_L** during the same cycle in which **P%CPU_WB_ONLY_L** is asserted, NVAX may drive another non-writeback command on the NDAL in the following cycle which was granted.

There is one interesting error case which can occur when **P%CPU_WB_ONLY_L** is asserted. It is as follows:

Normally, when the Cbox has a READ_LOCK outstanding and it receives an O_INVAL cache coherency request (OREAD or WRITE to the block), it sets the OIP bit in the FILL_CAM (O_INVAL pending). If the Cbox receives an R_INVAL cache coherency request, it sets the RIP bit in the FILL_CAM (R_INVAL pending). When the Ebox issues the WRITE_UNLOCK and the Cbox arbiter sees that RIP or OIP is set, it issues a block writeback to the NDAL. This is done even if **P%CPU_WB_ONLY_L** is asserted.

If some error occurs which prevents the Ebox from issuing the WRITE_UNLOCK, it sends the Cbox an "IPR WRITE_UNLOCK" to clear the READ_LOCK out of the FILL_CAM. This "IPR WRITE_UNLOCK" clears the FILL_CAM entry but the Cbox arbiter DOES NOT check the status of RIP and OIP to see if we need to do a writeback.

The implication is that if the Cbox is in the middle of a READ_LOCK-WRITE_UNLOCK and a cache coherency transaction arrives for the block, AND the Ebox never issues the WRITE_UNLOCK due to some error (see below), the Cbox will NOT write back that block in response to the former invalidate. (The Cbox would write the block back if a subsequent cache coherency request arrived.) The following error would cause this situation: TB parity error after issuing the read lock; Ebox S3 stall timeout after issuing the read lock; an uncorrectable error in the Backup cache data RAMs on the first quadword of the read lock.

This could cause a deadlock in a system if the system had asserted **P%CPU_WB_ONLY_L** because it was waiting for the writeback. NVAX might never issue the writeback and the Cbox stops processing after the "IPR write unlock", until **P%CPU_WB_ONLY_L** is deasserted.

One solution to the deadlock is for the system element which is waiting for the writeback to have a timeout counter, so that it does not wait forever. Once the element times out, **P%CPU_WB_ONLY_L** should be deasserted and the system can continue to operate. Or if the cache coherency transaction is reissued on the NDAL after the completion of the "IPR WRITE_UNLOCK", the Cbox WILL service it.

13.9.8 Backup Cache Errors

In general, the Cbox logs as much state as possible concerning errors and notifies the Ebox and/or Mbox that an error has occurred. For every error, the Cbox asserts either **C%CBOX_H_ERR_H** or **C%CBOX_S_ERR_H** to notify the interrupt section of a hard error or a soft error, respectively. The Cbox also notifies the Mbox if the error occurred on a fill to the Mbox.

The backup cache goes into Error Transition Mode when it detects any uncorrectable error from the cache RAMs.

Table 13–58: Backup Cache ECC Errors and NVAX CPU Error Responses

General Problem	Specific Situation and Action Taken by NVAX CPU	
Correctable ECC error in the data RAMs	read hit for writeback or read hit for deallocate IPR	Cbox asserts C%CBOX_S_ERR_H. The data for the writeback is corrected and the writeback continues normally.
	read hit for Mbox	Cbox asserts C%CBOX_S_ERR_H. C%CBOX_ECC_ERR_H is asserted to tell the Mbox to ignore the uncorrected data. When the data has been corrected, it is driven to the Mbox. Hardware does not correct the error in the cache.
	read for write hit	Cbox asserts C%CBOX_S_ERR_H. The corrected data is merged with the write data and written into the RAMs.
	miss	No error is reported.
Correctable ECC error in the tag store	any read or write except WUNLOCK (hit or miss)	Cbox asserts C%CBOX_S_ERR_H, assumes the transaction missed, and sends a READ or an OREAD to memory. If the location was owned, making a deallocate necessary, the outgoing address is corrected for the writeback. Note that if the transaction actually hit-owned, the read or oread is sent to the NDAL followed by a writeback of the same block. The errored location is corrected by hardware when the tag and valid bit are written for the fill.
	WRITE_UNLOCK	No tag store lookup is done, so this case does not occur.
	cache coherence transaction miss	Cbox asserts C%CBOX_S_ERR_H. Hardware does not correct the bad location; it may be done by software.
	cache coherence transaction hit	Cbox asserts C%CBOX_S_ERR_H. Writes the corrected tag, valid, and owned bits back into the tag store when invalidating the entry. Uses corrected address for the writeback if necessary.

Table 13–58 (Cont.): Backup Cache ECC Errors and NVAX CPU Error Responses

General Problem	Specific Situation and Action Taken by NVAX CPU	
Uncorrectable ECC error in the data RAMs (includes addressing errors)	read for writeback or deallocate IPR	Cbox asserts C%CBOX_S_ERR_H, puts backup cache into ETM. The data cycle command for the NDAL is changed to BADWDATA and the writeback continues normally.
	VALID-OWNED or VALID-UNOWNED read for Mbox	Cbox asserts C%CBOX_S_ERR_H, puts backup cache into ETM. The CM_OUT_LATCH is loaded with the data and marked bad by asserting C%CBOX_HARD_ERR_H.
	VALID-OWNED DREAD_LOCK for Mbox, first quadword fails	Cbox asserts C%CBOX_S_ERR_H, puts backup cache into ETM. The CM_OUT_LATCH is loaded with the data and marked bad by asserting C%CBOX_HARD_ERR_H. The DREAD_LOCK entry remains in the FILL_CAM until microcode issues the "IPR write unlock". If RIP or OIP is set, it is not processed.
	VALID-OWNED DREAD_LOCK for Mbox, quadword other than the first one fails	Cbox asserts C%CBOX_S_ERR_H, puts backup cache into ETM. The CM_OUT_LATCH is loaded with the data and marked bad by asserting C%CBOX_HARD_ERR_H. The Ebox/Mbox issues the WRITE_UNLOCK since data for the DREAD_LOCK was returned.
	read for write or write-unlock, valid-owned hit	Cbox asserts C%CBOX_H_ERR_H, puts backup cache into ETM. When the error is detected, write data has already been merged with the corrupted data. The Cbox inverts two of the ECC check bits (bits 3,7) which gives a high probability that when the data is read again, an uncorrectable error will be detected. See description after this table.
	miss	No error is reported.
Uncorrectable ECC error in the tag store (includes addressing errors)	read for Mbox	Cbox asserts C%CBOX_S_ERR_H, puts backup cache into ETM. The read is sent to memory; if the backup cache actually owned the block the read will time out. If fill data is returned, the fill is done to the Bcache and the fill data is sent to the Mbox.
	write	Cbox asserts C%CBOX_S_ERR_H, puts backup cache into ETM. The Oread for the write is sent to memory. If the cache actually owned the block, the read will time out and the write will then be sent to memory. The write will then time out as well unless error handling software cleans up the problem. If the cache did not own the block, the Oread will complete, the write will be merged with it, and the merged data will be written to the cache.
	WRITE_UNLOCK	No tag store lookup is done, so this case does not occur.
	cache coherence transaction	Cbox asserts C%CBOX_S_ERR_H, puts backup cache into ETM. Transaction is treated as a miss with regard to the backup cache; the invalidate is forwarded to the Mbox if the cache coherence transaction was due to an OREAD or a WRITE.

One action noted in the table deserves further explanation. When an uncorrectable ECC error is detected in the data RAMs during a read-modify-write, the Bcache controller has already begun to write the new data into the cache, overwriting the errored data. The new data may have been corrupted by the errored data which was read from the cache. If this were allowed to be written into the cache with correct ECC, it might be read back later with no errors and incorrect data would be returned to the CPU.

In order to prevent this from occurring, the Bcache controller inverts two of the checkbits which are being written to the cache to deliberately cause errored data to be written. This increases the likelihood that when the data is read back, an uncorrectable error will be detected whether the data is read back as written or with single-bit or multiple-bit errors.

Due to layout constraints, only checkbits 3,6, and 7 were potential candidates to be inverted in the circumstance described. The probabilities for reading the data back as uncorrectable are shown in Table 13–59.

Table 13–59: Probability of reading data with an uncorrectable error after writing it with inverted checkbits

Bits Inverted	no error read back	single bit error read back	double bit error read back	triple single nibble error read back	quad single nibble error read back
3, 6	1.00	.3425	.9909	.4306	.6111
3, 7	1.00	.3973	.9916	.4861	.6667
6, 7	1.00	.1233	.9878	1.0000	1.0000
3, 6, 7	0.00	.9863	.4429	1.0000	1.0000

Choosing bits 3 and 7 results in uncorrectable errors a high percentage of the time if you assume a high likelihood that the data will be read back with no error (as it would be if the original error were transient) or with a double-bit error (as it would be if the original error were a hard double-bit error).

13.9.9 Backup Cache Errors Incurred While in Error Transition Mode

Table 13–60 describes error handling when the backup cache is already in ETM.

NOTE

The table below only describes ETM error cases which differ from error handling when the cache is in normal mode.

Table 13–60: Backup Cache ECC Error handling during ETM

General Problem	Specific Situation and Action Taken by NVAX CPU	
Correctable ECC error in the tag store	WRITE_UNLOCK	C%CBOX_S_ERR_H. The error is corrected and the WRITE_UNLOCK is handled as it normally is in ETM: it is written to the Bcache if it hits owned, and it is written to memory if it misses or hits valid.
Uncorrectable ECC error in the tag store (includes addressing errors)	read for Mbox	Cbox asserts C%CBOX_S_ERR_H, puts backup cache into ETM. The read is sent to memory; if the backup cache actually owned the block the read will time out. If fill data is returned, the fill is not done to the Bcache but is sent to the Mbox.
	write	C%CBOX_S_ERR_H. The write is sent to memory. If the cache actually owned the block, the write will time out in the memory interface unless software forces the Cbox to disown the block. If the cache did not own the block, the system handles the write as it normally does for a cache which is off.
	WRITE_UNLOCK	C%CBOX_S_ERR_H. The write is sent to memory as a QW WDISOWN. Since the READ_LOCK was done just previously, memory always believes that we own the block. In most cases, the cache itself does not have a record of owning the block since a READ_LOCK to an owned block during ETM forces a writeback of the block. In these cases the WRITE_UNLOCK handling is very consistent. There is only one case where the cache does own the block: if we entered ETM on or after the READ_LOCK and before the WRITE_UNLOCK. In this case, the cache may contain previously written data which is not now reflected into memory. This may be handled by software.

13.9.10 NDAL Parity Errors

The Cbox response to NDAL parity errors is described in Chapter 3.

13.10 Testability

The testability features provided in the Cbox make key Cbox control visible for debug purposes. The testability features do not specifically address fault coverage for manufacturing, since Cbox activity is very visible on the NDAL and cache interface pins.

Many of the Cbox IPRs should be useful for testing and debug. The IPRs are described in Section 13.5. This section describes additional Cbox testability features.

13.10.1 Parallel port

The parallel port is useful for real-time debugging and for manufacturing test. The Cbox does not control any nodes using the parallel port; it is used for observation only. **C%PP_DATA_H<11:7>** are driven as shown in Table 13–61. The Mbox contains the circuitry which enables **C%PP_DATA_H<11:7>** to drive the parallel port when **T%MBOX_DR_PP_H** is asserted.

Table 13–61: Cbox Parallel Port Connections

Parallel port signal	Cbox signal	Cbox Signal Meaning
C%PP_DATA_H<11>	BC_TS_CMD<2>	Given in Table 13–62
C%PP_DATA_H<10>	BC_TS_CMD<1>	Given in Table 13–62
C%PP_DATA_H< 9>	BC_TS_CMD<0>	Given in Table 13–62
C%PP_DATA_H< 8>	DEALLOC	Asserted when the tag store starts a deallocate.
C%PP_DATA_H< 7>	BC_HIT	Backup cache hit; factors in the type of request with VALID, OWNED, and the result of the tag compare.

BC_TS_CMD<2:0> is decoded as follows:

Table 13–62: Interpretation of BC_TS_CMD<2:0>

BC_TS_CMD	Name	Tag store operation
000	DREAD	Data-stream tag lookup
001	IREAD	Instruction-stream tag lookup
010	OREAD	Ownership-read tag lookup for a write or a READ_LOCK
011	WUNLOCK	Ownership-read tag lookup for a WRITE_UNLOCK (done only under ETM)
100	R_INVAL	Cache coherency tag lookup as the result of NDAL DREAD or IREAD
101	O_INVAL	Cache coherency tag lookup as the result of NDAL OREAD or write
110	IPR_DEALLOC	Tag lookup for an explicit IPR deallocate operation
111	unused	

13.10.2 Internal scan chain

A scan chain is provided on both entries of the FILL_CAM. A Linear Feedback Shift Register is provided on this scan chain. This serves two purposes: it helps the debug effort and it increases fault coverage in manufacturing. The scan chain bits are loaded when **M%C_ISR_LOAD_L** is asserted; they are shifted out when it is deasserted. The LFSR is enabled when **M%C_ISR_LFSR_L** is asserted. When **M%C_ISR_LFSR_L** is not asserted, the scan chain becomes an observe-only register.

The FILL_CAM gives cycle-by-cycle information on what is happening in the Cbox, as every potential cache miss is loaded into the FILL_CAM before the miss actually occurs. There is information relating to cache coherency requests as well.

The Cbox scan chain covers the following bits of the FILL_CAM:

Table 13–63: FILL_CAM scan chain

Name	Extent	Type	Description
RDLK_0	0	WC	Indicates that the outstanding read is a READ_LOCK.
IREAD_0	1	RO	This is an Istream read from the Mbox which may be aborted.
OREAD_0	2	RO	This is an outstanding OREAD.
WRITE_0	3	RO	This read was done for a write.
TO_MBOX_0	4	RO	Data is to be returned to the Mbox.
RIP_0	5	RO	READ invalidate pending.
OIP_0	6	RO	OREAD invalidate pending.
DNF_0	7	RO	Do not fill - data not to be written into the cache or validated when the fill returns.
RDLK_FL_DONE_0	8	RO	Indicates that the last fill for a READ_LOCK arrived.
REQ_FILL_DONE_0	9	RO	Indicates that the requested quadword was successfully received.
COUNT_0	11:10	RO	How many of the fill quadwords have been returned successfully.
VALID_0	12	WC	Indicates that an error occurred and the register is locked.
RDLK_1	13	WC	Indicates that the outstanding read is a READ_LOCK.
IREAD_1	14	RO	This is an Istream read from the Mbox which may be aborted.
OREAD_1	15	RO	This is an outstanding OREAD.
WRITE_1	16	RO	This read was done for a write.
TO_MBOX_1	17	RO	Data is to be returned to the Mbox.
RIP_1	18	RO	READ invalidate pending.
OIP_1	19	RO	OREAD invalidate pending.
DNF_1	20	RO	Do not fill - data not to be written into the cache or validated when the fill returns.
RDLK_FL_DONE_1	21	RO	Indicates that the last fill for a READ_LOCK arrived.
REQ_FILL_DONE_1	22	RO	Indicates that the requested quadword was successfully received.
COUNT_1	24:23	RO	How many of the fill quadwords have been returned successfully.

Table 13–63 (Cont.): FILL_CAM scan chain

Name	Extent	Type	Description
VALID_1	25	WC	Indicates that an error occurred and the register is locked.

There are two FILL_CAM entries. Thirteen bits in each are covered, for a total of 26 bits in this scan path.

The Cbox scan chain is connected in the order shown in the table, with bit <0> shifted out first and sent to the Mbox scan chain. When the Cbox scan chain is in shift mode, a "0" is shifted into bit <25> of the Cbox scan chain. Bit <0> is driven onto **C%ISR2_TDO_H**, which is input to the Mbox scan chain.

13.11 Performance Monitoring

The Cbox sends two signals, **C%PMUX0_H** and **C%PMUX1_H** to the performance counters. CCTL<PM_ACCESS_TYPE> controls the mux which outputs **C%PMUX0_H**. CCTL<PM_HIT_TYPE> controls the mux which outputs **C%PMUX1_H**.

The correspondence between CCTL<PM_ACCESS_TYPE> and **C%PMUX0_H** is shown in Table 13–64.

Table 13–64: Cbox Performance Monitoring Control

CCTL: PM_ACCESS_TYPE<13:11>	Signal muxed onto C%PMUX0_H	Signal functionality
000	BC_COH	Bcache coherency access (as a result of an NDAL DREAD, IREAD, OREAD, or WRITE)
001	BC_COH_READ	Bcache coherency access as a result of an NDAL DREAD or IREAD
010	BC_COH_OREAD	Bcache coherency access as a result of an NDAL OREAD or WRITE
011	unused	
100	BC_CPU	Bcache CPU access (as a result of an NVAX Iread, Dread, or Oread)
101	BC_CPU_IREAD	Bcache CPU access as a result of an NVAX Iread
110	BC_CPU_DREAD	Bcache CPU access as a result of an NVAX Dread or Dread-modify
111	BC_CPU_OREAD	Bcache CPU access as a result of an NVAX Oread due to a read lock, a write, or a write unlock.

The correspondence between CCTL<PM_HIT_TYPE> and **C%PMUX1_H** is shown in Table 13–65.

Table 13–65: Cbox Performance Monitoring Control

CCTL: PM_HIT_TYPE<15:14>	Signal muxed onto C%PMUX1_H	Signal functionality
00	BC_HIT	Bcache hit; factors in VALID and OWNED as necessary, based on the transaction.
01	BC_HIT_OWNED	Bcache hit owned; tag matched, VALID and OWNED were set.
10	BC_HIT_VALID	Bcache hit valid; tag matched, VALID was set, OWNED was either set or clear.
11	BC_MISS_OWNED	Bcache miss; tag did not match, VALID and OWNED were set (triggers writeback).

The HIT signals which produce **C%PMUX1_H** are valid during the same cycle in which the ACCESS signals which produce **C%PMUX0_H** are asserted. They must be valid at the same time because in the central performance monitoring hardware, **C%PMUX1_H** is conditioned with **C%PMUX0_H**.

13.12 Initialization

When the CPU powers up, K_C%RESET_L and K%RESET_CCTL_L are asserted, clearing the main queues and latches in the Cbox and putting the Cbox state machines into their idle states. The only Cbox IPR which is initialized on reset is the Cbox control register, CCTL. It is initialized as described in Section 13.5.1.

K_C%RESET_L is also asserted when the Ebox timeout counter expires. At this time K%RESET_CCTL_L is not asserted. Thus, the Cbox is initialized just as on power-up except that CCTL is not changed.

K_C%RESET_L must be asserted for 18 internal cycles (6 NDAL cycles) in order to properly reset the Cbox.

The backup cache must be initialized and turned on as described in Section 13.9.6. Software must write CCTL to the desired state. The W1C error registers should be cleared so that they are starting with no error bits set.

When the CPU powers up, K%EXT_RESET_L is asserted which puts the pads into their reset state:

- Tristates **P%NDAL_H<63:0>**, **P%CMD_H<3:0>**, **P%ID_H<2:0>**, and **P%PARITY_H<2:0>**. This occurs when internal reset is asserted, and is not qualified with any clock.
- Releases **P%ACK_L**. is occurs when internal reset is asserted, and is not qualified with any clock.
- Deasserts **P%CPU_REQ_L**, **P%CPU_HOLD_L**, and **P%CPU_SUPPRESS_L**. This occurs when K%EXT_RESET_L is asserted, and is not qualified with any clock.
- Deasserts **P%TS_OE_L**, **P%TS_WE_L**, **P%DR_OE_L**, and **P%DR_WE_L**. This occurs when K%EXT_RESET_L is asserted, and is not qualified with any clock.
- Tristates **P%TS_TAG_H<31:17>**,**P%TS_ECC_H<5:0>**,**P%TS_VALID_H**, **P%TS_OWNED_H**, **P%DR_DATA_H<63:0>**, and **P%DR_ECC_H<7:0>**. This occurs when K%EXT_RESET_L is asserted, and is not qualified with any clock.

13.13 Cbox Interfaces

The Cbox interfaces with the Mbox, the NDAL, the backup cache, the Interrupt section, and the Clock section. The signals the Cbox uses for each of these interfaces are listed here.

Table 13–66: CBOX interface signals

Signal	Number	I/O	Description
NDAL SIGNALS (80 total)			
P%CPU_REQ_L	1	O	Requests the NDAL.
P%CPU_HOLD_L	1	O	Holds the NDAL.
P%CPU_GRANT_L	1	I	Grants NVAX the NDAL.
P%CPU_SUPPRESS_L	1	O	Suppresses the NDAL.
P%CPU_WB_ONLY_L	1	I	Suppresses non-writeback NVAX transactions.
P%NDAL_H<63:0>	64	I/O	NDAL address/data, multiplexed lines.
P%CMD_H<3:0>	4	I/O	NDAL command.
P%ID_H<2:0>	3	I/O	Identifies the NDAL driver.
P%PARITY_H<2:0>	3	I/O	Parity on the NDAL.
P%ACK_L	1	I/O	Acknowledges NDAL cycles as correctly received.
BACKUP CACHE TAG STORE SIGNALS (41 total)			
P%TS_INDEX_H<20:5>	16	O	Index into the tag store.
P%TS_OE_L	1	O	Tag Store Output Enable.
P%TS_WE_L	1	O	Tag Store Write Enable.
P%TS_TAG_H<31:17>	15	I/O	Backup cache tag.
P%TS_ECC_H<5:0>	6	I/O	Tag store ECC.
P%TS_OWNED_H	1	I/O	Indicates ownership of the block.
P%TS_VALID_H	1	I/O	Indicates the block is valid.
BACKUP CACHE DATA RAM SIGNALS (92 total)			
P%DR_INDEX_H<20:3>	18	O	Index into the data rams.
P%DR_OE_L	1	O	Data RAM output enable.
P%DR_WE_L	1	O	Data RAM write enable.
P%DR_DATA_H<63:0>	64	I/O	Backup cache data.
P%DR_ECC_H<7:0>	8	I/O	Backup cache data ECC.

Table 13–66 (Cont.): CBOX Interface signals

Signal	Number	I/O	Description
CLOCK PINS (4 total)			
P%PHI12_IN_H	1	I	NDAL clock used in the pads.
P%PHI23_IN_H	1	I	NDAL clock used in the pads.
P%PHI34_IN_H	1	I	NDAL clock used in the pads.
P%PHI41_IN_H	1	I	NDAL clock used in the pads.
CLOCK SECTION INTERFACE (5 total)			
K_MCB%PHI_1_H	1	I	Clock used in the Cbox.
K_MCB%PHI_2_H	1	I	Clock used in the Cbox.
K_MCB%PHI_3_H	1	I	Clock used in the Cbox.
K_MCB%PHI_4_H	1	I	Clock used in the Cbox.
K_PAD%PHI_1_H	1	I	Clock used in the upper pad ring.
K_PAD%PHI_3_H	1	I	Clock used in the upper pad ring.
K_PAD%PHI_4_H	1	I	Clock used in the upper pad ring.
K_PADL%PHI_1_H	1	I	Clock used in the lower pad ring.
K_PADL%PHI_2_H	1	I	Clock used in the lower pad ring.
K_PADL%PHI_3_H	1	I	Clock used in the lower pad ring.
K_PADL%PHI_4_H	1	I	Clock used in the lower pad ring.
K%EXT_RESET_L	1	I	Puts the cache and NDAL pads into their reset state.
K_C%RESET_L	1	I	Resets the Cbox except for CCTL.
K%RESET_CCTL_L	1	I	Resets the Cbox control register, CCTL.
K_CE%RESET_H	1	I	Resets the BIU cycle counter which relates internal to external time.
EBOX INTERFACE SIGNALS (2 total)			
C%CBOX_H_ERR_H	1	O	Indicates a hard error in the backup cache or on the NDAL.
C%CBOX_S_ERR_H	1	O	Indicates a soft error in the backup cache or on the NDAL.
E%TIMEOUT_BASE_H	1	I	Controls the NDAL read timeout counters.
E%TIMEOUT_ENABLE_H	1	I	Controls the NDAL read timeout counters.

Table 13–66 (Cont.): CBOX interface signals

Signal	Number	I/O	Description
TEST AND PERFORMANCE MONITORING SIGNALS (10 total)			
C%PP_DATA_H<11:7>	5	O	Cbox internal state, driven to the Mbox, where it is driven to the parallel port when selected.
C%ISR2_TDO_H	1	O	Cbox internal scan chain output which hooks up to the Mbox scan chain.
M%C_ISR_LOAD_L	1	I	Tells the Cbox LFSR/internal scan chain whether to load or shift.
M%C_ISR_LFSR_L	1	I	Puts the Cbox LFSR/internal scan chain into Linear Feedback Shift Register mode.
T_JTG%DRCLK_L	1	I	Clocks the boundary scan cells.
T_JTG%DRCLK_H	1	I	Clocks the boundary scan cells.
T_JTG%CAPTURE_L	1	I	When asserted, the boundary scan cells are in load mode; otherwise, they are in shift mode.
T_JTG%BSR_EXTEST_L	1	I	When asserted, the pins are driven with data from the boundary scan cells rather than with NVAX internal data.
T_JTG%BSR_UPDATE_L	1	I	Controls the update of the cache I/O pads, when driven by JTAG.
C_PAD_N%BSR_NDAL_H<63>	1	O	Boundary scan chain output from the Cbox pads.
E_PAD_INT%BSR_MACHINE_CHECK_L	1	I	Boundary scan chain input from the Ebox pads.
K_PAD_CK2%DISABLE_OUT_H	1	I	Asynchronously disables all NVAX outputs from driving; equivalent to the inversion of **P%DISABLE_OUT_L**.
C%PMUX0_H	1	O	Cbox performance monitoring output.
C%PMUX1_H	1	O	Cbox performance monitoring output.

Table 13–66 (Cont.): CBOX Interface signals

Signal	Number	I/O	Description
MBOX INTERFACE SIGNALS (157 total)			
M%S6_CMD_H<4:0>	5	I	Mbox reference command field.
M%S6_PA_H<31:3>	29	I	Physical address of Mbox reference.
M%C_S6_PA_H<2:0>	3	I	Physical address of Mbox reference, lower three bits.
M%S6_BYTE_MASK_H<7:0>	8	I	Byte enable field of Mbox reference.
M%CBOX_REF_ENABLE_L	1	I	Indicates that the current S6 reference packet should be latched and processed by the Cbox; not asserted for writes as all writes are processed by the Cbox.
M%CBOX_LATE_EN_H	1	I	This is equivalent to M%CBOX_REF_ENABLE_L, but driven to the Cbox with later timing, after the Mbox detects a Pcache parity error. It indicates that the S6 reference packet should be processed by the Cbox.
M%ABORT_CBOX_IRD_H	1	I	Indicates that any IREAD which the Cbox may be processing should be immediately terminated.
M%CBOX_BYPASS_ENABLE_H	1	I	Indicates that the Cbox may drive B%S6_DATA_H<63:0> during the following cycle in order to attempt a fill data bypass.
B%S6_DATA_H<63:0>	64	I/O	Bus used to receive data from the Mbox and to send data to the Mbox.
C%S6_DP_H<7:0>	8	O	Byte data parity for B%S6_DATA_H<63:0>.
C%CBOX_CMD_H<1:0>	2	O	Command field of Cbox reference sent to Mbox
C%CBOX_ADDR_H<31:5>	27	O	Hexaword address for invalidate sent to Mbox
C%MBOX_FILL_QW_H<4:3>	2	O	Address bits to indicate to which quadword within the hexaword the current fill data belongs.
C%REQ_DQW_H	1	O	Indicates that the requested quadword of data is being returned. This is asserted for both DREADs and IREADs; it is also asserted if a hard error occurs on fill data and the requested quadword has not yet been returned.
C%LAST_FILL_H	1	O	Indicates that this is the last fill sent for the read being processed
C%CBOX_HARD_ERR_H	1	O	Indicates that a hard error is associated with the data being returned. The Mbox treats this as a fill with an error.
C%CBOX_ECC_ERR_H	1	O	Indicates that an ECC error is associated with the data being returned. The Mbox ignores the data and waits for another fill from the Cbox.
C%WR_BUF_BACK_PRES_H	1	O	Indicates that the Cbox cannot accept any more entries in its WRITE_QUEUE.

13.14 Resolved Issues

1. Issue: Does the Cbox need to check for conflicts between writes into the Istream and IREADs? Resolution: Yes, it does. The following case illustrates why.

   Suppose that the Cbox did not check for conflicts between writes and Istream reads. Also note that the SRM requires that an REI be done after any write into the instruction stream. REI flushes all write buffers and flushes the VIC.

   Suppose that the Ibox is prefetching and issues IREAD A, IREAD A+1. A and A+1 are adjacent hexawords. Around the same time the Ebox is doing unaligned WRITE A,A+1,REI which was caused by Istream previous to that now being fetched by the Ibox.

   Suppose the sequence as seen by the Mbox is IREAD A, unaligned WRITE A,A+1, IREAD A+1, REI. The first IREAD is prefetching Istream data and should retrieve the new A. If the first IREAD misses in the VIC and the Pcache, the Bcache will return old data for the IREAD. The write will then be done into the Pcache, since it is write-through, and into the WRITE_QUEUE. At this point the new data for A is in the Pcache.

   Now the second IREAD misses the Pcache and appears in the IREAD_LATCH in the Cbox. It is serviced before the write since no conflict checking is done for IREADs, and they take priority over writes. Old data is returned to the Pcache for the second IREAD. Then the Clear Write Buffer command appears in the Cbox because the Ebox is executing the REI so the write is done.

   At this point the VIC has old data for the IREADs. This is ok because the REI flushes the VIC. Location A is updated in the Pcache because the write was done after the first IREAD. However, the Pcache has old data for A+1 because the Bcache returned the old data after the write missed into the Pcache.

   When the Istream re-fetches A+1, it will get old data from the Pcache. This is not the behavior we want. Thus, the Cbox implements conflict checking for IREADs and prevents the IREAD of A+1 from bypassing the write to A+1.

2. Issue: Is it ok that the Cbox reorders I/O space writes with respect to memory space writes? Resolution: Yes, it is OK per VAX ECO 95, Allow Write-and-Run to I/O space.

   This is the scenario where the Cbox may reorder I/O space writes with respect to memory writes: The Mbox issues Memory Write A followed by I/O Write B. Memory Write A hits owned in the backup cache and is written. I/O Write B goes to the NON_WRITEBACK_QUEUE.

   The NDAL is busy or **P%CPU_WB_ONLY_L** is asserted, so I/O Write B stays in the NON_WRITEBACK_QUEUE. Meantime, a cache coherency request arrives for memory location A. The data is retrieved from the backup cache and put into the WRITEBACK_QUEUE.

   Since
   the WRITEBACK_QUEUE contains a cache coherency request (or **P%CPU_WB_ONLY_L** is asserted), the WRITEBACK_QUEUE has priority over the NON_WRITEBACK_QUEUE. Therefore Memory Data A reaches the NDAL before I/O Write B, effectively reordering the writes.

13.15 NVAX CBOX Signal Name Cross-Reference

All CBOX signal names and pin names referenced in this chapter have appeared in bold and reflect the actual name appearing in the NVAX schematic set, with the exception of **K%EXT_RESET_L**, which is a behavioral model name only. For each signal and pin appearing in this chapter, the table below lists the corresponding name which exists in the behavioral model.

Table 13–67: Cross-reference of all names appearing in the CBOX chapter

Schematic Name	Behavioral Model Name
B%S6_DATA_H<63:0>	B%S6_DATA_H<63:0>
C%CBOX_ADDR_H<31:5>	C%CBOX_ADDR_H<31:5>
C%CBOX_CMD_H<1:0>	C%CBOX_CMD_H<1:0>
C%CBOX_ECC_ERR_H	C%CBOX_ECC_ERR_H
C%CBOX_HARD_ERR_H	C%CBOX_HARD_ERR_H
C%CBOX_H_ERR_H	C%CBOX_H_ERR_H
C%CBOX_S_ERR_H	C%CBOX_S_ERR_H
C%ISR2_TDO_H	C%ISR2_TDO_H
C%LAST_FILL_H	C%LAST_FILL_H
C%MBOX_FILL_QW_H<4:3>	C%MBOX_FILL_QW_H<4:3>
C%PMUX0_H	C%PMUX0_H
C%PMUX1_H	C%PMUX1_H
C%PP_DATA_H<11:7>	C%PP_DATA_H<11:7>
C%REQ_DQW_H	C%REQ_DQW_H
C%S6_DP_H<7:0>	C%S6_DP_H<7:0>
C%WR_BUF_BACK_PRES_H	C%WR_BUF_BACK_PRES_H
C_ADC%ABUS_H<31:0>	C_BUS%ABUS_H<31:0>
C_ADC%BIU_ADDR_OUT_H<31:0>	C_ADC%BIU_ADDR_OUT_H<31:0>
C_BIU%ADC_ADDR_IN_H<31:0>	C_BIU%ADC_ADDR_IN_H<31:0>
C_BIU%CYCLE_1_H	C_BIU%CYCLE_1_H
C_BIU%CYCLE_2_H	C_BIU%CYCLE_2_H
C_BIU%CYCLE_3_H	C_BIU%CYCLE_3_H
C_BIU_NOC%BXI_TIMO_0_LAT_H	C_BIU_NOC%BXI_TIMO_0_LAT_H
C_BIU_NOC%BXI_TIMO_1_LAT_H	C_BIU_NOC%BXI_TIMO_1_LAT_H
C_BIU_NOC_5%BXI_TIMO_0_EN_H	C_BIU_NOC%BXI_TIMO_0_EN_H
C_BIU_NOC_5%BXI_TIMO_1_EN_H	C_BIU_NOC%BXI_TIMO_1_EN_H
C_BUS%BIU_DATA_H<63:0>	C_BUS%BIU_DATA_H<63:0>
C_BUS%DBUS_H<63:0>	C_BUS%DBUS_H<63:0>
C_PAD_N%BSR_NDAL_H<63>	T_BSR%NDAL_HI_H<63>
E%TIMEOUT_BASE_H	E%TIMEOUT_BASE_H

Table 13–67 (Cont.): Cross-reference of all names appearing in the CBOX chapter

Schematic Name	Behavioral Model Name
E%TIMEOUT_ENABLE_H	E%TIMEOUT_ENABLE_H
E_PAD_INT%BSR_MACHINE_CHECK_L	T_BSR%MACHINE_CHECK_H
K%EXT_TMBS_H	K_PAD%EXT_TMBS_H
K%RESET_CCTL_L	K%RESET_L
K_C%RESET_L	K_C%RESET_L
K_CE%RESET_H	K_CE%RESET_H
K_MCB%PHI_1_H	K%PHI_1_H
K_MCB%PHI_2_H	K%PHI_2_H
K_MCB%PHI_3_H	K%PHI_3_H
K_MCB%PHI_4_H	K%PHI_4_H
K_PAD%PHI_1_H	K%PHI_1_H
K_PAD%PHI_3_H	K%PHI_3_H
K_PAD%PHI_4_H	K%PHI_4_H
K_PADL%PHI_1_H	K%PHI_1_H
K_PADL%PHI_2_H	K%PHI_2_H
K_PADL%PHI_3_H	K%PHI_3_H
K_PADL%PHI_4_H	K%PHI_4_H
K_PAD_CK2%DISABLE_OUT_H	P%DISABLE_OUT_L
K_PAD%EXT_RESET_TOP_L	K%EXT_RESET_L
K_PAD%EXT_RESET_BOT_L	K%EXT_RESET_L
M%ABORT_CBOX_IRD_H	M%ABORT_CBOX_IRD_H
M%CBOX_BYPASS_ENABLE_H	M%CBOX_BYPASS_ENABLE_H
M%CBOX_LATE_EN_H	M%CBOX_LATE_EN_H
M%CBOX_REF_ENABLE_L	M%CBOX_REF_ENABLE_H
M%C_ISR_LFSR_L	T%ISR_LFSR_H
M%C_ISR_LOAD_L	T%ISR_LOAD_H
M%C_S6_PA_H<2:0>	M%C_S6_PA_H<2:0>
M%S6_BYTE_MASK_H<7:0>	M%S6_BYTE_MASK_H<7:0>
M%S6_CMD_H<4:0>	M%S6_CMD_H<4:0>
M%S6_PA_H<31:3>	M%S6_PA_H<31:3>
T%MBOX_DR_PP_H	T%MBOX_DR_PP_H
T_JTG%BSR_EXTEST_L	T%BSR_EXTEST_H
T_JTG%BSR_UPDATE_L	T%BSR_UPDATE_H
T_JTG%CAPTURE_L	T%CAPTURE_H
T_JTG%DRCLK_H	T_JTG_TAP%DR_CLKEN_H
T_JTG%DRCLK_L	T_JTG_TAP%DR_CLKEN_H

Table 13–67 (Cont.): Cross-reference of all names appearing in the CBOX chapter

Schematic Name	Behavioral Model Name
P%ACK_L	P%ACK_L
P%CMD_H<3:0>	P%CMD_H<3:0>
P%CPU_GRANT_L	P%CPU_GRANT_L
P%CPU_HOLD_L	P%CPU_HOLD_L
P%CPU_REQ_L	P%CPU_REQ_L
P%CPU_SUPPRESS_L	P%CPU_SUPPRESS_L
P%CPU_WB_ONLY_L	P%CPU_WB_ONLY_L
P%DISABLE_OUT_L	P%DISABLE_OUT_L
P%DR_DATA_H<63:0>	P%DR_DATA_H<63:0>
P%DR_ECC_H<7:0>	P%DR_ECC_H<7:0>
P%DR_INDEX_H<20:3>	P%DR_INDEX_H<20:3>
P%DR_OE_L	P%DR_OE_L
P%DR_WE_L	P%DR_WE_L
P%ID_H<2:0>	P%ID_H<2:0>
P%NDAL_H<63:0>	P%NDAL_H<63:0>
P%OSC_TC1_H	P%OSC_TC1_H
P%PARITY_H<2:0>	P%PARITY_H<2:0>
P%PHI12_IN_H	P%PHI12_IN_H
P%PHI23_IN_H	P%PHI23_IN_H
P%PHI34_IN_H	P%PHI34_IN_H
P%PHI41_IN_H	P%PHI41_IN_H
P%PHI12_OUT_H	P%PHI12_OUT_H
P%TS_ECC_H<5:0>	P%TS_ECC_H<5:0>
P%TS_INDEX_H<20:5>	P%TS_INDEX_H<20:5>
P%TS_OE_L	P%TS_OE_L
P%TS_OWNED_H	P%TS_OWNED_H
P%TS_TAG_H<31:17>	P%TS_TAG_H<31:17>
P%TS_VALID_H	P%TS_VALID_H
P%TS_WE_L	P%TS_WE_L

13.16 Revision History

Table 13–68: Revision History

Who	When	Description of change
Rebecca Stamm	9-Oct-1991	Made the following change: Bcache data MUST be initialized with correct ECC on powerup, contrary to what was in previous revisions.
Rebecca Stamm	16-Aug-1991	Minor updates and clarifications. RDE and UNEXPECTED_FILL are both set if an unexpected RDE arrives on the NDAL. During ETM, a read modify that does not hit owned causes a read to memory, NOT an OREAD to memory. On uncorr error on RMW, checkbits 3 and 7 are inverted rather than 3,6,7. Added description of why the bits are inverted.
Rebecca Stamm	20-Feb-1991	Correct TS_CMD and DR_CMD encodings. Clarify some sections. Add description of NVAX-NDAL timing. Add statements that the contents of the Cbox error registers are not changed during reset. Added cache timing information. Added table of cache behavior while it is ON. Appended P% to the beginning of all pin names, since those match the schematics and the beh model. Add assertion levels to signal names.
Rebecca Stamm	14-Aug-1990	Remove E%MEMORY_RESET, add K%RESET_CCTL_L.
Rebecca Stamm	4-Jul-1990	Correct description of E%MEMORY_RESET. Added K_CE%RESET_H. Added CCTL<FORCE_NDAL_PERR>. Update description of Cbox behavior when **P%CPU_WB_ONLY_L** is asserted. Update conditions for servicing the write queue. Update cache coherency section with bug correction. Added to cache ram speed table, 16ns. Clarify CEFSTS<COUNT>. Clarify BCFLUSH during FORCE_HIT mode. Update handling of DREAD lock which fails on an uncorrectable error on the first quadword. Clarify handling of correctable error in the tag store. Added section about the FILL_CAM and block conflicts.
Rebecca Stamm	3-Jun-1990	Clarify handling of write, readlock in etm. Make CEFSTS<UNEXPECTED_FILL> W1C.
Rebecca Stamm	17-May-1990	Clarify invalidate handling sections. Always give the WRITEBACK_QUEUE priority over the NON_WRITEBACK_QUEUE. Change bit definitions in CEFSTS. Change WR_MRG_DONE to REQ_FILL_DONE in CEFSTS and FILL_CAM. Clarify stalling of IPR accesses to the tag store while a FILL_CAM entry to the same block is valid.

Table 13–68 (Cont.): Revision History

Who	When	Description of change
Rebecca Stamm	20-Feb-1990	Update error table. Add complete description of timeout counters. Change CCTL<TIMEOUT_EXT> to CCTL<TIMEOUT_TEST>, update description of that bit. Add E%TIMEOUT_BASE_H to Cbox interface signal list. Add control signal names for scan chain, updated scan chain section, removed two bits from the scan chain. Add control signal names for parallel port, updated parallel port section. Update description of CEFSTS RDLK bit. Clarified description of CEFADR. Clarified tag store actions on deallocates. Update performance monitoring hardware section and added control bits to CCTL. Correct clock names. Bcache read quadwords returned in wrapped order rather than in Grey code order. WRITEBACK_QUEUE full prevents all transactions from starting. Add BC_TS_CMD decodings for the parallel port. Added TS_CMD encodings to BCETSTS. Added DR_CMD encodings to BCEDSTS. More detail on NESTS bit descriptions. Better explanation of use of BCDECC register. Add detail to WRITE_UNLOCK explanation.
Rebecca Stamm	3-Feb-1990	External release. Eliminated BCEDHI and BCEDLO IPRs. Made updates based on internal review.
Rebecca Stamm	19-Jan-1990	Release for internal review.
Rebecca Stamm	13-Jan-1990	Intermediate release. Many edits. Eliminated backup cache data RAM access through IPR reads and writes. Updated Cbox internal bussing diagrams and description. Write queue is 8 entries.
Rebecca Stamm	21-Mar-1989	Release for external review
Rebecca Stamm	16-Mar-1989	Release for internal review

Chapter 14

Vector Interface

14.1 Description

The NVAX CPU chip does not fully support the VAX vector instruction set and any attempt to execute a vector instruction will result in a reserved instruction fault. Vector instructions are listed in Table 14–1.

Table 14–1: Vector Instruction Set

Opcode	Instruction
31FD	MFVP regnum.rw, dst.wl
34FD	VLDL cntrl.rw, base.ab, stride.rl
35FD	VGATHL cntrl.rw, base.ab
36FD	VLDQ cntrl.rw, base.ab, stride.rl
37FD	VGATHQ cntrl.rw, base.ab
80FD	VVADDL cntrl.rw
81FD	VSADDL cntrl.rw, scal.rl
82FD	VVADDG cntrl.rw
83FD	VSADDG cntrl.rw, scal.rq
84FD	VVADDF cntrl.rw
85FD	VSADDF cntrl.rw, scal.rl
86FD	VVADDD cntrl.rw
87FD	VSADDD cntrl.rw, scal.rq
88FD	VVSUBL cntrl.rw
89FD	VSSUBL cntrl.rw, scal.rl
8AFD	VVSUBG cntrl.rw
8BFD	VSSUBG cntrl.rw, scal.rq
8CFD	VVSUBF cntrl.rw
8DFD	VSSUBF cntrl.rw, scal.rl

Table 14–1 (Cont.): Vector Instruction Set

Opcode	Instruction
8EFD	VVSUBD cntrl.rw
8FFD	VSSUBD cntrl.rw, scal.rq
9CFD	VSTL cntrl.rw, base.ab, stride.rl
9DFD	VSCATL cntrl.rw, base.ab
9EFD	VSTQ cntrl.rw, base.ab, stride.rl
9FFD	VSCATQ cntrl.rw, base.ab
A0FD	VVMULL cntrl.rw
A1FD	VSMULL cntrl.rw, scal.rl
A2FD	VVMULG cntrl.rw
A3FD	VSMULG cntrl.rw, scal.rq
A4FD	VVMULF cntrl.rw
A5FD	VSMULF cntrl.rw, scal.rl
A6FD	VVMULD cntrl.rw
A7FD	VSMULD cntrl.rw, scal.rq
A8FD	VSYNC regnum.rw
A9FD	MTVP regnum.rw, src.rl
AAFD	VVDIVG cntrl.rw
ABFD	VSDIVG cntrl.rw, scal.rq
ACFD	VVDIVF cntrl.rw
ADFD	VSDIVF cntrl.rw, scal.rl
AEFD	VVDIVD cntrl.rw
AFFD	VSDIVD cntrl.rw, scal.rq
C0FD	VVCMPL cntrl.rw
C1FD	VSCMPL cntrl.rw, scal.rl
C2FD	VVCMPG cntrl.rw
C3FD	VSCMPG cntrl.rw, scal.rq
C4FD	VVCMPF cntrl.rw
C5FD	VSCMPF cntrl.rw, scal.rl
C6FD	VVCMPD cntrl.rw
C7FD	VSCMPD cntrl.rw, scal.rq
C8FD	VVBISL cntrl.rw
C9FD	VSBISL cntrl.rw, scal.rl
CCFD	VVBICL cntrl.rw

Table 14–1 (Cont.): Vector Instruction Set

Opcode	Instruction
CDFD	VSBICL cntrl.rw, scal.rl
E0FD	VVSRLL cntrl.rw
E1FD	VSSRLL cntrl.rw, scal.rl
E4FD	VVSLLL cntrl.rw
E5FD	VSSLLL cntrl.rw, scal.rl
E8FD	VVXORL cntrl.rw
E9FD	VSXORL cntrl.rw, scal.rl
ECFD	VVCVT cntrl.rw
EDFD	IOTA cntrl.rw, scal.rl
EEFD	VVMERGE cntrl.rw
EFFD	VSMERGE cntrl.rw, scal.rq

Although the vector instruction set is not fully implemented, some residual support is included in the NVAX CPU chip and should be considered:

- The Ibox, under control of the IROM, decodes the vector instructions listed above, including parsing and processing the instruction specifiers. If a memory management exception is detected on the instruction or one of the specifiers, the Ibox will report it to the Ebox, which will ignore it in favor of reporting a reserved instruction fault instead. However, if a hardware error is detected during the processing of the vector instruction or specifiers, that error will be reported in the usual way.
- The ECR<VECTOR_PRESENT> bit remains in the hardware, but a reserved instruction fault will result if a vector instruction is executed, independent of the state of this bit.
- A vector disabled fault will never be generated by the NVAX CPU chip microcode.
- References to vector processor registers in the range 90–97 (hex) are intercepted by the microcode and are not transmitted on the NDAL as is the normal case for an unimplemented processor register. Rather, writes to these registers are ignored, and reads from these registers return 0. The operating system depends on this behavior.

14.2 Revision History

Table 14–2: Revision History

Who	When	Description of change
Mike Uhler	06-Jan-1990	Initial release
Mike Uhler	02-Feb-1991	Update after pass 1 PG.

Chapter 15

Error Handling

This chapter describes the NVAX CPU error exceptions and interrupts as seen from the macrocoder's point of view. It is organized with respect to the SCB vectors through which the event is dispatched. The SCB layout and SCB vector format are described in Chapter 2. Exceptions and interrupts that are a result of normal system operation are described in Chapter 2.

15.1 Terminology

Term	Meaning
Fill	Any quadword of data returned to the NVAX CPU chip in response to read-type operation. The quadword containing the requested data is a fill.
Ownership bit	In the Bcache and the memory, a bit is stored with each hexaword called the ownership bit. In the Bcache it indicates the Bcache owns the block; it has the one valid copy of the data. In memory it indicates some cache or bus interface has the one good copy of the block, not the memory.
Memory cache state	In memory in various system environments, a certain amount of state is kept for each hexaword in memory. This state always includes the ownership bit. In some system environments, it includes additional information.
ETM	Error transition mode in the Bcache: in this mode the Bcache is not used except if it owns the addressed block. It continues to respond to NDAL coherency requests which require writeback.

15.2 Error handling Introduction and Summary

This chapter discusses all levels of hardware and microcode-detected errors. Errors notification occurs through one of the following events, listed in order of decreasing severity.

- Console error halt—A halt to console mode is caused by one of several errors such as Interrupt Stack Not Valid. For certain halt conditions, the console prompts for a command and waits for operator input. For other halt conditions, the console may attempt a system restart or a system bootstrap as defined by DEC Standard 032. The actual algorithms used are outside of the scope of this document.
- Machine check—A hardware error occurred synchronously with respect to the execution of instructions. Instruction-level recovery and retry may be possible.
- Power fail—The power supply asserted the power fail signal.

- Hard error interrupt—A hardware error occurred asynchronously with respect to the execution of instructions. Usually, data is lost or state is corrupted, and instruction-level recovery may not be possible.
- Soft error interrupt—A hardware error occurred asynchronously with respect to the execution of instructions. The error is not fatal to the execution of instructions, and instruction-level recovery is usually possible.
- Kernel stack not valid—During exception processing, a memory management exception occurred while trying to push information on the kernel stack.

This chapter explains in detail several of the SCB entry points. The purpose is to help the operating system programmer determine exactly what error occurred and to recommend an error recovery method. Since this chapter is only concerned with errors which are generic to all system environments, it may be used as the basis for a specification of error handling and recovery for particular systems based on the NVAX CPU chip.

The following information is given in this chapter for each SCB entry point:

- What parameters are pushed on the stack.
- What failure codes are defined.
- What additional information exists and should be collected for analysis.
- How to determine what error(s) actually occurred.
- How to restore the state of the machine, and what level of recovery is possible.

Table 15–1 shows the general error categories associated with each of these error notifications.

Table 15–1: Error Summary By Notification Entry Point

Entry Point	SCB Index (hex)	General Error Categories
Console Halt	N/A	Interrupt Stack not valid, kernel-mode halt, double error, illegal SCB vector, initial Power up, **HALT_L** assertion
Machine Check	04	Memory management, interrupt, microcode detected CPU errors, CPU stall timeout, TB parity errors, VIC tag or data parity errors, Bcache uncorrectable data read errors, memory/NDAL read errors (no-ACK, timeout, or RDE from system environment)
Power Fail	0C	system environment notification via **PWRFL_L**
Soft Error Interrupt	54	VIC tag or data parity errors, Pcache tag or data parity errors, Bcache uncorrectable tag errors, Bcache uncorrectable data read errors Bcache uncorrectable data errors in writebacks, Bcache correctable tag and data errors, memory/NDAL read errors (no-ACK, timeout, or RDE on reads), NDAL parity errors, system environment notification via **S_ERR_L**

Table 15–1 (Cont.): Error Summary By Notification Entry Point

Entry Point	SCB Index (hex)	General Error Categories
Hard Error Interrupt	60	Bcache uncorrectable data errors on write operations, NDAL no-ACK on writes, Bcache fill errors in NDAL ownership reads after merging write data in the cache data RAMs, system environment notification via **H_ERR_L**

15.3 Error Handling and Recovery

All errors (except those resulting in console halt) go through SCB vector entry points and are handled by service routines provided by the operating system. A console halt transfers control to a hardware-prescribed IO-space address. Software driven recovery or retry is not recommended for errors resulting in console halt.

Software error handling (by operating system routines) can be logically divided into the following steps:

- State collection.
- Analysis.
- Recovery.
- Retry.

These steps are discussed in general in the next four sections. After that, details are supplied on analysis, recovery and retry for each error event which results in an exception or interrupt. This information is organized by SCB entry point.

15.3.1 Error State Collection

Before error analysis can begin, all relevant state must be collected. The stack frame provides the PC/PSL pair for all exceptions and interrupts. For machine checks, the stack frame also provides details about the error.

In addition to the stack frame, machine checks and hard and soft error interrupts usually require analysis of other registers. It is strongly recommended that all the state listed below be read and saved in these cases. State is saved prior to analysis so that analysis is not complicated by changes in state in the registers as the analysis progresses, and so that errors incurred during analysis and recovery can be processed with that context.

Ibox
ICSR: Ibox (VIC) control and status register.
VMAR: VIC memory address register.

Ebox
ECR: Ebox control and status register.

Mbox
TBSTS: TB status register.
TBADR: TB address register.
PCSTS: Pcache status register.
PCADR: Pcache address register.

Cbox
CCTL: Cbox Control Register.
BCEDSTS: Bcache data error status register.
BCEDIDX: Bcache data error index register.
BCEDECC: Bcache data error ECC/syndrome register.
BCETSTS: Bcache tag error status register.
BCETIDX: Bcache tag error index/address register.
BCETAG: Bcache errored tag register.
CEFSTS: Read and Bcache fill status register.
CEFADR: Read and Bcache fill address register.
NESTS: NDAL error status register.
NEOADR: NDAL error output address register.
NEOCMD: NDAL error output command register.
NEICMD: NDAL error input command register.
NEDATHI: NDAL error input data register (HI).
NEDATLO: NDAL error input data register (LO).

System environment
All states (i.e., CSRs) which report error conditions or events.

For the purposes of the rest of this chapter, it is assumed that each of these states is saved in a variable whose name is constructed by prepending "S_" to the register name. For example, the ICSR would be saved in the variable S_ICSR.

The following example shows allocation of memory storage for the error state.

```
; ERROR STATE COLLECTION DATA STORAGE

                                          ;IBOX
S_ICSR:         .LONG   0                 ; IBOX VIC CONTROL AND STATUS REGISTER
S_VMAR:         .LONG   0                 ; IBOX VIC ERROR ADDRESS REGISTER

                                          ;EBOX
S_ECR:          .LONG   0                 ; EBOX CONTROL AND STATUS REGISTER

                                          ;MBOX
S_TBSTS:        .LONG   0                 ; TB STATUS REGISTER
S_TBADR:        .LONG   0                 ; TB ERROR ADDRESS REGISTER
S_PCSTS:        .LONG   0                 ; PCACHE STATUS REGISTER
S_PCADR:        .LONG   0                 ; PCACHE ERROR ADDRESS REGISTER
```

```
                                ;CBOX
S_CCTL:         .LONG   0       ; CBOX CONTROL REGISTER
S_BCEDSTS:      .LONG   0       ; BCACHE DATA RAM ERROR STATUS REGISTER
S_BCEDIDX:      .LONG   0       ; BCACHE DATA RAM ERROR INDEX REGISTER
S_BCEDECC:      .LONG   0       ; BCACHE DATA RAM ECC/SYNDROME REGISTER
S_BCETSTS:      .LONG   0       ; BCACHE TAG RAM ERROR STATUS REGISTER
S_BCETIDX:      .LONG   0       ; BCACHE TAG RAM ERROR INDEX REGISTER
S_BCETAG:       .LONG   0       ; BCACHE TAG RAM ERRORED TAG REGISTER
S_CEFSTS:       .LONG   0       ; READ AND BCACHE FILL ERROR STATUS REGISTER
S_CEFADR:       .LONG   0       ; READ AND BCACHE FILL ERROR ADDRESS REGISTER
S_NESTS:        .LONG   0       ; NDAL ERROR STATUS REGISTER
S_NEOADR:       .LONG   0       ; NDAL OUTPUT ERROR ADDRESS REGISTER
S_NEOCMD:       .LONG   0       ; NDAL OUTPUT ERROR COMMAND REGISTER
S_NEICMD:       .LONG   0       ; NDAL INPUT ERROR COMMAND REGISTER
S_NEDATHI:      .LONG   0       ; NDAL INPUT ERROR ADDRESS REGISTER (HI)
S_NEDATLO:      .LONG   0       ; NDAL INPUT ERROR ADDRESS REGISTER (LO)

; SYSTEM ENVIRONMENT:
; ERROR REGISTERS FROM THE SYSTEM ENVIRONMENT (MODULE, MEMORY(S), BUS INTERFACE(S))
; ARE SAVED HERE
```

The following example shows collection of error state which would normally be done early in the error handling routine. Note the handling of error registers which may be overwritten in the event of a more severe error. For example, after a correctable Bcache data RAM error, BCEDIDX would hold the index of the correctable error. If an uncorrectable Bcache data RAM error occurs, BCEDIDX would be reloaded with the index of the more sever uncorrectable error. To ensure the data in BCEDIDX and BCEDECC matches the report in BCEDSTS, a conditional test is performed and these two registers are recaptured if both an uncorrectable and correctable error are reported in BCEDSTS. Otherwise, BCEDIDX and BCEDECC could reflect a previous correctable error even though BCEDSTS reports a more severe error.

```
                                                ;SAVE ALL ERROR STATE UPON ENTRY TO ERROR HANDLING ROUTINE
SAVE_STATE:
                                                ;IBOX
                MFPR    #PR19$_ICSR,S_ICSR
                MFPR    #PR19$_VMAR,S_VMAR

                                                ;EBOX
                MFPR    #PR19$_ECR,S_ECR

                                                ;MBOX
                MFPR    #PR19$_TBSTS,S_TBSTS
                MFPR    #PR19$_TBADR,S_TBADR
                MFPR    #PR19$_PCSTS,S_PCSTS
                MFPR    #PR19$_PCADR,S_PCADR

                                                ;CBOX
                MFPR    #PR19$_CCTL,S_CCTL
                MFPR    #PR19$_BCEDIDX,S_BCEDIDX
                MFPR    #PR19$_BCEDECC,S_BCEDECC
                MFPR    #PR19$_BCEDSTS,S_BCEDSTS
                BICL3   #^C<BCEDSTS$M_CORR ! BCEDSTS$M_LOCK>,S_BCEDSTS,R0
                CMPL    R0,#BCEDSTS$M_CORR ! BCEDSTS$M_LOCK
                BNEQ    10$
                MFPR    #PR19$_BCEDIDX,S_BCEDIDX
                MFPR    #PR19$_BCEDECC,S_BCEDECC

10$:            MFPR    #PR19$_BCETIDX,S_BCETIDX
                MFPR    #PR19$_BCETAG,S_BCETAG
                MFPR    #PR19$_BCETSTS,S_BCETSTS
                BICL3   #^C<BCETSTS$M_CORR ! BCETSTS$M_LOCK>,S_BCETSTS,R0
                CMPL    R0,#BCETSTS$M_CORR ! BCETSTS$M_LOCK
                BNEQ    20$
                MFPR    #PR19$_BCETIDX,S_BCETIDX
                MFPR    #PR19$_BCETAG,S_BCETAG
```

```
20$:            MFPR    #PR19$_CEFSTS,S_CEFSTS
                MFPR    #PR19$_CEFADR,S_CEFADR
                MFPR    #PR19$_NESTS,S_NESTS
                MFPR    #PR19$_NEOADR,S_NEOADR
                MFPR    #PR19$_NEOCMD,S_NEOCMD
                MFPR    #PR19$_NEICMD,S_NEICMD
                MFPR    #PR19$_NEDATHI,S_NEDATHI
                MFPR    #PR19$_NEDATLO,S_NEDATLO

                                                  ;SYSTEM ENVIRONMENT
; COLLECTION OF SYSTEM ENVIRONMENT ERROR REGISTERS GOES HERE
```

Additional state collection is recommended while/after flushing the Bcache because certain errors may occur as a result of the flush operation. The following state should be collected immediately after flushing each Bcache location.

Cbox

CCTL: Cbox Control Register.

BCEDSTS: Bcache data error status register.

BCEDIDX: Bcache data error index register.

BCEDECC: Bcache data error ECC/syndrome register.

BCETSTS: Bcache tag error status register.

BCETIDX: Bcache tag error index/address register.

BCETAG: Bcache errored tag register.

NESTS: NDAL error status register.

NEOADR: NDAL error output address register.

NEOCMD: NDAL error output command register.

System environment

All states (i.e., CSRs) which report the event of NVAX sending a BADWDATA cycle on the NDAL.

For the purposes of the rest of this chapter, it is assumed that each of these states is saved in a variable whose name is constructed by prepending "SS_" to the register name. For example, the BCEDSTS register would be saved in the variable SS_BCEDSTS.

The following example shows allocation of memory storage for additional error state collected while/after flushing the Bcache.

```
; ADDITIONAL ERROR STATE COLLECTION DATA STORAGE FOR AFTER BCACHE FLUSH

                                        ;CBOX
SS_CCTL:        .LONG   0       ; CBOX CONTROL REGISTER
SS_BCEDSTS:     .LONG   0       ; BCACHE DATA RAM ERROR STATUS REGISTER
SS_BCEDIDX:     .LONG   0       ; BCACHE DATA RAM ERROR INDEX REGISTER
SS_BCEDECC:     .LONG   0       ; BCACHE DATA RAM ECC/SYNDROME REGISTER
SS_BCETSTS:     .LONG   0       ; BCACHE TAG RAM ERROR STATUS REGISTER
SS_BCETIDX:     .LONG   0       ; BCACHE TAG RAM ERROR INDEX REGISTER
SS_BCETAG:      .LONG   0       ; BCACHE TAG RAM ERRORED TAG REGISTER
SS_NESTS:       .LONG   0       ; NDAL ERROR STATUS REGISTER
SS_NEOADR:      .LONG   0       ; NDAL OUTPUT ERROR ADDRESS REGISTER
SS_NEOCMD:      .LONG   0       ; NDAL OUTPUT ERROR COMMAND REGISTER

; SYSTEM ENVIRONMENT:
; ADDITIONAL ERROR STATE COLLECTION DATA STORAGE FOR AFTER BCACHE FLUSH
;
; REGISTERS WHICH ARE AFFECTED BY A BADWDATA CYCLE FROM NVAX ARE SAVED HERE
; AFTER THE BCACHE FLUSH
```

The following example shows collection of error state which would normally be collected during and just after flushing the Bcache.

```
AFTER_BCFLUSH:
                                                       ;CBOX
                MFPR   #PR19$_CCTL,SS_CCTL
                MFPR   #PR19$_BCEDIDX,SS_BCEDIDX
                MFPR   #PR19$_BCEDECC,SS_BCEDECC
                MFPR   #PR19$_BCEDSTS,SS_BCEDSTS
                BICL3  #^C<BCEDSTS$M_CORR ! BCEDSTS$M_LOCK>,SS_BCEDSTS,R0
                CMPL   R0,#BCEDSTS$M_CORR ! BCEDSTS$M_LOCK
                BNEQ   30$
                MFPR   #PR19$_BCEDIDX,SS_BCEDIDX
                MFPR   #PR19$_BCEDECC,SS_BCEDECC
;
30$:            MFPR   #PR19$_BCETIDX,SS_BCETIDX
                MFPR   #PR19$_BCETAG,SS_BCETAG
                MFPR   #PR19$_BCETSTS,SS_BCETSTS
                BICL3  #^C<BCETSTS$M_CORR ! BCETSTS$M_LOCK>,SS_BCETSTS,R0
                CMPL   R0,#BCETSTS$M_CORR ! BCETSTS$M_LOCK
                BNEQ   40$
                MFPR   #PR19$_BCETIDX,SS_BCETIDX
                MFPR   #PR19$_BCETAG,SS_BCETAG
;
40$:            MFPR   #PR19$_NESTS,SS_NESTS
                MFPR   #PR19$_NEOADR,SS_NEOADR
                MFPR   #PR19$_NEOCMD,SS_NEOCMD

                                                       ;SYSTEM ENVIRONMENT
; COLLECTION OF SYSTEM ENVIRONMENT ERROR REGISTERS AFFECTED BY A BADWDATA CYCLE FROM NVAX GOES HERE
```

15.3.2 Error Analysis

With the error state obtained during the collection process, the error condition can be analyzed. The purpose is to determine what error event caused the particular notification being handled (to the extent possible), and what other errors may also have occurred. Analysis of machine checks and hard and soft error interrupts should be guided by the parse trees given in the appropriate sections below.

NOTE

> Errors detected in or by one of the caches usually result in the cache automatically being disabled. However, to minimize the possibility of nested errors, it is suggested that error analysis and recovery for memory or cache-related errors be performed with the Pcache disabled and the Bcache in ETM.

In some cases, a notification for a single error occurs in two ways. For example, an uncorrectable error in the Bcache data RAMs will cause a soft error interrupt and may also cause a machine check. Software should handle cases where a machine check handler clears error bits and then the soft error handler is entered with no error bits set.

In certain cases one error event results in two related reports. For example, a Bcache uncorrectable data error during a writeback will be reported in NESTS as a BADWDATA event. In this case, the BADWDATA event captures the full address of the errored data (that is why BADWDATA is an error event). Cases like this are handled as single error events.

In general an error reporting register can report events which lead to machine check, soft error, or hard error. A given error event can result in machine check and soft error interrupt, or in just one or the other. Events which lead to hard error interrupts generally can not also cause machine check or soft error interrupt. Sometimes an error event which leads to machine check or soft error interrupt is closely related to an event which leads to hard error interrupt (e.g., Bcache fill error on first quadword of a fill for an OREAD done for a write causes soft error interrupt, but the same error on a later quadword causes hard error interrupt).

Multiple simultaneous errors may make useful recovery impossible. However, in cases where no conflict exists in the reporting of the multiple errors (i.e., no one error register is used to report two errors), and recovery from each error is possible, then recovery from the set of errors is accomplished by recovering from all of them. For example, recovery from a Pcache tag parity error and a Bcache correctable data error being reported together is possible by following the recovery procedures for each error in sequence.

The error cause determination parse tree for machine check exception is directed at causes or possible causes of machine checks. It ignores errors which lead to hard or soft error interrupts but not to machine checks. Similarly, the hard error interrupt cause determination ignores errors which lead to machine check or soft error interrupt, and the soft error interrupt cause determination ignores errors which lead to machine check or hard error interrupt.

There is a natural order between machine check, hard error interrupt, and soft error interrupt because the IPL for hard error interrupts is higher than that of soft error interrupts and the IPL in the machine check exception is higher than either of the error interrupts. This hierarchy is important because knowledge of which notification event occurred is used to discriminate between certain error events (e.g., an error on the initial fill quadword for a read-lock is distinguished from a fill error on a subsequent quadword by the fact of machine check notification).

15.3.3 Error Recovery

Recovery from errors consists of clearing any latched error state, repairing damaged state (if necessary and possible), and restoring the system to normal operation. There are special considerations involved in analysis and recovery from cache or memory errors, which are covered in the next sections.

Recovery from multiple error scenarios is possible when there is no conflict in the error registers which report the errors and there is no conflict in the recovery procedures for the errors. However all recovery procedures in this chapter assume that only one error is present. None of the procedures are valid in multiple error scenarios without further analysis.

In some instances, it may be desirable to stop using the hardware which is the source of a large number of errors. For example, if a cache reports a large number of errors, it may be better to disable it. It is suggested that software maintain error counts which should be compared against error thresholds on every error report. If the count (per unit time) exceeds the threshold, the hardware should be disabled.

NOTE

> Hard failures of one bit in the tag store can lead to unrecoverable errors requiring a full system crash. It would be appropriate to have an extremely low threshold for tag store correctable errors, especially if they recur in the same location or bit position.

NOTE

> NVAX CPU utilization of the NDAL and memory is extremely high if the Bcache is disabled. In multiprocessor systems a CPU should probably be removed from the system rather than being used with the Bcache off. In a single processor system there

may be effects to IO subsystem performance and latency due to the high NDAL and memory utilization.

15.3.3.1 Special Considerations for Cache and Memory Errors

Cache and memory error recovery requires special considerations:

- Cache and memory error recovery should always be done with the Pcache and VIC off and the Bcache in error transition mode (ETM). (In certain cases, the last part of recovery must be done with the Bcache off.) See Section 15.3.3.1.1.1, Cache Enable, Disable, and Flush Procedures.
- Bcache flush is necessary before re-enabling the Bcache whenever it is in ETM. See Section 15.3.3.1.1, Cache Coherence in Error Handling.
- Bcache flush should be always be done one block at a time, recapturing the relevant error registers between each block flush.
- Cache coherence requires a specific procedure for re-enabling the caches. See Section 15.3.3.1.1, Cache Coherence in Error Handling.
- Error recovery should be performed starting with the most distant component and working toward the CPU and Ebox. System environment memory errors should be processed first, followed by NDAL errors, Bcache fill errors, Bcache tag store and data RAM errors, Pcache errors, TB errors, and, finally, VIC errors.
- NDAL errors are cleared by writing the write-one-to-clear bits in NESTS. The suggested way to do this is to write a one to the specific error bit.
- Bcache fill errors are cleared by writing the write-one-to-clear bits in CEFSTS. The suggested way to do this is to write a one to the specific error bit. Special recovery procedures may be necessary after Bcache fill errors. See Section 15.3.3.1.2, Special Writeback Cache Recovery Situations and Procedures.
- Bcache tag store errors are cleared by writing the write-one-to-clear bits in BCETSTS. The suggested way to do this is to write a one to the specific error bit. Special recovery procedures may be necessary after Bcache uncorrectable tag store errors. See Section 15.3.3.1.2, Special Writeback Cache Recovery Situations and Procedures.
- Bcache data RAM errors are cleared by writing the write-one-to-clear bits in BCEDSTS. The suggested way to do this is to write a one to the specific error bit. Special recovery procedures may be necessary after Bcache uncorrectable data RAM errors. See Section 15.3.3.1.2, Special Writeback Cache Recovery Situations and Procedures.
- Hardware ETM is cleared by writing the write-one-to-clear bit in CCTL. The suggested way to do this is to write the value saved during error state collection back to the register.
- Pcache tag and data store errors are cleared by writing the write-one-to-clear bits in PCSTS. The suggested way to do this is to write a one to the specific error bit. Pcache flush is necessary after Pcache tag store parity errors. See Section 15.3.3.1.1.1, Cache Enable, Disable, and Flush Procedures.
- TB errors are cleared by writing the write-one-to-clear bits in TBSTS. The suggested way to do this is to write a one to the specific error bit.
- PTE read errors are cleared by writing the PTE error write-one-to-clear bits in PCSTS. The suggested way to do this is to write a one to the specific error bit.

- VIC errors are cleared by writing the write-one-to-clear bits in ICSR. The suggested way to do this is to write a one to the specific error bit. VIC flush and re-enable is necessary after VIC tag store parity errors. See Section 15.3.3.1.1.1, Cache Enable, Disable, and Flush Procedures.

15.3.3.1.1 Cache Coherence In Error Handling

Certain procedures must be followed in order to maintain cache coherence while enabling NVAX caches. Since many errors cause caches to be disabled, and since cache and memory error recovery is normally done with the Pcache and VIC off and the Bcache in ETM, the complete cache enable procedure is done as part of recovery from all cache and memory errors.

Once the Bcache is in ETM mode, it will not be coherent with memory if it is re-enabled before being flushed. This is because writes (from the Mbox) to blocks which happen to be VALID_UNOWNED in the Bcache are not copied into the Bcache data RAMs. These writes are only sent out on the NDAL. Once the Bcache is put in ETM by hardware or software action, a Bcache flush must be done before re-enabling the Bcache. The procedure is described in the next section.

While the Bcache in in ETM or off, the Pcache will stay coherent with memory. However, before the Bcache is re-enabled, the Pcache must be disabled. After the Bcache is re-enabled, the Pcache must be flushed before it is re-enabled. The procedure is described in the next section. If a Pcache tag parity error occurred, the flush procedure given is sufficient to clean up the Pcache tag store.

The VIC (virtual instruction cache) is not automatically kept coherent with memory. It is flushed as a side effect of the REI instruction (as required by the VAX architecture). Normally in error recovery, there is no definite need to flush the VIC. For consistency and for the sake of beginning error retry in a known state, flushing the VIC during error recovery is recommended. However, in the event of VIC tag parity errors, the complete VIC flush procedure described in the next section must be done.

The TB is not automatically kept coherent with memory. Software uses the TBIS and TBIA functions to maintain coherence, and the LDPCTX instruction clears the process PTEs in the TB. Normally in error recovery, there is no definite need to flush the TB. For consistency and for the sake of beginning error retry in a known state, flushing the TB during error recovery is recommended. When a TB parity error occurs, Mbox hardware flushes the TB by itself (via an internally generated TBIA), but it would be appropriate for software to test the TB after a parity error. This is discussed in Section 15.3.3.1.3.

15.3.3.1.1.1 Cache Enable, Disable, and Flush Procedures

To enable the NVAX caches, the caches are flushed and enabled in a specific order. The ordering is necessary for coherence between the Bcache, Pcache, and memory. For simplicity, one procedure is given for enabling the NVAX caches, even though variations on the procedure may also produce correct results. Disabling the caches can be done in any order, though one procedure is given here.

In error handling, the VIC and Pcache are disabled while the Bcache is placed in ETM. The Bcache flush from ETM procedure is done to turn off the Bcache altogether. The cache enable procedure assumes that the Bcache is completely off at the start.

15.3.3.1.1.1.1 Disabling the NVAX Caches for Error Handling (Leaving the Bcache in ETM)

This is the procedure for disabling the NVAX caches (placing the Bcache in ETM):

NOTE

> These procedures will be supplied with MACRO coding examples.

- Disable the VIC:

```
TBS (MTPR to ICSR)
```

- Disable the Pcache:

```
TBS (MTPR to PCCTL)
```

- Put the Bcache in software ETM:

```
TBS (MTPR to CCTL)
```

15.3.3.1.1.1.2 Flushing and Disabling the Bcache

This is the procedure for flushing the Bcache and disabling it:

- Flush and disable the Bcache:

```
TBS (Loop on MTPR to BCFLUSH IPRs, then clear ETM bits in CCTL)
```

Errors can occur as a result of flushing the Bcache. Before carrying out the procedure, BCEDSTS and BCETSTS should be clear of unrecoverable errors, and NESTS should be clear of unrecoverable outgoing errors. The MTPRs to BCFLUSH IPRs should be done one block at a time, checking the BCEDSTS and BCETSTS error registers after each one. (The MFPR from BCEDSTS or BCETSTS will not finish until all the Bcache accesses which result from the MTPR to BCFLUSH are done.) Otherwise any unrecoverable error which occurs during the flush may become a lost unrecoverable error and a system crash will most likely be necessary.

Errors which occur while flushing the Bcache are separate errors and should be handled independently of the initial error. However, certain errors may be expected during the flush procedure, based on the initial error. Also, the successful outcome of the Bcache flush procedure is important in determining whether to retry or restart the interrupted or machine checked instruction stream.

15.3.3.1.1.1.3 Enabling the NVAX Caches

The procedure for enabling the NVAX caches after an error is the same as is used to initialize the caches after power-up. See Section 16.4, Cache initialization). This procedure ensures that error retry/restart occurs with the caches in a known state. The procedure is outlined below.

- **The caches must all be disabled and the Bcache must be disabled (not just in ETM).** Follow the above procedures to reach this state.
- Flush the Bcache (Loop on MTPR to BCTAG IPRs).
- Enable the Bcache (MTPR to CCTL).
- Flush the Pcache (Loop on MTPR to PCTAG IPRs).
- Enable the Pcache (MTPR to PCCTL).

- Flush the TB:

```
MTPR #0, #PR19$_TBIA
```

- Flush the VIC (Loop on MTPRs to VMAR and VTAG, writing an initial value).
- Enable the VIC (MTPR to ICSR).

15.3.3.1.2 Special Writeback Cache Recovery Situations and Procedures

Writeback caching can lead to a couple of special error cases. Some of them can be recovered. Sometimes, further state determination or state capture is required after the error cause determination guided by the parse trees in the sections on machine check exceptions and hard and soft errors. Further analysis may also be necessary.

15.3.3.1.2.1 Bcache Uncorrectable Error During Writeback

When a Bcache uncorrectable data RAM error occurs in a writeback, the status, cache index, and error syndrome are captured in BCEDSTS, BCEDIDX, and BCEDECC. As it is written back, the data is tagged-bad via the BADWDATA NDAL command. However, the address of the lost data is not captured in the Bcache error registers (for implementation reasons). For this reason, sending BADWDATA on the NDAL is treated as if it were an error by the bus interface unit (BIU). This means the full address is captured in NEOADR while the status is captured in NESTS. This writeback can sit in the writeback queue in the BIU for an indefinite amount of time. If a Bcache uncorrectable error on writeback is detected, but NESTS does not show any outgoing error status, the writeback queue must be drained to continue the analysis and recovery. This is most easily accomplished by the following IPR write.

```
MTPR #PR19$_CWB,R0
```

S_NESTS should be reloaded from NESTS after this operation. If S_NESTS does not show the the BADWDATA error status after draining the writeback queue, and it shows no other outgoing error, then there is a serious inconsistency and the system should be crashed.

15.3.3.1.2.2 Memory State

Memory in NVAX systems supports the writeback cache by maintaining some amount of state for each hexaword (each cachable block) in memory. In XMI2 systems with XMA2 memory modules, an ownership bit, and interlock bit, and an owner ID is stored for each hexaword. In OMEGA systems, only an ownership bit is stored for each block. Other system environments are possible.

The effect of a given error on the stored ownership bit in memory is system specific. Since the system environment is not directly aware of errors which occur inside the NVAX CPU chip, the system specific behavior is limited to the result of system environment errors.

It is always assumed that a an ownership read command no-ACKed on the NDAL doesn't affect the ownership bit in memory. Depending on the system, the state of memory's ownership bit (and other such state) may be UNPREDICTABLE or determinate after errors in data returned for ownership reads. If it is determinate, it may be set or reset, possibly depending on which fill quadword had the error and on the sort of error that occurred.

This specification assumes that memory does not reset a set ownership bit on a WDISOWN until all four quadwords have been successfully received by memory (as is stated in Chapter 3).

15.3.3.1.2.2.1 Accessing Memory State

In recovering from certain errors it is necessary to read (or access by some means) the state memory has stored with each hexaword. This specification assumes a routine called MEMORY_STATE exists which returns this state given a block address.

MEMORY_STATE may have system specific errors and side effects. For example, in XMI2 systems this routine may cause a read timeout error in the memory module and a corresponding machine check. Software must be prepared to handle this. Before calling MEMORY_STATE, software should confirm that all registers which may end up reporting expected errors are clear of errors. This helps minimize the possibility that an unrelated error event is ignored because it appears to be an expected error. In the XMI2 example, within the NVAX CPU, CEFSTS is the register to check because a memory read timeout is the only error which is expected as a side effect of MEMORY_STATE.

15.3.3.1.2.2.2 Repairing Memory State (Fill Errors)

In recovering from various Bcache fill errors it is necessary to reset the ownership state in memory. In some system environments, this can be done without writing the data in memory. In others reseting the ownership state may have the side effect of altering the data stored in the memory block.

In cases where the fill error resulted from "lost"[1] data which can not be recovered, the ownership bit may still be set in memory while no cache owns the block. If the data is private to one process, then the system may be able to continue operating after killing that one job. The system dependent procedure is then used to reset the ownership bit.

For certain Bcache fill errors, an attempt is made to reset the ownership bit in memory, while maintaining or restoriong the correct data to the memory block.

- All the data is in memory. One or more quadwords of (the same) data are also in the cache. Memory's ownership bit is set (meaning it "thinks" a cache owns the block). The owner ID stored with the block in memory indicates this CPU. The cache tag for the block does not indicate the block is owned. (In general, if no writes to this block timeout, and the block is private to one process, then the repair can be done.)
- All the data is in memory. One or more quadwords of data are also in the cache, and one quadword has been altered by the Cbox in processing a write to that block from the Mbox. Memory's ownership bit is set (meaning it "thinks" a cache owns the block). The owner ID stored with the block in memory indicates this CPU. The cache tag for the block does not indicate the block is owned. (In general, if no writes to this block timeout, and the block is private to one process, then the repair can be done.)

NOTE

> If an owner ID for each block is not stored in memory, then recovery of the lost data is not recommended. The data should be treated as lost, and the appropriate system actions should be taken.

[1] In this case the more general sense of "lost" is implied. That is, memory's ownership bit is set but no cache writes the data back when a read is done to that location. In some systems, it may be possible to identify which CPU memory "thinks" owns the data, but it is often not possible to determine which error caused this situation to arise.

To recover from the first situation listed above in an XMI2 system, for instance, one of the correct quadwords in the Bcache is accessed (see Section 15.3.3.1.2.3) and used in the XMI2 procedure for reseting mekory's ownership bit. The side effect of this procedure is the the data extracted from the Bcache is written to memory. Given that the block is private to one process and no writes have timed out in memory, this data is still correct. (Note that software must somehow ensure that no writes to this block are pending in the memory before beginning the repair. This can be done by waiting an amount of time equal to an XMA2 write timeout time.)

To recover from the second situation listed above in an XMI2 system, the same procedure is followed, but the data written back is part of the known-altered quadword. The remainder of the known-altered quadword is written to the block after the repair.

15.3.3.1.2.2.3 Repairing Memory State (Tagged-Bad Locations)

In recovering from Bcache uncorrectable data RAM errors on writebacks is necessary to reset the tagged-bad-data state for a block in memory. This is a system specific procedure. In general, before clearing the tagged-bad data state of memory, software must first ensure that no more accesses to the block can occur. Otherwise there is the danger that some process on some other processor or a DMA IO device will see incorrect data and not detect an error.

In XMI2, a sequence of operations involving writes to registers in a memory module followed by a write to the memory block in question is required. To do this the Bcache should be off, because NVAX will not issue a write to memory when the cache is enabled (or is in ETM and the block's tag indicates VALID-OWNED).

In OMEGA, reseting tagged-bad-data state in memory requires that a full quadword write to the tagged-bad quadword be accomplished. The most straightforward way for NVAX software to do this is to fill in the Bcache tag store and data RAMs with a VALID-OWNED block and force a writeback (via a MTPR to BCFLUSH).

15.3.3.1.2.3 Extracting Data from the Bcache

To extract data from the Bcache, the Bcache is placed in FORCE_HIT mode. Before this is done, the Bcache must be off.

With the Bcache flushed and disabled, set the Bcache in FORCE_HIT mode and extract the data. Note that the code which executes this procedure and its local data must be in IO space. The TB entries (PTEs) which map this code and local data must be fixed in the TB. (This is most easily done by flushing the TB via an MTPR to TBIA and then accessing all the relevant pages in pages in sequence.) Otherwise Bcache FORCE_HIT will interfere with instruction fetch, operand access, and PTE fetches in TB miss sequences.

The following instruction places the Bcache in FORCE_HIT mode:

```
TBS (MTPR to CCTL)
```

With the Bcache in FORCE_HIT mode, a read in memory space of any address whose index portion matches the index of the cache data will return the data (provided there is no uncorrectable data RAM error). This is most easily accomplished by reading from the true address of the data.

NOTE

> In FORCE_HIT mode, Bcache data RAM ECC errors are detected (unless CCTL<DISABLE_ERRORS> is set). Software should prepare for an ECC error (BCEDSTS unrecoverable error bits should be clear).

The Bcache is restored to the disabled state by:

```
TBS (MTPR to CCTL)
```

15.3.3.1.2.4 Address Determination Procedure for Recovery from Uncorrectable Bcache Data RAM Errors

After an uncorrectable data RAM error in the Bcache, only the index of the block is stored, not the complete physical address. The procedure for constructing the physical address of the error is given here. It depends on the assumption that the block has not been replaced. The detailed error descriptions only refer to this procedure when this assumption is valid.

This is the procedure for constructing a physical address from the contents of S_BCEDIDX and the tag indicated by that register. It uses the Bcache tag ECC check routine found in Section 15.10. If an unrecoverable ECC error if found in the tag, then the address can not be determined directly.

```
TBS
(Read tag via MFPR from BCTAG IPRs. Check that the tag data and check bits are correct or correctable.
Extract the address tag portion of the corrected result and combine with S_BCEDIDX.)
```

NOTE

> The above procedure is used in the event of a Bcache data RAM error. If it fails because the tag also has an uncorrectable error, then the error should be considered unrecoverable. However, the search procedure described in the next section could be used to obtain useful information for the error log (specifically, which blocks this CPU has marked owned in memory for this cache index).

15.3.3.1.2.5 Special Address Determination Procedure for Recovery from Uncorrectable Bcache Tag Store Errors

An uncorrectable tag store error in the Bcache can cause certain interesting error cases. In some of these cases data may be lost (the copy in the Bcache was overwritten). In other cases, the data is still good in the cache. In all cases, the address of the lost data is not directly known. A special procedure must be used to determine this address.

This section describes the generic address determination procedure for use in recovering from uncorrectable tag store errors. Specific error event descriptions in Section 15.5.2, Section 15.7.1, and Section 15.8.1 refer to this procedure for address determination. The possible outcomes of this procedure are:

- The single address of a lost data block is found. Retry and recovery information for the error is found in the specific error event description which referred to this address determination procedure.
- No address is found. It can be assumed that no block was owned by the Bcache (or the error was transient). Retry and recovery information for the error is found in the specific error event description which referred to this address determination procedure.

- Multiple addresses are found. This is a multiple unrecoverable error situation, and the system should be crashed.

The procedure for determining the address of a lost data block follows. Note that this procedure assumes the relevant tag in the Bcache is not (correct or correctable) VALID-OWNED. This procedure is for analyzing the result of errors in that tag.

This procedure assumes that MEMORY_STATE will return the ownership state and the physical ID of the CPU which memory "thinks" owns the block. If memory does not store an owner ID and there is exactly one writeback cache in the system, then the lack of an owner ID might not prevent error recovery.

- The Bcache should be in ETM.
- Search for the address:

```
TBS
(Search all memory block addresses whose index portion matches the index of the Bcache tag with the
error. Check memory state for the block. If this CPU is the owner of that block, then the block is lost
Continue the search even if one lost block is found. Zero, one, or multiple lost blocks could be presen
Note that in systems with no owner ID bits in memory and exactly one CPU, it may or may not be possible
to assume that every owned block is owned by the CPU. It may be necessary to confirm each set owned bit
reading the marked location.  If it is owned by this CPU, the read should timeout.)
```

NOTE

This procedure is specific to recovering from tag store errors in one CPU. So when the memory state for a block indicates another cache in the system owns a particular block, that block is not counted as lost. That block may be "lost" in the more general sense (if the cache indicated as the owner no longer "knows" that it owns the block or is somehow unable to write it back.) The purpose here is only to find blocks that are definitely lost as a result of errors involving this CPU.

15.3.3.1.3 Cache and TB Test Procedures

TBS

OUTLINE OF TO-BE-SPECIFIED TEST PROCEDURES

Testing is generally done using the force hit mode of a cache. The code and data of the test procedure must reside in IO space. Assuming memory management is enabled during this procedure, the needed PTEs must be in the TB before entering force hit mode in the Pcache or Bcache. For the Bcache, testing should be done with errors disabled. The ECC logic should be tested thoroughly on one location by forcing various check bit patterns and examining the syndrome latched on the read (BCEDECC is loaded on every read in Bcache disable-errors mode). Pcache and VIC parity checking should be tested by writing bad parity into the arrays. TB testing may be accomplished by writing to MTBTAG and MTBPTE (with care to not change any TB entry necessary for the test code and data and not to cause two TB entries to exist for one address). PROBER and PROBEW (setting PSL<PRV_MOD>) are then used to verify the protection bits. Testing the modify bit would be difficult, though approaches exist.

15.3.4 Error Retry

Error retry is a function of the error notification (machine check or error interrupt), error type, and error state. The sections below specify the conditions under which the instruction stream may be restarted.

If retry is to be attempted, the stack must be trimmed of all parameters except the PC/PSL pair. This is necessary only for machine checks, because error interrupts do not provide any additional parameters on the stack. An REI will then restart the instruction stream and retry the error. Some form of software loop control should be provided to limit the possibility of an error loop. Note that pending error interrupts may be taken before the retry occurs, depending on the IPL of the interrupted or machine checked code.

Strictly speaking, an REI from a hard or soft error interrupt handler is not a retry since these interrupts are recognized between macroinstructions. A machine check exception is an instruction abort, and an REI from the handler will cause the failing instruction to be retried (provided retry is indicated by analysis). What these cases all have in common is that the interrupted instruction stream is restarted. This is only done when the result of error analysis and recovery is such that all damaged state has been repaired and there is no reason to suspect that incorrect results will be produced if the image is restarted and another error does not occur.

If complete recovery from one or more errors is not possible (i.e., some state is lost or it is impossible to determine what state is lost), possibly the entire system will have to be crashed, a single process will have to be deleted, or some other action will have to be taken. Software must determine if the error is fatal to the current process, to the processor, or to the entire system, and take the appropriate action.

It is expected that software handles machine checks, soft error interrupts, and hard error interrupts independently. For example, after handling a machine check from which retry is to occur, software does not check for errors which might cause a pending hard or soft error interrupt. The machine check handler is exited via REI (after trimming the machine check information off the stack). If the IPL of the machine checked instruction stream is low enough, any pending hard or soft error interrupt is taken before the retry occurs. However, if the interrupted instruction stream was running at high IPL, then it will continue oblivious of remaining errors.

15.3.4.1 General Multiple Error Handling Philosophy

Multiple errors may be reported at the same time. In some cases the NVAX CPU pipeline will contain multiple operand prefetches to the same memory block. This can cause multiple errors from a single non-transient failure. It could also occur that two separate errors occur at nearly the same time and are thus reported simultaneously.

Multiple error scenarios may be grouped into the following three classes:

1. Multiple distinct errors for which no error report interferes with the analysis of any other (e.g., no lost error bits set).
2. Multiple errors which could have been caused by the NVAX CPU pipeline issuing more than one reference to a given block before the error interrupt or machine check forced a pipeline flush.
3. Multiple errors for which analysis is complicated because the reports interfere with each other.

It is the intent of this chapter to recover from class 1 (above) by simply treating the errors as separate and recovering from each in turn. Retry or restart evaluation is based on the cumulative result of the recovery and repair procedures for each error.

For class 2, specific cases are identified in which lost errors are tolerated. These cases are selected because the NVAX pipeline can easily cause them (given one error), and because sufficient safeguards exist to ensure that correct operation is maintained. Section 15.3.4.2 lists these cases.

Class 3 scenarios are generally not considered recoverable. The system is simply crashed in those cases.

Note that lost correctable errors are not considered serious problems since hardware recovers from those automatically.

15.3.4.2 Retry Special Cases

The multiple error scenarios which are handled are listed below. They are made likely by the NVAX pipeline's tendency to prefetch operands. The safeguard that exists in all cases is that errors inconsistent with correct operation after the error (such as lost data) will invariably cause a hard error interrupt or be detectable by the analysis accompanying the machine check or soft error interrupt.

- Lost Bcache data RAM uncorrectable ECC errors and addressing errors. (BCEDSTS<LOST_ERR>)
- Lost Bcache fill errors (timeouts and RDEs). (CEFSTS<LOST_ERR>)
- Lost NDAL output errors (No-ACKs). (NESTS<LOST_OERR>)

NOTE

Retry from a machine check is done even when a hard error interrupt might be pending. If the machine checked I-stream were running at high enough IPL, it would not be interrupted immediately. Typical hard error causes are write errors. They can not cause a machine check. So the fact that a serious error is ignored in the machine check retry equation is not considered a problem. The other error would probably have occurred anyway and it would not have interrupted the I-stream until IPL was lowered.

15.4 Console Halt and Halt Interrupt

A console halt is not an exception, but rather a transfer of control by the NVAX CPU microcode directly into console macrocode at the boot ROM address E0040000 (hex). Console halts are initiated at powerup, by certain microcode-detected double error conditions, and by the assertion of the external halt interrupt pin, HALT_L.

There is no exception stack frame associated with a console halt. Instead, the SAVPC and SAVPSL processor registers provide the necessary information. The format of SAVPC (IPR 42) is shown in Figure 15–1.

Figure 15–1: IPR 2A (hex), SAVPC

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                           Saved PC                                            | :SAVPC
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

The PSL, halt code, MAPEN<0>, and a validity bit are saved in SAVPSL (IPR 43). The format of SAVPSL is shown in Figure 15–2. The halt codes are shown in Table 15–2.

Figure 15–2: IPR 2B (hex), SAVPSL

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                  PSL<31:16>                   |  |  |   Halt Code     |      PSL<7:0>         | :SAVPSL
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                 |  |
                                      MAPEN<0> --+  |
                                Invalid SAVPSL if 1 --+
```

The possible halt codes that may appear in SAVPSL<13:8> are listed in Table 15–2.

Table 15–2: Console Halt Codes

Mnemonic	Code (Hex)	Meaning
ERR_HLTPIN	02	HALT_L pin asserted
ERR_PWRUP	03	Initial power up
ERR_INTSTK	04	Interrupt stack not valid
ERR_DOUBLE	05	Machine check during exception processing
ERR_HLTINS	06	HALT instruction in kernel mode
ERR_ILLVEC	07	Illegal SCB vector (bits <1:0> = 11)
ERR_WCSVEC	08	WCS SCB vector (bits <1:0> = 10)
ERR_CHMFI	0A	CHMx on interrupt stack
ERR_IE0	10	ACV/TNV during machine check processing
ERR_IE1	11	ACV/TNV during kernel-stack-not-valid processing

Table 15–2 (Cont.): Console Halt Codes

Mnemonic	Code (Hex)	Meaning
ERR_IE2	12	machine check during machine check processing
ERR_IE3	13	machine check during kernel-stack-not-valid processing
ERR_IE_PSL_26_24_101	19	PSL<26:24> = 101 during interrupt or exception
ERR_IE_PSL_26_24_110	1A	PSL<26:24> = 110 during interrupt or exception
ERR_IE_PSL_26_24_111	1B	PSL<26:24> = 111 during interrupt or exception
ERR_REI_PSL_26_24_101	1D	PSL<26:24> = 101 during REI
ERR_REI_PSL_26_24_110	1E	PSL<26:24> = 110 during REI
ERR_REI_PSL_26_24_111	1F	PSL<26:24> = 111 during REI

NOTE

In certain error conditions detected during the execution of a string instruction, the state packup sequence leaves the FPD bit set in the SAVPSL register, but the SAVPC register pointing at the instruction following the string instruction, rather than at the string instruction itself. If the FPD bit is no set in the SAVPSL register, SAVPC is correct. As error halts are not normally restartable, this is not a problem. For a console halt due to the assertion of the HALT_L pin, which is the only normally restartable console halt, SAVPC is always correct, even if the halt interrupt was detected during the execution of a string instruction.

At the time of the halt, the current stack pointer is saved in the appropriate IPR (0 to 4), and SAVPSL<31:16,7:0> are loaded from PSL<31:16,7:0>. SAVPSL<15> is set to MAPEN<0>. SAVPSL<14> is set to 0 if the PSL is valid and to 1 if it is not (SAVPSL<14> is undefined after a halt due to a system reset). SAVPSL<13:8> is set to the console halt code.

To complete the hardware restart sequence and thereby pass control to the console macrocode, the state shown in Table 15–3 is initialized.

Table 15–3: CPU State Initialized on Console Halt

State	Initialized Value
SP	IPR 4 (IS)
PSL	041F0000 (hex)
PC	E0040000 (hex)
MAPEN	0
ICCS	0 (after reset, code=3, only)
SISR	0 (after reset, code=3, only)
ASTLVL	4 (after reset, code=3, only)
PAMODE	0 (after reset, code=3, only)
BPCR<31:16>	FECA(hex) (after reset, code=3, only)

Table 15–3 (Cont.): CPU State Initialized on Console Halt

State	Initialized Value
CPUID	0 (after reset, code=3, only)
all else	undefined

15.5 Machine Checks

The machine check exception indicates a serious system error. Under certain conditions, the error may be recoverable by restarting the instruction. The recoverability is a function of the machine check code, the VAX Restart bit (VR) in the machine check stack frame, the opcode, the state of PSL<FPD>, the state of certain second-error bits in internal error registers, and most probably, the external error state.

A machine check results from an internally detected consistency error (e.g., the microcode reaches an "impossible" state), or a hardware detected error (e.g., an uncorrectable Bcache ECC error on a data read).

A machine check is technically a macro instruction abort. The NVAX CPU microcode attempts to convert the condition to a fault by unwinding the current instruction, but there is no guarantee that the instruction can be properly restarted. As much diagnostic information as possible is pushed on the stack and provided in other error registers. The rest of the error parsing is then left to the operating system.

When the software machine check handler receives control, it must explicitly acknowledge receipt of the machine check via a write of any value to the MCESR processor register with the following instruction:

```
        MTPR     #0,#PR19$_MCESR
```

Figure 15–3: IPR 26 (hex), MCESR

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x   x| :MCESR
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

15.5.1 Machine Check Stack Frame

The machine check stack frame is shown in Figure 15–4. The fields of the stack frame are described in Table 15–4, and the possible machine check codes are listed in Table 15–5. The contents of all fields not explicitly defined in Table 15–4 are UNDEFINED.

Figure 15–4: Machine Check Stack Frame

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                24 (byte count of parameters, not including this longword)                     | :(SP)
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| ASTLVL | x  x  x  x  x| Machine Check Code    | x  x  x  x  x  x  x  x|         CPUID         |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                      INT.SYS register                                         |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                      SAVEPC register                                          |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                        VA register                                            |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                        Q register                                             |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|    Rn     | x  x|Mode |        Opcode         | x  x  x  x  x  x  x|VR| x  x  x  x  x  x  x|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                           PC                                                  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                           PSL                                                 |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
```

Table 15–4: Machine Check Stack Frame Fields

Longword	Bits	Contents
(SP)+0	31:0	Byte count—This longword contains the size of the stack frame in bytes, not including the PC, PSL, or the byte count longword. Stack frame PC and PSL values should always be referenced using this count as an offset from the stack pointer.
(SP)+4	31:29	ASTLVL—This field contains the current value of the VAX ASTLVL register.
	23:16	Machine check code—This longword contains the reason for the machine check, as listed in Table 15–5.
	7:0	CPUID—This field contains the current value of the VAX CPUID register.
(SP)+8	31:0	INT.SYS register—This longword contains the value of the INT.SYS register and read onto the Abus by the microcode. The fields in this register are described in Chapter 10.
(SP)+12	31:0	SAVEPC—This field contains the SAVEPC register which is loaded by microcode with the PC value in certain circumstances. It is used in error handling for PTE read errors with PSL<FPD> set in this stack frame.
(SP)+16	31:0	VA register—This longword contains the contents of the Ebox VA register, which may be loaded from the output of the ALU.

Table 15–4 (Cont.): Machine Check Stack Frame Fields

Longword	Bits	Contents
(SP)+20	31:0	Q register—This longword contains the contents of the Ebox Q register, which may be loaded from the output of the shifter.
(SP)+24	31:28	Rn—This field contains the value of the Rn register, which is used to obtain the register number for the CVTPL and EDIV instructions. In general, the value of this field is UNPREDICTABLE.
	25:24	Mode—This field contains a copy of PSL<CUR_MOD>.
	23:16	Opcode—This field contains bits <7:0> of the instruction opcode. The FD bit is not included.
	7	VR—This field contains the VAX Restart bit, which is used to communicate restart information between the microcode and the operating system. If this bit is set, no architectural state has been changed by the instruction which was executing when the error was detected. If this bit is not set, architectural state was modified by the instruction.

Table 15–5: Machine Check Codes

Mnemonic	Code (Hex)	Meaning
MCHK_UNKNOWN_MSTATUS	01	Unknown memory management fault parameter returned by the Mbox (see Section 15.5.2.1)
MCHK_INT.ID_VALUE	02	Illegal interrupt ID value returned in INT.SYS (see Section 15.5.2.2)
MCHK_CANT_GET_HERE	03	Illegal microcode dispatch occurred (see Section 15.5.2.3)
MCHK_MOVC.STATUS	04	Illegal combination of state bits detected during string instruction (see Section 15.5.2.4)
MCHK_ASYNC_ERROR	05	Asynchronous hardware error occurred (see Section 15.5.2.5)
MCHK_SYNC_ERROR	06	Synchronous hardware error occurred (see Section 15.5.2.6)

15.5.2 Events Reported Via Machine Check Exceptions

This section describes all the errors which can cause a machine check exception. A parse tree is given which shows how to determine the cause of a given machine check. After that, there is a description of each error. For each error, the recovery procedure is given. Where appropriate, the conditions for retry are given. See Section 15.3.3 and Section 15.3.4 for more on error recovery and error retry.

Figure 15–5 is a parse tree which should be used to analyze the cause of a machine check exception. The errors shown in the parse tree are described in detail in the sections following the figure. The section is indicated in parenthesis with each error. Note that it is assumed that the state being analyzed is the saved state, as described in Section 15.3.1. Otherwise the state could change during the analysis procedure, leading to possibly incorrect conclusions. (See Section 15.3.2 for general information about error analysis.)

Figure 15–5: Cause Parse Tree for Machine Check Exceptions

```
MACHINE CHECK
----+ (select one)
    |
    |  MCHK_UNKNOWN_MSTATUS
    +--------------------------------------------------> Unknown memory management status error (Section 15.5.2.1)
    |
    |  MCHK_INT.ID_VALUE
    +--------------------------------------------------> Illegal interrupt ID error (Section 15.5.2.2)
    |
    |  MCHK_CANT_GET_HERE
    +--------------------------------------------------> Presumed impossible microcode address reached
    |                                                    (Section 15.5.2.3)
    |  MCHK_MOVC.STATUS
    +--------------------------------------------------> MOVCx status encoding error (Section 15.5.2.4)
    |
    |  MCHK_ASYNC_ERROR
    +-----+ (select all, at least one)
    |     |
    |     |  S_TBSTS<LOCK>
    |     +-----+ (select all)
    |     |     |
    |     |     |  S_TBSTS<DPERR>
    |     |     +--------------------------------------> TB PTE data parity error (Section 15.5.2.5.1)
    |     |     |
    |     |     |  S_TBSTS<TPERR>
    |     |     +--------------------------------------> TB tag parity error  (Section 15.5.2.5.1)
    |     |     |
    |     |     |  none of the above
    |     |     +--------------------------------------> Inconsistent status (no TBSTS error bits set)
    |     |                                              (Section 15.5.2.7)
    |     |  S_ECR<S3_STALL_TMEOUT>
    |     +--------------------------------------------> S3 stall timeout error (Section 15.5.2.5.2)
    |     |
    |     |  none of the above
    |     +--------------------------------------------> Inconsistent status (no asynchronous machine check error
    |                                                    set) (Section 15.5.2.7)
    |  MCHK_SYNC_ERROR
    +-----+ (select all, at least one)
    |     |
    |     |  S_ICSR<LOCK>
    |     +-----+ (select all, at least one)
    |     |     |
    |     |     |  S_ICSR<DPERR>
    |     |     +--------------------------------------> VIC (virtual instruction cache) data parity error
    |     |     |                                        (Section 15.5.2.6.1)
    |     |     |  S_ICSR<TPERR>
    |     |     +--------------------------------------> VIC tag parity error (Section 15.5.2.6.1)
    |     |     |
    |     |     |  none of the above
    |     |     +--------------------------------------> Inconsistent status (no ICSR error bits set)
    |     |                                              (Section 15.5.2.7)
    v     v
    1     2
```

Figure 15–5 Cont'd on next page

Figure 15–5 (Cont.): Cause Parse Tree for Machine Check Exceptions

```
1    2
v    v
|    | S_BCEDSTS<LOCK> AND
|    | NOT S_PCSTS<PTE_ER>
|    +----+ (select one)
|    |    |
|    |    | S_BCEDSTS<BAD_ADDR>
|    |    +----+ (select one)
|    |    |    |
|    |    |    | S_BCEDSTS<DR_CMD>=DREAD
|    |    |    +----------------------------------> Bcache data RAM addressing error on D-stream read
|    |    |    |                                    or read-lock (Section 15.5.2.6.2)
|    |    |    | S_BCEDSTS<DR_CMD>=IREAD
|    |    |    +----------------------------------> Bcache data RAM addressing error on I-stream read
|    |    |    |                                    (Section 15.5.2.6.2)
|    |    |    | otherwise
|    |    |    +----------------------------------> Not a synchronous machine check cause (see soft and
|    |    |                                         hard error interrupt events)
|    |    | S_BCEDSTS<UNCORR>
|    |    +----+ (select one)
|    |    |    |
|    |    |    | S_BCEDSTS<DR_CMD>=DREAD
|    |    |    +----------------------------------> Bcache data RAM uncorrectable ECC error on D-stream read
|    |    |    |                                    or read-lock (Section 15.5.2.6.2)
|    |    |    | S_BCEDSTS<DR_CMD>=IREAD
|    |    |    +----------------------------------> Bcache data RAM uncorrectable ECC error on I-stream read
|    |    |    |                                    (Section 15.5.2.6.2)
|    |    |    | otherwise
|    |    |    +----------------------------------> Not a synchronous machine check cause (see soft and
|    |    |                                         hard error interrupt events)
|    |    | none of the above
|    |    +---------------------------------------> Inconsistent status (no BCEDSTS unrecoverable error bits
|    |                                              set) (Section 15.5.2.7)
|    | S_BCEDSTS<LOST_ERR> AND
|    | NOT S_PCSTS<PTE_ER>
|    +--------------------------------------------> Lost unrecoverable Bcache data RAM error
|    |                                              (Section 15.5.2.6.3)
|    | S_CEFSTS<LOCK> AND
|    | NOT S_PCSTS<PTE_ER>
|    +----+ (select one)
|    |    |
|    |    | S_CEFSTS<TIMEOUT>
|    |    +----+ (select one)
|    |    |    |
|    |    |    | S_CEFSTS<TO_MBOX> AND
|    |    |    | (NOT S_CEFSTS<REQ_FILL_DONE>)
|    |    |    +----+ (select one)
|    |    |    |    |
|    |    |    |    | S_CEFSTS<IREAD>
|    |    |    |    +-----------------------------> I-stream NDAL read timeout error (Section 15.5.2.6.4)
|    |    |    |    |
|    |    |    |    | S_CEFSTS<OREAD>
|    |    |    |    +-----------------------------> D-stream NDAL ownership read timeout error
|    |    |    |    |                               (Section 15.5.2.6.4)
|    |    |    |    | otherwise
|    |    |    |    +-----------------------------> D-stream NDAL read timeout error (read only operand)
|    |    |    |                                    (Section 15.5.2.6.4)
|    |    |    | otherwise
|    |    |    +----------------------------------> Not a synchronous machine check cause (see soft and
|    |    |                                         hard error interrupt events)
v    v    v
1    2    3
```

Figure 15–5 Cont'd on next page

Figure 15–5 (Cont.): Cause Parse Tree for Machine Check Exceptions

```
1    2    3
v    v    v
|    |    | S_CEFSTS<RDE>
|    |    +----+ (select one)
|    |    |    |
|    |    |    | S_CEFSTS<TO_MBOX> AND
|    |    |    | (NOT S_CEFSTS<REQ_FILL_DONE>)
|    |    |    +----+ (select one)
|    |    |    |    |
|    |    |    |    | S_CEFSTS<IREAD>
|    |    |    |    +-----------------------------> I-stream NDAL read data error (Section 15.5.2.6.5)
|    |    |    |    |
|    |    |    |    | S_CEFSTS<OREAD>
|    |    |    |    +-----------------------------> D-stream NDAL ownership read data error
|    |    |    |    |                               (modify operand or read-lock) (Section 15.5.2.6.5)
|    |    |    |    | otherwise
|    |    |    |    +-----------------------------> D-stream NDAL read data error (read only operand)
|    |    |    |                                    (Section 15.5.2.6.5)
|    |    |    | otherwise
|    |    |    +----------------------------------> Not a synchronous machine check cause (see soft and
|    |    |                                         hard error interrupt events)
|    |    | S_CEFSTS<UNEXPECTED_FILL>
|    |    +---------------------------------------> Not a synchronous machine check cause (see soft error
|    |    |                                         interrupt events)
|    |    | otherwise
|    |    +---------------------------------------> Inconsistent status (either CEFSTS<RDE>, CEFSTS<TIMEOUT>,
|    |                                              or CEFSTS<UNEXPECTED_FILL> should be set)
|    |                                              (Section 15.5.2.7)
|    | S_CEFSTS<LOST_ERR> AND
|    | NOT S_PCSTS<PTE_ER>
|    +--------------------------------------------> Lost Bcache fill error (Section 15.5.2.6.6)
|    |
|    | S_NESTS<NOACK> AND
|    | NOT S_PCSTS<PTE_ER>
|    +----+
|    |    |
|    |    | S_NEOCMD<CMD>=IREAD
|    |    +---------------------------------------> Unacknowledged I-stream NDAL read (Section 15.5.2.6.7)
|    |    |
|    |    | S_NEOCMD<CMD>=DREAD
|    |    +---------------------------------------> Unacknowledged D-stream NDAL read (read only operand)
|    |    |                                         (Section 15.5.2.6.7)
|    |    | S_NEOCMD<CMD>=OREAD
|    |    +---------------------------------------> Unacknowledged D-stream NDAL read (modify operand or reac
|    |    |                                         (Section 15.5.2.6.7)
|    |    | S_NEOCMD<CMD>=WRITE OR WDISOWN
|    |    +---------------------------------------> Not a synchronous machine check cause (see hard error
|    |    |                                         interrupt events)
|    |    | otherwise
|    |    +---------------------------------------> Inconsistent status (invalid command in NEOCMD<CMD>)
|    |                                              (Section 15.5.2.7)
|    | S_NESTS<LOST_OERR> AND
|    | NOT S_PCSTS<PTE_ER>
|    +--------------------------------------------> Lost unrecoverable NDAL output error (Section 15.5.2.6.8)
|    |
v    v
1    2
```

Figure 15–5 Cont'd on next page

Figure 15–5 (Cont.): Cause Parse Tree for Machine Check Exceptions

```
1    2
v    v
|    |  S_BCEDSTS<LOCK> AND
|    |  S_PCSTS<PTE_ER>[1]
|    +----+ (select one)
|    |    |
|    |    |  S_BCEDSTS<BAD_ADDR>
|    |    +----+ (select one)
|    |    |    |
|    |    |    |  S_BCEDSTS<DR_CMD>=DREAD
|    |    |    +----------------------------------> Bcache data RAM addressing error on PTE read
|    |    |    |                                    (Section 15.5.2.6.9.2)
|    |    |    |  S_BCEDSTS<DR_CMD>=IREAD
|    |    |    +----+ (select one)
|    |    |    |    |
|    |    |    |    |  S_BCEDSTS<LOST_ERR>
|    |    |    |    +-----------------------------> Multiple errors in context of PTE read error
|    |    |    |    |                               (Section 15.5.2.6.9.6)
|    |    |    |    |  otherwise
|    |    |    |    +-----------------------------> Bcache data RAM error addressing error on I-stream read
|    |    |    |                                    (Section 15.5.2.6.2)
|    |    |    |  otherwise
|    |    |    +----+ (select one)
|    |    |         |
|    |    |         |  S_BCEDSTS<LOST_ERR>
|    |    |         +-----------------------------> Multiple errors in context of PTE read error
|    |    |         |                               (Section 15.5.2.6.9.6)
|    |    |         |  otherwise
|    |    |         +-----------------------------> Not a synchronous machine check cause (see soft and
|    |    |                                         hard error interrupt events)
|    |    |  S_BCEDSTS<UNCORR>
|    |    +----+ (select one)
|    |    |    |
|    |    |    |  S_BCEDSTS<DR_CMD>=DREAD
|    |    |    +----------------------------------> Bcache data RAM uncorrectable ECC error on PTE read
|    |    |    |                                    (Section 15.5.2.6.9.2)
|    |    |    |  S_BCEDSTS<DR_CMD>=IREAD
|    |    |    +----+ (select one)
|    |    |    |    |
|    |    |    |    |  S_BCEDSTS<LOST_ERR>
|    |    |    |    +-----------------------------> Multiple errors in context of PTE read error
|    |    |    |    |                               (Section 15.5.2.6.9.6)
|    |    |    |    |  otherwise
|    |    |    |    +-----------------------------> Bcache data RAM error uncorrectable error on I-stream read
|    |    |    |                                    (Section 15.5.2.6.2)
|    |    |    |  otherwise
|    |    |    +----+ (select one)
|    |    |         |
|    |    |         |  S_BCEDSTS<LOST_ERR>
|    |    |         +-----------------------------> Multiple errors in context of PTE read error
|    |    |         |                               (Section 15.5.2.6.9.6)
|    |    |         |  otherwise
|    |    |         +-----------------------------> Not a synchronous machine check cause (see soft and
|    |    |                                         hard error interrupt events)
|    |    |  none of the above
|    |    +---------------------------------------> Inconsistent status (no BCEDSTS unrecoverable error bits
|    |                                              set) (Section 15.5.2.7)
v    v
1    2
```

Figure 15–5 Cont'd on next page

[1] At least one potential PTE cause must be found or the status is inconsistent (see Section 15.5.2.7). Some of the outcomes indicate a potential synchronous machine check cause which is not a potential PTE read error cause. These errors should be treated separately.

Figure 15–5 (Cont.): Cause Parse Tree for Machine Check Exceptions

```
1    2
v    v
|    | S_CEFSTS<LOCK> AND
|    | S_PCSTS<PTE_ER>[1]
|    +----+ (select one)
|    |    |
|    |    | S_CEFSTS<TIMEOUT>
|    |    +----+ (select one)
|    |    |    |
|    |    |    | S_CEFSTS<TO_MBOX> AND
|    |    |    | (NOT S_CEFSTS<REQ_FILL_DONE>)
|    |    |    +----+ (select one)
|    |    |    |    |
|    |    |    |    | S_CEFSTS<IREAD>
|    |    |    |    +----+ (select one)
|    |    |    |    |    |
|    |    |    |    |    | S_CEFSTS<LOST_ERR>
|    |    |    |    |    +------------------------> Multiple errors in context of PTE read error
|    |    |    |    |    |                          (Section 15.5.2.6.9.6)
|    |    |    |    |    | otherwise
|    |    |    |    |    +------------------------> I-stream NDAL read timeout error (Section 15.5.2.6.4)
|    |    |    |    |
|    |    |    |    | S_CEFSTS<OREAD>
|    |    |    |    +----+ (select one)
|    |    |    |    |    |
|    |    |    |    |    | S_CEFSTS<LOST_ERR>
|    |    |    |    |    +------------------------> Multiple errors in context of PTE read error
|    |    |    |    |    |                          (Section 15.5.2.6.9.6)
|    |    |    |    |    | otherwise
|    |    |    |    |    +------------------------> D-stream NDAL ownership read timeout error
|    |    |    |    |                               (Section 15.5.2.6.4)
|    |    |    |    | otherwise
|    |    |    |    +-----------------------------> D-stream NDAL read timeout error (PTE read)
|    |    |    |                                    (Section 15.5.2.6.9.3)
|    |    |    | otherwise
|    |    |    +----+  (select one)
|    |    |         |
|    |    |         | S_CEFSTS<LOST_ERR>
|    |    |         +-----------------------------> Multiple errors in context of PTE read error
|    |    |         |                               (Section 15.5.2.6.9.6)
|    |    |         | otherwise
|    |    |         +-----------------------------> Not a synchronous machine check cause (see soft and
|    |    |                                         hard error interrupt events)
v    v    v
1    2    3
```

Figure 15–5 Cont'd on next page

[1] At least one potential PTE cause must be found or the status is inconsistent (see Section 15.5.2.7). Some of the outcomes indicate a potential synchronous machine check cause which is not a potential PTE read error cause. These errors should be treated separately.

Figure 15–5 (Cont.): Cause Parse Tree for Machine Check Exceptions

```
1    2    3
v    v    v
|    |    |  S_CEFSTS<RDE>
|    |    +----+ (select one)
|    |    |    |
|    |    |    |  S_CEFSTS<TO_MBOX> AND
|    |    |    |  (NOT S_CEFSTS<REQ_FILL_DONE>)
|    |    |    +----+ (select one)
|    |    |    |    |
|    |    |    |    |  S_CEFSTS<IREAD>
|    |    |    |    +----+ (select one)
|    |    |    |    |    |
|    |    |    |    |    |  S_CEFSTS<LOST_ERR>
|    |    |    |    |    +------------------------> Multiple errors in context of PTE read error
|    |    |    |    |    |                          (Section 15.5.2.6.9.6)
|    |    |    |    |    |  otherwise
|    |    |    |    |    +------------------------> I-stream NDAL read data error (Section 15.5.2.6.5)
|    |    |    |    |
|    |    |    |    |  S_CEFSTS<OREAD>
|    |    |    |    +----+ (select one)
|    |    |    |    |    |
|    |    |    |    |    |  S_CEFSTS<LOST_ERR>
|    |    |    |    |    +------------------------> Multiple errors in context of PTE read error
|    |    |    |    |    |                          (Section 15.5.2.6.9.6)
|    |    |    |    |    |  otherwise
|    |    |    |    |    +------------------------> D-stream NDAL ownership read data error
|    |    |    |    |                               (Section 15.5.2.6.5)
|    |    |    |    |  otherwise
|    |    |    |    +-----------------------------> D-stream NDAL read timeout error (PTE read)
|    |    |    |                                    (Section 15.5.2.6.9.4)
|    |    |    |  otherwise
|    |    |    +----+  (select one)
|    |    |         |
|    |    |         |  S_CEFSTS<LOST_ERR>
|    |    |         +-----------------------------> Multiple errors in context of PTE read error
|    |    |         |                               (Section 15.5.2.6.9.6)
|    |    |         |  otherwise
|    |    |         +-----------------------------> Not a synchronous machine check cause (see soft and
|    |    |                                         hard error interrupt events)
|    |    |  S_CEFSTS<UNEXPECTED_FILL>
|    |    +----+ (select one)
|    |    |    |
|    |    |    |  S_CEFSTS<LOST_ERR>
|    |    |    +----------------------------------> Multiple errors in context of PTE read error
|    |    |    |                                    (Section 15.5.2.6.9.6)
|    |    |    |  otherwise
|    |    |    +----------------------------------> Not a synchronous machine check cause (see hard error
|    |    |                                         interrupt events)
|    |    |  otherwise
|    |    +---------------------------------------> Inconsistent status (either CEFSTS<RDE>, CEFSTS<TIMEOUT>,
|    |                                              or CEFSTS<UNEXPECTED_FILL> should be set)
|    |                                              (Section 15.5.2.7)
v    v
1    2
```

Figure 15–5 Cont'd on next page

[1] At least one potential PTE cause must be found or the status is inconsistent (see Section 15.5.2.7). Some of the outcomes indicate a potential synchronous machine check cause which is not a potential PTE read error cause. These errors should be treated separately.

Figure 15–5 (Cont.): Cause Parse Tree for Machine Check Exceptions

```
1    2
v    v
|    |  S_NESTS<NOACK> AND
|    |  S_PCSTS<PTE_ER>[1]
|    +----+
|    |    |
|    |    |  S_NEOCMD<CMD>=IREAD
|    |    +----+ (select one)
|    |    |    |
|    |    |    |  S_NESTS<LOST_OERR>
|    |    |    +-----------------------------------> Multiple errors in context of PTE read error
|    |    |    |                                     (Section 15.5.2.6.9.6)
|    |    |    |  otherwise
|    |    |    +-----------------------------------> Unacknowledged I-stream NDAL read (Section 15.5.2.6.7)
|    |    |
|    |    |  S_NEOCMD<CMD>=DREAD
|    |    +----------------------------------------> Unacknowledged D-stream NDAL read (PTE read)
|    |    |                                          (Section 15.5.2.6.9.5)
|    |    |  S_NEOCMD<CMD>=OREAD
|    |    +----+ (select one)
|    |    |    |
|    |    |    |  S_NESTS<LOST_OERR>
|    |    |    +-----------------------------------> Multiple errors in context of PTE read error
|    |    |    |                                     (Section 15.5.2.6.9.6)
|    |    |    |  otherwise
|    |    |    +-----------------------------------> Unacknowledged D-stream NDAL read (modify operand or rea
|    |    |                                          (Section 15.5.2.6.7)
|    |    |  S_NEOCMD<CMD>=WRITE OR WDISOWN
|    |    +----+ (select one)
|    |    |    |
|    |    |    |  S_NESTS<LOST_OERR>
|    |    |    +-----------------------------------> Multiple errors in context of PTE read error
|    |    |    |                                     (Section 15.5.2.6.9.6)
|    |    |    |  otherwise
|    |    |    +-----------------------------------> Not a synchronous machine check cause (see hard error
|    |    |                                          interrupt events)
|    |    |  otherwise
|    |    +----------------------------------------> Inconsistent status (invalid command in NEOCMD<CMD>)
|    |                                               (Section 15.5.2.7)
|    |  none of the above
|    +---------------------------------------------> Inconsistent status (no cause found for synchronous mach
|                                                    (Section 15.5.2.7)
|  otherwise
+--------------------------------------------------> Inconsistent status (unknown machine check code)
                                                     (Section 15.5.2.7)

Notation:
    (select one)               - Exactly one case must be true. If zero or more than one is
                                 true, the status is inconsistent.
    (select all)               - More than one case may be true.
    (select all, at least one) - All the cases are possible causes of a particular machine check.
                                 More than one may be true. At least one must be true or the status
                                 is inconsistent.  A case is not considered true if it evaluates to
                                 "Not a machine check cause".
    otherwise                  - fall-through case for (select one) if no other case is true.
    none of the above          - fall-through case for (select all) or (select all, at least one)
                                 if no other case is true.
```

NOTE

References to VR and PSL<FPD> in the "retry condition" parts of the following descriptions of machine check causes should be understood to refer to the named bit in the machine check stack frame.

[1] At least one potential PTE cause must be found or the status is inconsistent (see Section 15.5.2.7). Some of the outcomes indicate a potential synchronous machine check cause which is not a potential PTE read error cause. These errors should be treated separately.

15.5.2.1 MCHK_UNKNOWN_MSTATUS

Description: An unknown memory management status was returned from the Mbox in response to a microcode memory management probe. This is probably due to an internal error in the Mbox, Ebox, or microsequencer.

Recovery procedures: No explicit error recovery is required in response to this error.

Retry condition: This error can only happen in microcode processing of memory management faults for a virtual memory reference. Retry if:

(VR = 1) OR (PSL<FPD> = 1).

15.5.2.2 MCHK_INT.ID_VALUE

Description: An illegal interrupt ID was returned in INT.SYS during interrupt processing in microcode. This is probably due to an internal error in the interrupt hardware, Ebox, or microsequencer.

Recovery procedures: No explicit error recovery is required in response to this error.

Retry condition: This error can only happen in microcode processing of interrupts which occurs between instructions or the middle of interruptable instructions. Retry if:

(VR = 1) OR (PSL<FPD> = 1).

15.5.2.3 MCHK_CANT_GET_HERE

Description: Microcode execution reached a presumably impossible address. This is probably due to a microcode bug or an internal error in the Ebox or microsequencer.

Recovery procedures: No explicit error recovery is required in response to this error.

Retry condition: Retry if:

(VR = 1) OR (PSL<FPD> = 1).

15.5.2.4 MCHK_MOVC.STATUS

Description: During the execution of MOVCx, the two state bits that encode the state of the move (forward, backward, fill) were found set to the fourth (illegal) combination. This is probably due to an internal error in the Ebox or microsequencer.

Recovery procedures: No explicit error recovery is required in response to this error.

Retry condition: Because the state bits encode the operation, the instruction can not be restarted in the middle of the MOVCx. If software can determine that no specifiers have been over-written (MOVCx destroys R0-R5 and memory due to string writes), the instruction may be restarted from the beginning by clearing PSL<FPD>. This should be done only if the source and destination strings do not overlap and if:

(PSL<FPD> = 1).

15.5.2.5 MCHK_ASYNC_ERROR

This machine check code reports serious errors which interrupt the microcode at an arbitrary point. Many internal machine states (e.g., bits in the PSL, the PC or SP) are questionable. Recovery is typically not possible.

15.5.2.5.1 TB Parity Errors

Description: Parity errors in tags and PTE data in the TB cause an asynchronous machine check by directly forcing a microtrap in the microsequencer. The reference being processed by the Mbox may be for and explicit Ebox reference, an operand prefetch or DEST_ADDR reference from the specifier queue, or an instruction prefetch from the IREF latch. Also the reference could be a read generated by the Mbox within a TB miss for a process space virtual address since process page tables are stored in virtual memory (system space).

Description (TB PTE Data Parity Error): A parity error in the PTE data portion of a TB entry which hit had a parity error.

Description (TB Tag Parity Error): A parity error in the tag portion of a TB entry which hit had a parity error.

Recovery procedures: To recover, clear TBSTS<LOCK>.

Retry condition: Since the Ibox is nearly always able to issue instruction prefetches, TB parity errors could occur at practically any time. This makes it impossible to determine what machine state is incorrect. There is no guarantee that all writes with a different PSL<CUR_MOD> completed successfully. Therefore even the stack frame PSL<CUR_MOD> can't be used to determine whether system data is uncorrupted.

So retry is not possible. Crash the system.

15.5.2.5.2 Ebox S3 Stall Timeout Error

Description: S3 stall timeout errors occur when the Ebox microcode is stalled waiting for some result or action which will probably never occur. S4 stalls in the Ebox cause S3 stalls and therefore can lead to S3 stall timeout. Additionally, field queue stall and instruction queue stall can cause this timeout. (These last two situations are not Ebox pipeline stalls, but they are similar in effect.) The timeout can occur in any microflow for a number of reasons. Machine state may be corrupted. This timeout is probably due to an internal error in the NVAX CPU such that one box is waiting for another to do something which it isn't going to do. An example would be if the Ebox microcode expected one more source specifier than the Ibox delivered. The Ebox will stall until the timeout occurs waiting for the Ibox to deliver one more source operand via the source queue.

S3 timeout errors can be caused by failures of various pipeline control circuits in the Ebox. Also a deadlock within a box or across multiple boxes can cause this error.

Recovery procedures: To recover, clear the S3_STALL_TIMEOUT bit in ECR.

Retry condition: Because this error can occur at any time, it is not possible to determine what machine state is incorrect. Also, this error should never happen and indicates either a serious failure in the NVAX CPU chip or a design bug. So retry is not possible. Crash the system.

15.5.2.6 MCHK_SYNC_ERROR

This machine check code reports errors which occur in memory or IO space instruction fetches or data reads. Except in the case of PTE read errors, core machine state should be consistent since microcode has to explicitly access an operand or instruction in order incur this error. Microcode does not access memory results or dispatch for a new instruction execution with core machine state in an inconsistent state.

PTE read errors on write transactions can cause a microtrap at an arbitrary time, and so core machine state may be inconsistent.

Many of the error events described below for synchronous machine check are possible causes. If more than one is present, there is no way to determine which actually caused the machine check. If exactly one possible cause is discovered, then the machine check may be attributed to that cause. The reason multiple causes may be present is that the NVAX CPU prefetches instructions and data. If the CPU branches or takes an exception before using data it has requested, then the pending machine check is taken as a soft error interrupt (though it might not be recoverable in the final analysis).

If multiple errors occur, recovery and retry may be possible. It is recommended that retry from multiple errors be done only if one error report does not interfere with analysis of, and recovery from, another error.

An example of such interference is when S_BCEDSTS reports a Bcache data RAM uncorrectable error on a writeback while S_NESTS is reporting a NDAL command no-ACK error. Normally, S_NESTS<BADWDATA> would be reported by the writeback error and S_NEOADR would report the address of the lost writeback. The no-ACK error makes recovery from the writeback error much more difficult. But there it is unlikely that these two errors would occur together since they are understood to be uncorrelated events. So this case is considered unrecoverable.

If two errors are entirely separate, neither interfering with the analysis and recovery of the other, then it is acceptable to retry from these errors provided all the error analyses and recovery procedures result in a retry indication.

In several cases, lost errors are tolerated. See Section 15.3.4.2 for a list of these special cases. In each case, the strong tendency to prefetch data exhibited by the NVAX pipeline makes the particular lost error likely, given that one error of that kind occurred. Also, in each case, if data is lost in the lost error, a hard error interrupt is posted. So these errors are tolerated as long as they do not cause a hard error interrupt.

Errors in opcode or operand specifier fetching are always detected before architecturally visible state within the CPU is modified. This means the VR bit from the machine check stack frame should be 1. This error handling analysis attempts to recover from multiple errors, so the retry condition for each error is made as general as possible. If the machine check handler finds only errors of the kind listed here, then VR should be 1 and it is an inconsistent report if it is not (see Section 15.5.2.7).

- VIC parity errors.
- Bcache data RAM uncorrectable ECC and addressing errors in I-stream reads.
- Bcache timeout errors and fill read data errors in I-stream reads.
- Unacknowledged NDAL I-stream reads

15.5.2.6.1 VIC Parity Errors

Description: A parity error was detected in the VIC tag or data store in the Ibox. VIC parity errors cause a machine check when the Ebox microcode requests dispatch to a new instruction execution microflow or attempts to access an operand within an instruction execution microflow.

VIC Data Parity Errors: A parity error occurred in the data portion of the VIC.

VIC Tag Parity Errors: A parity error occurred in the tag portion of the VIC.

In all cases, the quadword virtual address of the error is in VMAR.

Pending Interrupts: A soft error interrupt should be pending.

Recovery procedures: To recover, disable and flush the VIC by re-writing all the tags (using the procedure in Section 15.3.3.1.1.1). Also, clear ICSR<LOCK>.

Retry condition: Retry if:

(VR = 1) OR (PSL<FPD> = 1).

15.5.2.6.2 Bcache Data RAM Uncorrectable ECC Errors and Addressing Errors

Description (addressing errors): A Bcache addressing error was detected by the Cbox in an I-stream or D-stream read during a Bcache hit. Addressing errors are the result of a mismatch between the address the Cbox drives to the RAMs for a read access and the address used to write that location. A multiple bit data error can appear to be addressing error, though it is extremely unlikely.

Description (uncorrectable ECC errors): A Bcache uncorrectable data error was detected by the Cbox in an I-stream or D-stream read during a Bcache hit. Uncorrectable data errors are the result of a multiple bit error in the data read from the Bcache. An addressing error with a single bit data error will appear as an uncorrectable data error.

Description (all cases): The Bcache is in ETM. S_BCEDIDX contains the cache index of the error, and S_BCEDECC contains the syndrome calculated by the ECC logic.

The physical address of the reference can be found by reading the tag for the data block (using the procedure in Section 15.3.3.1.2.4). (If the physical address is found to be in IO space, it is an inconsistent status. See Section 15.5.2.7.) If the block's tag is found to contain an uncorrectable ECC error, then the address can not be determined.

It should never be the case that both S_BCEDSTS<BAD_ADDR> and S_BCEDSTS<UNCORR> are set. If they are, it is an inconsistent status (see Section 15.5.2.7).

Pending Interrupts: A soft error interrupt should be pending.

Recovery procedures (addressing errors): To recover, clear BCEDSTS<LOCK, BAD_ADDR>.

Recovery procedures (uncorrectable ECC errors): To recover, clear BCEDSTS<LOCK, UNCORR>.

Recovery procedures (both cases): Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache). If the data is owned by the Bcache and if the error repeats itself (is not transient), then a writeback error will result from the flush procedure. Software should prepare for this by clearing NESTS and BCEDSTS errors.

Retry condition: If no writeback error occurs in the Bcache flush, retry if:

(VR = 1) OR (PSL<FPD> = 1).

If a writeback error occurs in the Bcache flush, then the data is presumed to be unrecoverable. See Section 15.8.1.10 for a description of handling an error in a writeback. Given that the address is available (no error in the tag store), software should determine if the error is fatal to one process or the whole system and take appropriate action. Otherwise, crash the system.

15.5.2.6.3 Bcache Lost Data RAM Access Error

Description: A lost Bcache data RAM error may have been a machine check cause. It also might not have been. Lost Bcache data RAM errors which cause machine checks are always read errors, and can be retried unless the aborted instruction has altered essential state. Whether or not it is a machine check cause, the error will have caused either a soft or hard error interrupt. Lost Bcache data RAM errors which can not have caused a machine check are dealt with in the sections on hard and soft error interrupts.

Lost Bcache data RAM errors may be caused by more than one operand prefetch to the same cache block.

Recovery for lost Bcache data RAM errors depends on whether the pending interrupt is a hard or soft error interrupt. The machine check error handling software should defer recovery until the expected hard or soft error interrupt occurs. Once the interrupt is taken, the error recovery and restart instructions found in the hard error interrupt and soft error interrupt sections should be referenced. See Section 15.7.1.3.2 and Section 15.8.1.15.

Software should employ some mechanism to record that an interrupt for a lost Bcache data RAM error is pending. This mechanism should allow detection of a case in which an expected interrupt does not occur (once IPL is lowered). If the expected interrupt does not occur when IPL is lowered, then a serious inconsistency exists and the system should be crashed.

The Bcache in in ETM.

Pending Interrupts: A hard or soft error interrupt should be pending, or possibly both.

Recovery procedures: No specific recovery action is required. Note that BCEDSTS<LOST_ERR> is not cleared. It will be cleared by the hard or soft error interrupt handler. Also, the Bcache must remain in ETM until the error interrupt occurs.

Retry condition: Retry only if:

(VR = 1) OR (PSL<FPD> = 1).

15.5.2.6.4 NDAL I-Stream or D-Stream Read or D-Stream Ownership Read Timeout Errors

Description: An I-stream or D-stream read or D-stream ownership read timed out before any fill quadword was received. This is not an accepted means for a system environment to notify the NVAX CPU of "non-existent memory or IO location". The error could be caused by an error in the system environment or an NDAL parity error on the returned data. It also could be caused by some previous error in the system environment or this CPU which leaves a cache block marked as owned in memory and not marked as owned in any cache in the system.

S_CEFSTS<COUNT> indicates the number of quadwords received before the error. (S_CEFSTS<COUNT> should always be 11 (binary) if the address is in IO space.) The physical address is in S_CEFADR.

CEFSTS<WRITE> should not be set. If it is, it is an inconsistent status (see Section 15.5.2.7).

I-stream read: The Bcache is not in ETM.

I-stream errors cause a machine check when the Ebox microcode requests dispatch to a new instruction execution microflow or attempts to access an operand within an instruction execution microflow where the I-stream data with the error is required for the dispatch or access.

D-stream read: The Bcache is not in ETM.

D-stream read errors cause a machine check when the Ebox microcode accesses prefetched operand data or when the Mbox returns data tagged with an error indication to the Ebox register file.

D-stream ownership read: The Bcache is in ETM. No write data has been merged with the returning fills.

The address should not be in IO space. If it is, it is an inconsistent status (see Section 15.5.2.7).

D-stream ownership read errors cause a machine check when the Ebox microcode accesses prefetched operand data or when the Ebox issues a read-lock.

Pending Interrupts (all cases): A soft error interrupt should be pending.

Recovery procedures (all cases): Clear CEFSTS<LOCK,TIMEOUT>.

Additional Recovery procedures for D-stream ownership read: Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache).

Depending on the system environment, memory may have set its ownership bit for this block. The data in memory is presumably still good. The Bcache block is marked invalid in the Bcache tag store.

If S_CEFSTS<COUNT> is greater than 0, then part of the data also is in the Bcache. In general, it is not possible to determine which quadwords are valid. However, if the S_CEFSTS<COUNT> is 11 (binary) and S_CEFSTS<REQ_FILL_DONE> is not set, then the three quadwords in the Bcache block other than the quadword pointed to by S_CEFADR are valid.

If S_CEFSTS<COUNT> is greater than 0, and the address in S_CEFADR is not in IO space, then the block was not owned before the operation began. In this case, use the procedures in Section 15.3.3.1.2.2 to determine if memory's ownership bit is set and this CPU owns the block. If so, use the system specific procedure (see Section 15.3.3.1.2.2.2) to reset it. In some systems (the XMI2 for example) this may require a quadword of correct data be written to memory to reset the ownership bit. Section 15.3.3.1.2.3 describes procedures for extracting data from the Bcache data RAMs in this case.

If memory's ownership bit was left set as a result of this error and no non-destructive procedure exists for restoring it, then the hexaword block is lost.

Retry condition (I-stream or D-stream read): Retry if the address is not in IO space and:

(VR = 1) OR (PSL<FPD> = 1).

Retry condition (D-stream ownership read): Given that no data is lost, retry if the memory state repair procedure is successful or not called for and if:

(VR = 1) OR (PSL<FPD> = 1).

If the hexaword block could not be repaired or data is lost, software must determine if the error is fatal to one process or the whole system and take appropriate action. (If it is fatal only to one process, use the system dependent procedure for reseting memory's ownership bit.)

Post Retry Recovery: If the same fill error recurs on retry, then the block is probably "lost".[1] Software must determine if the error is fatal to one process or the whole system and take appropriate action. (If it is fatal only to one process, use the system dependent procedure for reseting memory's ownership bit.)

NOTE

> It may be appropriate in this case to first cause each CPU in the system to flush its Bcache, and then retry once more.

NOTE

> It may be that another error (such as an uncorrectable tag store error on a coherence request) will be repaired by the soft error interrupt handler before the retry actually occurs, fortuitously repairing the cause of the fill error.

15.5.2.6.5 NDAL I-Stream or D-Stream Read or D-Stream Ownership Read Data Errors

Description: An I-stream or D-stream read or D-stream ownership read ended with an RDE (read data error) NDAL cycle before any the fill quadwords were received. If S_CEFSTS<COUNT> is 0 or the address in S_CEFADR is an IO space address, this is an accepted means for a system environment to notify the NVAX CPU of "non-existent memory or IO location". Otherwise, the error could be caused by an error in the system environment. It also could be caused by some previous error in the system environment or this CPU which leaves a cache block marked as owned in memory and not marked as owned in any cache in the system.

S_CEFSTS<COUNT> indicates the number of quadwords received before the error. (S_CEFSTS<COUNT> should always be 11 (binary) if the address is in IO space.) The physical address is in S_CEFADR.

CEFSTS<WRITE> should not be set. If it is, it is an inconsistent status (see Section 15.5.2.7).

I-stream read: The Bcache is not in ETM.

I-stream errors cause a machine check when the Ebox microcode requests dispatch to a new instruction execution microflow or attempts to access an operand within an instruction execution microflow where the I-stream data with the error is required for the dispatch or access.

D-stream read: The Bcache is not in ETM.

D-stream read errors cause a machine check when the Ebox microcode accesses prefetched operand data or when the Mbox returns data tagged with an error indication to the Ebox register file.

[1] In this case the more general sense of "lost" is implied. That is, memory's ownership bit is set but no cache writes the data back when a read is done to that location. In some systems, it may be possible to identify which CPU memory "thinks" owns the data, but it is often not possible to determine which error caused this situation to arise.

D-stream ownership read: The Bcache is in ETM. No write data has been merged with the returning fills.

The address should not be in IO space. If it is, it is an inconsistent status (see Section 15.5.2.7).

D-stream ownership read errors cause a machine check when the Ebox microcode accesses prefetched operand data or when the Ebox issues a read-lock.

Pending Interrupts (all cases): A soft error interrupt should be pending.

Recovery procedures (all cases): Clear CEFSTS<LOCK,RDE>.

Additional Recovery procedures for D-stream ownership read: Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache).

Depending on the system environment, memory may have set its ownership bit for this block. The data in memory could still be good. The Bcache block is marked invalid in the Bcache tag store.

If S_CEFSTS<COUNT> is greater than 0, then part of the data also is in the Bcache. In general, it is not possible to determine which quadwords are valid. However, if the S_CEFSTS<COUNT> is 11 (binary) and S_CEFSTS<REQ_FILL_DONE> is not set, then the three quadwords in the Bcache block other than the quadword pointed to by S_CEFADR are valid.

If S_CEFSTS<COUNT> is greater than 0, and the address in S_CEFADR is not in IO space, then the block was not owned before the operation began. In this case, use the procedures in Section 15.3.3.1.2.2 to determine if memory's ownership bit is set and this CPU owns the block. If so, use the system specific procedure (see Section 15.3.3.1.2.2.2) to reset it. In some systems (the XMI2 for example) this may require a quadword of correct data be written to memory to reset the ownership bit. Section 15.3.3.1.2.3 describes procedures for extracting data from the Bcache data RAMs in this case.

If memory's ownership bit was left set as a result of this error and no non-destructive procedure exists for restoring it, then the hexaword block is lost.

Retry condition (I-stream or D-stream read): Retry if the address is not in IO space and:

(VR = 1) OR (PSL<FPD> = 1).

Retry condition (D-stream ownership read): Given that no data is lost, retry if the memory state repair procedure is successful or not called for and if:

(VR = 1) OR (PSL<FPD> = 1).

If the hexaword block could not be repaired or data is lost, software must determine if the error is fatal to one process or the whole system and take appropriate action. (If it is fatal only to one process, use the system dependent procedure for reseting memory's ownership bit.)

Post Retry Recovery: If the same fill error recurs on retry, then the block is probably "lost".[1] Software must determine if the error is fatal to one process or the whole system and take appropriate action. (If it is fatal only to one process, use the system dependent procedure for reseting memory's ownership bit.)

[1] In this case the more general sense of "lost" is implied. That is, memory's ownership bit is set but no cache writes the data back when a read is done to that location. In some systems, it may be possible to identify which CPU memory "thinks" owns the data, but it is often not possible to determine which error caused this situation to arise.

NOTE

It may be appropriate in this case to first cause each CPU in the system to flush its Bcache, and then retry once more.

NOTE

It may be that another error (such as an uncorrectable tag store error on a coherence request) will be repaired by the soft error interrupt handler before the retry actually occurs, fortuitously repairing the cause of the fill error.

15.5.2.6.6 Lost Bcache Fill Error

Description: Some number of fill errors occurred and were not latched because CEFSTS and CEFADR already contained a report of an unrecoverable error. There is no guarantee this error could have caused a machine check, though it may be a cause. Lost Bcache fill errors which cause machine checks are always read errors, and can be retried unless the aborted instruction has altered essential state. If it is a machine check cause, the error will have caused a a soft error interrupt. Lost Bcache fill errors which can not have caused a machine check are dealt with in the sections on hard and soft error interrupts.

Lost Bcache fill errors may be caused by more than one operand prefetch to the same cache block.

Recovery for lost Bcache fill errors depends on whether the pending interrupt is a hard or soft error interrupt. The machine check error handling software should defer recovery until the expected hard or soft error interrupt occurs. Once the interrupt is taken, the error recovery and restart instructions found in the hard error interrupt and soft error interrupt sections should be referenced. See Section 15.7.1.3.2 and Section 15.8.1.15.

Software should employ some mechanism to record that an interrupt for a lost Bcache fill error is pending. This mechanism should allow detection of a case in which an expected interrupt does not occur (once IPL is lowered). If the expected interrupt does not occur when IPL is lowered, then a serious inconsistency exists and the system should be crashed.

The Bcache may be in ETM (S_CCTL<HW_ETM> will be set if it is).

Pending Interrupts: A hard or soft error interrupt should be pending, or possibly both.

Recovery procedures: No specific recovery action is required. Note that CEFSTS<LOST_ERR> is not cleared. It will be cleared by the hard or soft error interrupt handler. Also, the Bcache must remain in ETM (if it is already) until the error interrupt occurs.

Retry condition: Retry only if:

(VR = 1) OR (PSL<FPD> = 1).

15.5.2.6.7 Unacknowledged NDAL I-Stream or D-Stream Read or D-Stream Ownership Read

Description: An I-stream or D-stream read or D-stream ownership read was no-ACKed by the system environment. This could be because the external component(s) received bad NDAL parity or it could be due to a system-specific notification of "non-existent memory or IO location". The physical address is in S_NEOADR.

I-stream read: The Bcache is not in ETM.

I-stream errors cause a machine check when the Ebox microcode requests dispatch to a new instruction execution microflow or attempts to access an operand within an instruction execution microflow where the I-stream data with the error is required for the dispatch or access.

D-stream read: The Bcache is not in ETM.

D-stream read errors cause a machine check when the Ebox microcode accesses prefetched operand data or when the Mbox returns data tagged with an error indication to the Ebox register file.

D-stream ownership read: The Bcache is in ETM.

The address should not be in IO space. If it is, it is an inconsistent status (see Section 15.5.2.7).

D-stream ownership read errors cause a machine check when the Ebox microcode accesses prefetched operand data.

Pending Interrupts (all cases): A soft error interrupt should be pending.

Recovery procedures (all cases): Clear NESTS<NOACK>.

Additional Recovery procedure for D-stream ownership read: Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache).

Retry condition: Retry if:

(VR = 1) OR (PSL<FPD> = 1).

15.5.2.6.8 Lost NDAL Output Error

Description: Some number of NDAL output errors occurred and were not latched because NESTS, NEOADR, NEDATHI, and NEDATLO already contained a report of an unrecoverable error. There is no guarantee this error could have caused a machine check, though it may be a cause. Lost NDAL output errors which cause machine checks are always read errors, and can be retried unless the aborted instruction has altered essential state. If it is a machine check cause, the error will have caused a a soft error interrupt. Lost NDAL output errors which can not have caused a machine check are dealt with in the sections on hard and soft error interrupts.

Recovery for lost NDAL output errors depends on whether the pending interrupt is a hard or soft error interrupt. The machine check error handling software should defer recovery until the expected hard or soft error interrupt occurs. Once the interrupt is taken, the error recovery and restart instructions found in the hard error interrupt and soft error interrupt sections should be referenced. See Section 15.7.1.5 and Section 15.8.1.17.

Software should employ some mechanism to record that an interrupt for a lost NDAL output error is pending. This mechanism should allow detection of a case in which an expected interrupt does not occur (once IPL is lowered). If the expected interrupt does not occur once IPL is lowered, then a serious inconsistency exists and the system should be crashed.

The Bcache may be in ETM (S_CCTL<HW_ETM> will be set if it is).

Pending Interrupts: A hard or soft error interrupt should be pending, or possibly both.

Recovery procedures: No specific recovery action is required. Note that NESTS<LOST_ERR> is not cleared. It will be cleared by the hard or soft error interrupt handler. Also, the Bcache must remain in ETM (if it is already) until the error interrupt occurs.

Retry condition: Retry only if:

(VR = 1) OR (PSL<FPD> = 1).

15.5.2.6.9 PTE read errors

The following sections describe error handling for PTE read errors. PTE read errors are read errors which happen in reads issued by the Mbox in handling a TB miss. Handling of these errors is different from handling the same underlying error (Bcache data RAM error, Bcache fill error, or NDAL no-ACK error) when PTE read isn't the cause.

If S_PCSTS<PTE_ER> is set, then a PTE read issued by the Mbox in processing a TB miss had an unrecoverable error. The TB miss sequence was aborted because of the error. The original reference can be any I-stream or D-stream read or write. If the original reference was issued by the Ebox, then the PTE read which incurred the error will have been retried once (because of a special hardware/microcode mechanism for handling PTE read errors on Ebox references).

PTE read errors are difficult to analyze, partly because the read error report in the Cbox does not directly indicate that the failing read was a PTE read. Because of this and because PTE read errors should be rare (a very small percentage of the reads issued by the Mbox are PTE reads), multiple errors which interfere with the analysis of the PTE error are not considered recoverable.

The mechanism for reporting PTE read errors on Ebox references involves the Mbox forcing the Ebox (via a microtrap) into the microcode routine which normally handles memory management faults. This routine probes the address of the original reference, effectively retrying the failing PTE read. Assuming the error is not transient, the probe by microcode will cause a machine check. If the error does not occur on the probe, microcode restarts the current instruction stream. So machine checks caused by PTE read errors can easily occur with the particular PTE read error having occurred twice (with a lost error bit set in the relevant Cbox error register). The analysis here tolerates these particular multiple error reports and allows retry in those cases, provided the remainder of the error analysis indicates retry is appropriate. (Note that there is no way to tell from the information available to the machine check handler whether the original reference was an Ebox or Ibox reference.)

If the reference which incurs the PTE read error is a write, S_PCSTS<PTE_ER_WR> will be set. In this case the original write is lost. No retry is possible partly because the instruction which took the machine check may be subsequent to the one which issued the failing write. Also, PTE read errors on write transactions can cause a machine check at a practically arbitrary time in a microcode flow, and core machine state may not be consistent.

15.5.2.6.9.1 PTE Read Errors In Interruptable Instructions

Another special case associated with PTE read errors exists for interruptable instructions (specifically CMPC3, CMPC5, LOCC, MOVC3, MOVC5, SCANC, SKPC, and SPANC). For these instructions, if the PTE read error occurred for an Ebox reference, the PC in the machine check stack frame points to the instruction following the interrupted instruction. In this case, the SAVEPC element in the machine check stack frame is the PC of the interrupted instruction. However in all other cases, SAVEPC is UNPREDICTABLE. This case is not considered recoverable because analysis of the error information can not unambiguously conclude that this case is present. To tell that this case might be present, the error handler examines the FPD bit in the PSL in the machine check stack frame. If FPD is set in the stack frame (in the case of a PTE read error) then one of the following is true:

- One of the interruptable instructions listed above incurred the PTE read error. In this case, SAVEPC from the machine check stack frame points to the interrupted instruction, and PC in the stack frame points to the next instruction.
- An REI instruction loaded a PSL with FPD set and a certain PC. The Ibox incurred the PTE read error in fetching the opcode pointed to by that PC. In this case, the PC in the stack frame points to the instruction which was the target of the REI and SAVEPC from the stack frame is unpredictable.

It is not possible to determine with certainty which of the two above cases is the cause of a machine check with S_PCSTS<PTE_ER> set and stack frame PSL<FPD> set. Retry is not possible since software can not tell which PC to restart with. However, software may wish to probe the location pointed to by the PC in the stack frame, expecting a possible machine check as a result. If a machine check does occur, that is information indicating that the second case occurred (not totally unambiguously, of course). A very good guess may be made by a person examining the error report if the machine check stack frame and the result of this probe is available in the report.

15.5.2.6.9.2 Bcache Data RAM Uncorrectable ECC Errors and Addressing Errors on PTE Reads

Description (addressing errors): A Bcache addressing error was detected by the Cbox in a PTE read during a Bcache hit. Addressing errors are the result of a mismatch between the address the Cbox drives to the RAMs for a read access and the address used to write that location. A multiple bit data error can appear to be addressing error, though it is extremely unlikely.

Description (uncorrectable ECC errors): A Bcache uncorrectable data error was detected by the Cbox in a PTE read during a Bcache hit. Uncorrectable data errors are the result of a multiple bit error in the data read from the Bcache. An addressing error with a single bit data error will appear as an uncorrectable data error.

Description (all cases): The Bcache in in ETM. S_BCEDIDX contains the cache index of the error, and BCEDECC contains the syndrome calculated by the ECC logic. The physical address of the PTE read can be found by reading the tag for the data block (using the procedure in Section 15.3.3.1.2.4). (If the physical address is found to be in IO space, it is an inconsistent status. See Section 15.5.2.7.)

If the block's tag is found to contain an ECC error, then the address can not be determined.

S_BCEDSTS<LOST_ERR> may be set. This error is probably due to the same PTE error occurring more than once. This is an acceptable assumption unless a hard error interrupt occurs after handling this error.

It should never be the case that both S_BCEDSTS<BAD_ADDR> and S_BCEDSTS<UNCORR> are set. If they are, it is an inconsistent status (see Section 15.5.2.7).

Pending Interrupts: A soft error interrupt should be pending.

Recovery procedures (addressing errors): To recover, clear BCEDSTS<LOCK, BAD_ADDR>.

Recovery procedures (uncorrectable ECC errors): To recover, clear BCEDSTS<LOCK, UNCORR>.

Recovery procedures (both cases): Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache). Clear PCSTS<PTE_ER>. If the data is owned by the Bcache and if the error repeats itself (is not transient), then a writeback error will result from the flush procedure. Software should prepare for this by clearing NESTS and BCEDSTS errors.

Retry condition: If no writeback error occurs in the Bcache flush, retry if:

(VR = 1) AND (PSL<FPD> = 0) AND (S_PCSTS<PTE_ER_WR> = 0).

If

(PSL<FPD> = 1) OR (S_PCSTS<PTE_ER_WR> = 1),

crash the system.

If a writeback error occurs in the Bcache flush, then the data is presumed to be unrecoverable. See Section 15.8.1.10 for a description of handling an error in a writeback (software must determine if the error is fatal to one process or the whole system and take appropriate action).

15.5.2.6.9.3 NDAL PTE Read Timeout Errors

Description: A PTE read timed out before any fill quadword was received. This is not an accepted means for a system environment to notify the NVAX CPU of "non-existent memory or IO location". The error could be caused by an error in the system environment or an NDAL parity error on the returned data. It also could be caused by some previous error in the system environment or this CPU which leaves a cache block marked as owned in memory and not marked as owned in any cache in the system.

S_CEFSTS<COUNT> indicates the number of quadwords received before the error. (S_CEFSTS<COUNT> should always be 11 (binary) if the address is in IO space.) The physical address is in S_CEFADR.

CEFSTS<WRITE> should not be set. If it is, it is an inconsistent status (see Section 15.5.2.7).

The Bcache is not in ETM. The read was not an ownership read, so this error can not have caused the ownership bits in memory to be left in the wrong state.

S_CEFSTS<LOST_ERR> may be set. This error is probably due to the same PTE error occurring more than once. This is an acceptable assumption unless a hard error interrupt occurs after handling this error.

Pending Interrupts: A soft error interrupt should be pending.

Recovery procedures: Clear CEFSTS<LOCK, TIMEOUT>. Clear PCSTS<PTE_ER>.

Retry condition: Retry if:

(VR = 1) AND (PSL<FPD> = 0) AND (S_PCSTS<PTE_ER_WR> = 0).

Otherwise, crash the system.

Post Retry Recovery: If the same fill error recurs on retry, then the block is probably "lost".[1] Software must determine if the error is fatal to one process or the whole system and take appropriate action. (If it is fatal only to one process, use the system dependent procedure for reseting memory's ownership bit.)

[1] In this case the more general sense of "lost" is implied. That is, memory's ownership bit is set but no cache writes the data back when a read is done to that location. In some systems, it may be possible to identify which CPU memory "thinks" owns the data, but it is often not possible to determine which error caused this situation to arise.

NOTE

It may be appropriate in this case to first cause each CPU in the system to flush its Bcache, and then retry once more.

NOTE

It may be that another error (such as an uncorrectable tag store error on a coherence request) will be repaired by the soft error interrupt handler before the retry actually occurs, fortuitously repairing the cause of the fill error.

15.5.2.6.9.4 NDAL PTE Read Data Errors

Description: A PTE read ended with an RDE (read data error) NDAL cycle before any the fill quadwords were received. If S_CEFSTS<COUNT> is 0 or the address in S_CEFADR is an IO space address, this is an accepted means for a system environment to notify the NVAX CPU of "non-existent memory or IO location". Otherwise, the error could be caused by an error in the system environment. It also could be caused by some previous error in the system environment or this CPU which leaves a cache block marked as owned in memory and not marked as owned in any cache in the system.

S_CEFSTS<COUNT> indicates the number of quadwords received before the error. (S_CEFSTS<COUNT> should always be 11 (binary) if the address is in IO space.) The physical address is in S_CEFADR.

CEFSTS<WRITE> should not be set. If it is, it is an inconsistent status (see Section 15.5.2.7).

The physical address of the PTE is in S_CEFADR. The Bcache is not in ETM. The read could not have been an ownership read, so this error can not have caused the ownership bits in memory to be left in the wrong state.

S_CEFSTS<LOST_ERR> may be set. This error is probably due to the same PTE error occurring more than once. This is an acceptable assumption unless a hard error interrupt occurs after handling this error.

Pending Interrupts: A soft error interrupt should be pending.

Recovery procedures: Clear CEFSTS<LOCK, RDE>. Clear PCSTS<PTE_ER>.

Retry condition: Retry if:

(VR = 1) AND (PSL<FPD> = 0) AND (S_PCSTS<PTE_ER_WR> = 0).

Otherwise, crash the system.

Post Retry Recovery: If the same fill error recurs on retry, then the block is probably "lost".[1] Software must determine if the error is fatal to one process or the whole system and take appropriate action. (If it is fatal only to one process, use the system dependent procedure for reseting memory's ownership bit.)

[1] In this case the more general sense of "lost" is implied. That is, memory's ownership bit is set but no cache writes the data back when a read is done to that location. In some systems, it may be possible to identify which CPU memory "thinks" owns the data, but it is often not possible to determine which error caused this situation to arise.

NOTE

It may be appropriate in this case to first cause each CPU in the system to flush its Bcache, and then retry once more.

NOTE

It may be that another error (such as an uncorrectable tag store error on a coherence request) will be repaired by the soft error interrupt handler before the retry actually occurs, fortuitously repairing the cause of the fill error.

15.5.2.6.9.5 Unacknowledged NDAL PTE Read

Description: A PTE read was no-ACKed by the system environment. This could be because the external component(s) received bad NDAL parity or it could be due to a system-specific notification of "non-existent memory or IO location".

The physical address of the PTE is in S_NEOADR. The Bcache is not in ETM.

S_CEFSTS<LOST_OERR> may be set. This error is probably due to the same PTE error occurring more than once. This is an acceptable assumption unless a hard error interrupt occurs after handling this error.

Pending Interrupts: A soft error interrupt should be pending.

Recovery procedures: Clear NESTS<NOACK>. Clear PCSTS<PTE_ER>.

Retry condition: Retry if:

(VR = 1) AND (PSL<FPD> = 0) AND (S_PCSTS<PTE_ER_WR> = 0).

Otherwise, crash the system.

15.5.2.6.9.6 Multiple Errors Which Interfere with Analysis of PTE Read Error

Because PTE read errors lead to several unusual cases, retry is not recommended in the event that other errors cloud the analysis of the PTE read error.

Pending Interrupts: A hard or soft error interrupt should be pending, or possibly both.

Recovery procedures: No specific recovery action is called for.

Retry condition: No retry is possible. Crash the system.

15.5.2.7 Inconsistent Status in Machine Check Cause Analysis

Description: A presumed impossible error report was found in the error registers. This could be due to a hardware failure or bug, or to incomplete analysis in this spec.

Pending Interrupts: A hard or soft error interrupt should be pending, or possibly both.

Recovery procedures: No specific recovery action is called for.

Retry condition: No retry is possible. The integrity of the entire system is questionable. Crash the system.

15.6 Power Fail Interrupt

Power fail interrupts are requested to report imminent loss of power to the CPU. Power fail interrupts are requested via the PWRFL_L pin at IPL 1E (hex) and are dispatched to the operating system through SCB vector 0C (hex).

The stack frame for a power fail interrupt is shown in Figure 15–6.

Figure 15–6: Power Fail Interrupt Stack Frame

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                             PC                                                | :(SP)
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                             PSL                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

15.7 Hard Error Interrupts

Hard error interrupts are requested to report an error that was detected asynchronously with respect to instruction execution. This results in an interrupt at IPL 1D (hex) to be dispatched through SCB vector 60 (hex). Typically, these error indicate that machine state has been corrupted and that retry is not possible.

The stack frame for a hard error interrupt is shown in Figure 15–7.

Figure 15–7: Hard Error Interrupt Stack Frame

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              PC                                               | :(SP)
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              PSL                                              |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

15.7.1 Events Reported Via Hard Error Interrupts

This section describes all the errors which can cause a hard error interrupt. A parse tree is given which shows how to determine the cause of a given hard error. After that, there is a description of each error. For each error, the recovery procedure is given. Where appropriate, the conditions for restart are given. See Section 15.3.3 and Section 15.3.4 for more on error recovery and error retry.

Figure 15–8 is a parse tree which should be used to analyze the cause of a hard error interrupt. It is assumed that the state being analyzed is the saved state, as described in Section 15.3.1. Otherwise the state could change during the analysis procedure, leading to possibly incorrect conclusions. (See Section 15.3.2 for general information about error analysis.)

Figure 15–8: Cause Parse Tree for Hard Error Interrupts

```
HARD ERROR INTERRUPT
----+ (select all, at least one)
    |
    |  S_BCEDSTS<LOCK>
    +----+ (select one)
    |    |
    |    |  S_BCEDSTS<BAD_ADDR>
    |    +----+
    |    |    |
    |    |    |  S_BCEDSTS<DR_CMD>=RMW
    |    |    +---------------------------------------> Bcache data RAM addressing error on a write or write-unl
    |    |    |                                         from Mbox (Section 15.7.1.1)
    |    |    |  otherwise
    |    |    +---------------------------------------> Not a hard error interrupt cause (see soft error interru
    |    |                                              events)
    |    |  S_BCEDSTS<UNCORR>
    |    +----+
    |    |    |
    |    |    |  S_BCEDSTS<DR_CMD>=RMW
    |    |    +---------------------------------------> Bcache data RAM uncorrectable ECC error on a write or wri
    |    |    |                                         unlock from Mbox (Section 15.7.1.1)
    |    |    |  otherwise
    |    |    +---------------------------------------> Not a hard error interrupt cause (see soft error interru
    |    |                                              events)
    |    |  none of the above
    |    +--------------------------------------------> Inconsistent status (no BCEDSTS unrecoverable error bits
    |                                                   set) (Section 15.7.1.7)
    |  S_BCEDSTS<LOST_ERR>
    +-------------------------------------------------> Lost unrecoverable Bcache data RAM error
    |                                                   (Section 15.7.1.2)
    |  S_CEFSTS<LOCK>
    +----+ (select one)
    |    |
    |    |  S_CEFSTS<TIMEOUT> AND S_CEFSTS<REQ_FILL_DONE>
    |    |  AND S_CEFSTS<WRITE> AND S_CEFSTS<OREAD>
    |    +--------------------------------------------> NDAL timeout on OREAD for write from Mbox after write dat
    |    |                                              merged with fill data in cache (Section 15.7.1.3)
    |    |  S_CEFSTS<RDE> AND S_CEFSTS<REQ_FILL_DONE>
    |    |  AND S_CEFSTS<WRITE> AND S_CEFSTS<OREAD>
    |    +--------------------------------------------> NDAL read data error on OREAD for write from Mbox after
    |                                                   write data merged with fill data in cache (Section 15.7.1
    |    |  S_CEFSTS<UNEXPECTED_FILL>
    |    +--------------------------------------------> Unexpected NDAL fill received.
    |    |                                              (Section 15.7.1.3.1)
    |    |  otherwise
    |    +--------------------------------------------> Not a hard error interrupt cause (see soft error interru
    |                                                   events)
    |  S_CEFSTS<LOST_ERR>
    +-------------------------------------------------> Lost Bcache fill error
    |                                                   (Section 15.7.1.3.2)
    v
    1
```

Figure 15–8 Cont'd on next page

Figure 15–8 (Cont.): Cause Parse Tree for Hard Error Interrupts

```
1
v
|  S_NESTS<NOACK>
+----+ (select one)
|    |
|    |  S_NEOCMD<CMD>=WRITE
|    +--------------------------------------------> no-ACK on WRITE command or data cycle
|    |                                               (Section 15.7.1.4)
|    |  S_NEOCMD<CMD>=WDISOWN
|    +--------------------------------------------> no-ACK on WDISOWN command or data cycle
|    |                                               (Section 15.7.1.4)
|    |  otherwise
|    +--------------------------------------------> Not a hard error interrupt cause (see soft error interrupt
|                                                    events)
|  S_NESTS<LOST_OERR>
+-------------------------------------------------> Lost no-ACK error
|                                                    (Section 15.7.1.5)
|  (status consistent with hard error interrupt
|  in system environment error registers)
+-------------------------------------------------> Hard error interrupt from system environment
|                                                    (Section 15.7.1.6)
|  otherwise
+-------------------------------------------------> Inconsistent status (Section 15.7.1.7)

Notation:
    (select one)             - Exactly one case must be true. If zero or more than one is
                               true, the status is inconsistent.
    (select all)             - More than one case may be true.
    (select all, at least one) - All the cases are possible causes of a hard error interrupt.
                               More than one may be true. At least one must be true or the status
                               is inconsistent.  A case is not considered true if it evaluates to
                               "Not a hard error interrupt cause".
    otherwise                - fall-through case for (select one) if no other case is true.
    none of the above        - fall-through case for (select all) or (select all, at least one)
                               if no other case is true.
```

15.7.1.1 Uncorrectable Data Errors and Addressing Errors During Write or Write-Unlock Processing

Description: In processing a write or write-unlock, the Cbox detected an addressing error or an uncorrectable ECC error on the data read from the Bcache data RAMs. The write data has already been merged with the corrupted Bcache data and the write of the merged ("bad") data occurred. Data from the write is lost.

There are two types of uncorrectable Bcache data RAM errors: addressing errors and uncorrectable ECC errors. Both are detected through the ECC check logic. Uncorrectable ECC errors indicate that two or more bits of the stored data quadword have changed and the error correcting code can not correct the data. A multiple-bit data error can appear to be addressing error, though it is extremely unlikely. A single-bit error combined with an addressing error appears as an uncorrectable error.

Addressing errors indicate that the location read from the data RAM was probably written using a different address than the one used to read it out. The actual hardware failure could have occurred in the previous data RAM write or the current read. Addressing errors are more serious than uncorrectable ECC errors since they indicate the integrity of the entire Bcache is questionable. Also, there is less than a 100% chance that a given addressing error will result in recognition of an addressing error. This is because addressing errors are recognized by encoding the parity of the address with the data and checking it on read back. All single-bit addressing errors are detectable. Note that addressing errors on writes are never detected if that data is never read out again.

The Cbox inverts three of the check bits being written back into the data RAMs to ensure that if the data is read again an uncorrectable error will be detected. If a subsequent read occurs, S_BCEDSTS<LOST_ERR> should be set, and the instruction which issued the read will machine check. However this mechanism is not fully reliable at ensuring that a subsequent read will detect the error (see Section 15.11.1, Note On Tagged-Bad Data Mechanisms).

For either case, the physical address is determined from the contents of S_BCEDIDX using the procedure in Section 15.3.3.1.2.4. (If the physical address is found to be in IO space, it is an inconsistent status. See Section 15.7.1.7.) S_BCEDECC contains the syndrome calculated by the ECC logic. The Bcache is in ETM.

If the block's tag is found to contain an ECC error, then the address can not be determined.

It should never be the case that both S_BCEDSTS<BAD_ADDR> and S_BCEDSTS<UNCORR> are set. If they are, it is an inconsistent status (see Section 15.7.1.7).

Recovery procedures (addressing error): Clear BCEDSTS<BAD_ADDR, LOCK>.

Recovery procedures (uncorrectable ECC error): Clear BCEDSTS<UNCORR, LOCK>.

Recovery procedures (both cases): The data in this block is lost. Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache). Flushing the Bcache should cause a writeback error (in which BADWDATA will be sent on the NDAL), so BCEDSTS and NESTS should be cleared beforehand. Then use the system specific procedure to clear the tagged-bad state from this block in memory.

It is possible that no writeback error will occur, or that it will happen at the wrong address. This would occur if an error in the data RAMs caused the data to appear as correctable or without error even though it was written with three ECC bits inverted. Also, this could occur if the data was written to a different location than intended (addressing error). If this happens, then the block in memory will incorrectly appear to be good data.

NOTE

When clearing the tagged-bad data state of memory, software must first ensure that no more accesses to the block can occur. Otherwise there is the danger that some process on some other processor or a DMA IO device will see incorrect data and not detect an error.

Restart condition (addressing error): Addressing errors occur on data RAM reads and writes. Because the Cbox writes "bad" data back into the location, there is no way to distinguish transient read errors from transient write errors. Therefore, the worst case has to be assumed: some previous write was written to the wrong place in the Bcache or the failing write has been written to the wrong location in the Bcache. In other words, not only is the block whose address is known corrupted, but another block is as well. No restart is possible. The integrity of the entire system is questionable. Crash the system.

Restart condition (uncorrectable ECC error): If the address of the data is available and no unexpected writeback errors occurred during the Bcache flush, software must determine if the lost data is fatal to one process or the whole system and take the appropriate action.

If the address of the data could not be determined or unexpected errors occurred during the Bcache flush, crash the system.

15.7.1.2 Lost Bcache Data RAM Hard Errors

Description: Some number of unrecoverable Bcache data RAM errors occurred and were not latched because BCEDSTS already contained a report of an unrecoverable error. There is no guarantee this error could have caused a hard error interrupt, though it may be a cause.

Lost Bcache data RAM errors may be caused by more than one operand prefetch to the same cache block.

Bcache data RAM errors which cause hard error interrupt indicate that write data has been lost. Specifically, a read-modify-write operation for a write or write-unlock had an uncorrectable ECC error or an addressing error. The data was written back into the RAMs with three check bits inverted.

The Bcache is in ETM.

Pending interrupts: A soft error interrupt may be pending.

Recovery procedures: Clear BCEDSTS<LOST_ERR>. Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache).

Restart condition: No restart is possible since the errors which were not recorded could potentially have caused lost write data and no indication of what data is lost exists (based on the fact that this error was reported by hard error interrupt). Also, the possibility exists that a subsequent read to any location which had this error could receive incorrect data with no error indication. Crash the system.

NOTE

> The lost data should be marked bad through the Bcache tagged-bad scheme. But there is a significant probability of an error converting that tagged-bad location back to good data. This is because precisely the location which had the data error is being depended on to store a different value without an error. The Bcache tagged-bad scheme does not reliably preserve the bad data status of the location in the presence of errors (see Section 15.11.1, Note On Tagged-Bad Data Mechanisms). So the tagged-bad locations may appear good to a subsequent reader. This is why the system must be crashed.

15.7.1.3 · Bcache Timeout or Read Data Error in Quadword OREAD Fill After Write Data Merged

Description: A D-stream ownership read for a write or write-unlock timed out or terminated receiving an RDE fill response after the requested quadword was received. The error could be sue to an error in the system environment or to any previous error in the system environment or this CPU which leaves a cache block marked as owned in memory and not marked as owned in any cache in the system.

The quadword physical address is in S_CEFADR. The address should not be in IO space. If it is, it is an inconsistent status (see Section 15.7.1.7). The merged data is in the Bcache in the quadword indicated in S_CEFADR. The ownership and valid bits in the Bcache are not set.

CEFSTS<WRITE> should not be set. If it is, it is an inconsistent status (see Section 15.7.1.7).

Recovery procedures: Clear CEFSTS<LOCK>. Clear CEFSTS<TIMEOUT> if the error is a timeout, and CEFSTS<RDE> if it is a read data error. Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache).

Depending on the system environment, memory may have set its ownership bit for this block. This should be predictable for the given system environment because at least one quadword of data was received successfully. If the bit is set, then subsequent reads and writes to the same location may fail while the error is being handled.

The data in memory should be unchanged. The quadword containing the merged data is in the Bcache.

In general, the memory block can not be repaired. However, assuming the memory block is left owned, no writes to the block have timed out in memory, and the block is private to the interrupted job, it can be repaired by the following procedure.

- Extract the addressed quadword from the Bcache (see Section 15.3.3.1.2.3).
- Reset memory's ownership state (see Section 15.3.3.1.2.2.2) and write the extracted quadword to memory.

NOTE

> Software must somehow ensure that no writes to this block are pending in the memory before beginning the repair. This can be done by waiting an amount of time equal to a memory subsystem write timeout time.)

If memory's ownership bit is not set, the block can not be repaired.

Restart condition: If memory state repair is successful, restart. Otherwise, software must determine if the lost data is fatal to one process or the whole system and take the appropriate action.

15.7.1.3.1 Unexpected Fill Error

Description: At least one fill was received when none for that transaction ID was expected by the NVAX CPU. This can only occur if a serious NDAL error has occurred. Reads previous to this event may have received incorrect data.

If S_CEFSTS<RDE> is set, the unexpected fill was an RDE NDAL transaction.

The Bcache is in ETM. S_CEFADR is UNPREDICATBLE.

Recovery procedures: Clear CEFSTS<LOCK, UNEXPECTED_FILL>. Flush the Bcache and clear CCTL<HW_ETM> (in that order).

Restart condition: Data may have been corrupted in memory because of incorrect read data being processed. Crash the system.

15.7.1.3.2 Lost Bcache Fill Error

Description: Either at least one fill error occurred in an OREAD after write data was merged or an unexpected fill was received. The error was not latched because CEFSTS and associated registers already contained a report of an unrecoverable error. There is no guarantee this error could have caused a hard error interrupt, though it may be a cause.

The Bcache may be in ETM. Read S_CCTL<HW_ETM> to find out.

Pending interrupts: A soft error interrupt may be pending.

Recovery procedures: Clear CEFSTS<LOST_ERR>. If the Bcache is in ETM, flush it and clear CCTL<HW_ETM> (in that order).

Restart condition: Data has been corrupted but the address is unknown. Crash the system.

15.7.1.4 NDAL No-ACK During WRITE or WDISOWN

Description: When the Cbox issues an NDAL WRITE or WDISOWN on the NDAL and it is not acknowledged, the Cbox requests a hard error interrupt. This could be because the external component(s) received bad NDAL parity or it could be due to a system-specific notification of "non-existent memory or IO location". The transaction is not retried by hardware, so the data is lost. Typically, for writebacks, the Bcache location is overwritten soon after this error, so there is no way to recover the data from the Bcache.

The Bcache is in ETM. S_NEOADR contains the physical address. S_NEOCMD contains the byte mask and NDAL command.

Recovery procedures: Clear NESTS<NOACK>. Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache).

Retry condition: Software must determine if the lost data is fatal to one process or the whole system and take the appropriate action.

15.7.1.5 Lost NDAL No-ACK Hard Errors

Description: Some number of outgoing NDAL WRITE or WDISOWN commands were not acknowledged and were not latched because NESTS, NEOCMD, and NEOADR already contained a report of an NDAL output error. There is no guarantee this error could have caused the hard error interrupt, though it may be a cause.

Pending interrupts: A soft error interrupt may be pending.

Recovery procedures: Clear NESTS<LOST_NOACK>.

Restart condition: No restart is possible since the errors which were not recorded could potentially have caused lost write data. No indication of what data is lost exists. Crash the system.

15.7.1.6 System Environment Hard Error Interrupts

Description: Errors which occur in the system environment and result in loss of data and which can not notify the NVAX CPU by returning RDE notify the CPU of the error by asserting **H_ERR_L** (e.g., write errors). Errors which can be signaled by RDE should not use hard error interrupt notification. Errors which are corrected automatically by hardware and do not result in loss of data should use soft error interrupt notification instead.

NOTE

> It is very important that components in the system environment which assert **H_ERR_L** have a CPU accessible register which unambiguously reports the **H_ERR_L** assertion. Otherwise, system specific error handling for the hard error interrupt would always crash the system (every time).

It is also strongly recommended that an address be stored where applicable. This may allow the operating system to kill one process or job instead of crashing the system in the event of that hard error.

Recovery procedures: Clear the error status bits in the system registers and perform any necessary system dependent recovery procedure.

Restart condition: Depends on the error. If the system environment reports the address of the lost data (where applicable) software may be able to kill just one process instead of crashing the system.

15.7.1.7 Inconsistent Status in Hard Error Interrupt Cause Analysis

Description: A presumed impossible error report was found in the error registers. This could be due to a hardware failure or bug.

Recovery procedures: No specific recovery action is called for.

Restart condition: No retry is possible. The integrity of the entire system is questionable. Crash the system.

15.8 Soft Error Interrupts

Soft error interrupts are requested to report errors which were detected, but did not affect instruction execution. This results in an interrupt at IPL 1A (hex) to be dispatched through SCB vector 54 (hex).

The stack frame for a soft error interrupt is shown in Figure 15–9.

Figure 15–9: Soft Error Interrupt Stack Frame

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              PC                                               | :(SP)
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              PSL                                              |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

15.8.1 Events Reported Via Soft Error Interrupts

This section describes all the errors which can cause a soft error interrupt. A parse tree is given which shows how to determine the cause of a given soft error. After that, there is a description of each error. For each error, the recovery procedure is given. Where appropriate, the conditions for restart are given. See Section 15.3.3 and Section 15.3.4 for more on error recovery and error retry.

Figure 15–10 is a parse tree which should be used to analyze the cause of a soft error interrupt. It is assumed that the state being analyzed is the saved state, as described in Section 15.3.1. Otherwise the state could change during the analysis procedure, leading to possibly incorrect conclusions. (See Section 15.3.2 for general information about error analysis.)

Note that many errors which cause a soft error interrupt may also lead to a machine check exception. For this reason, a soft error interrupt with no apparent cause is not an inconsistent state unless the CPU has executed an instruction while IPL was lower than 1A (hex) since the most recent machine check exception.

When a soft error interrupt is the only notification for any memory read error which could cause a machine check, the error didn't cause a machine check for one of the following reasons.

- The error did not occur on the quadword the Ebox or Ibox requested (Pcache fill error).
- The Ebox took an interrupt before accessing an instruction or operand which was prefetched by the Ibox. (It could be this soft error interrupt.)
- A prefetched instruction or operand belonged to an instruction following a mispredicted branch, so the Ebox never executed the instruction (and it was flushed from the pipeline when the branch mispredict was recognized).
- The Ebox took an exception for a different reason before attempting to use an instruction execution dispatch or access an operand prefetched by the Ibox. (The pipeline was flushed because of the exception.)

Figure 15–10: Cause Parse Tree for Soft Error Interrupts

```
SOFT ERROR INTERRUPT
----+ (select all, at least one)
    |
    |  S_ICSR<LOCK>
    +----+ (select all, at least one)
    |    |
    |    |  S_ICSR<DPERR>
    |    +------------------------------------------> VIC (virtual instruction cache) data parity error
    |    |                                            (Section 15.8.1.1)
    |    |  S_ICSR<TPERR>
    |    +------------------------------------------> VIC tag parity error (Section 15.8.1.1)
    |    |
    |    |  none of the above
    |    +------------------------------------------> Inconsistent status (no ICSR error bits set)
    |                                                 (Section 15.8.1.22)
    |  S_PCSTS<LOCK>
    +----+ (select all, at least one)
    |    |
    |    |  S_PCSTS<DPERR>
    |    +------------------------------------------> Pcache data parity error (Section 15.8.1.2)
    |    |
    |    |  S_PCSTS<RIGHT_BANK>
    |    +------------------------------------------> Pcache tag parity error in right bank
    |    |                                            (Section 15.8.1.2)
    |    |  S_PCSTS<LEFT_BANK>
    |    +------------------------------------------> Pcache tag parity error in left bank
    |    |                                            (Section 15.8.1.2)
    |    |  otherwise
    |    +------------------------------------------> Inconsistent status (no PCSTS error bits set)
    |                                                 (Section 15.8.1.22)
    |  S_BCETSTS<LOCK>
    +----+ (select one)
    |    |
    |    |  S_BCETSTS<UNCORR>
    |    +----+ (select one)
    |    |    |
    |    |    |  S_BCETSTS<TS_CMD>=DREAD
    |    |    +-------------------------------------> Bcache tag store uncorrectable ECC error on D-stream rea
    |    |    |                                       (Section 15.8.1.3)
    |    |    |  S_BCETSTS<TS_CMD>=IREAD
    |    |    +-------------------------------------> Bcache tag store uncorrectable ECC error on I-stream rea
    |    |    |                                       (Section 15.8.1.3)
    |    |    |  S_BCETSTS<TS_CMD>=OREAD
    |    |    +-------------------------------------> Bcache tag store uncorrectable ECC error on write or
    |    |    |                                       read-lock (Section 15.8.1.3)
    |    |    |  S_BCETSTS<TS_CMD>=WUNLOCK
    |    |    +-------------------------------------> Bcache tag store uncorrectable ECC error on write-unlock
    |    |    |                                       (done only in ETM) (Section 15.8.1.3)
    |    |    |  S_BCETSTS<TS_CMD>=R_INVAL
    |    |    +-------------------------------------> Bcache tag store uncorrectable ECC error on writeback
    |    |    |                                       request type of NDAL operation (Section 15.8.1.3)
    |    |    |  S_BCETSTS<TS_CMD>=O_INVAL
    |    |    +-------------------------------------> Bcache tag store uncorrectable ECC error on
    |    |    |                                       writeback-and-invalidate type of NDAL operation (Section
    |    |    |  S_BCETSTS<TS_CMD>=IPR_DEALLOCATE
    |    |    +-------------------------------------> Bcache tag store uncorrectable ECC error on software
    |    |    |                                       forced deallocate (Section 15.8.1.3)
    |    |    |  otherwise
    |    |    +-------------------------------------> Inconsistent status (invalid command)
    |    |                                            (Section 15.8.1.22)
    v    v
    1    2
```

Figure 15–10 Cont'd on next page

Figure 15–10 (Cont.): Cause Parse Tree for Soft Error Interrupts

```
1    2
v    v
|    |  S_BCETSTS<BAD_ADDR>
|    +----+ (select one)
|    |    |
|    |    |  S_BCETSTS<TS_CMD>=DREAD
|    |    +----------------------------------------> Bcache tag store addressing error on D-stream read
|    |    |                                          (Section 15.8.1.3)
|    |    |  S_BCETSTS<TS_CMD>=IREAD
|    |    +----------------------------------------> Bcache tag store addressing error on I-stream read
|    |    |                                          (Section 15.8.1.3)
|    |    |  S_BCETSTS<TS_CMD>=OREAD
|    |    +----------------------------------------> Bcache tag store addressing error on write or
|    |    |                                          read-lock (Section 15.8.1.3)
|    |    |  S_BCETSTS<TS_CMD>=WUNLOCK
|    |    +----------------------------------------> Bcache tag store addressing error on write-unlock
|    |    |                                          (done only in ETM) (Section 15.8.1.3)
|    |    |  S_BCETSTS<TS_CMD>=R_INVAL
|    |    +----------------------------------------> Bcache tag store addressing error on writeback
|    |    |                                          request type of NDAL operation (Section 15.8.1.3)
|    |    |  S_BCETSTS<TS_CMD>=O_INVAL
|    |    +----------------------------------------> Bcache tag store addressing error on
|    |    |                                          writeback-and-invalidate type of NDAL operation (Section 15.8.1.3)
|    |    |  S_BCETSTS<TS_CMD>=IPR_DEALLOCATE
|    |    +----------------------------------------> Bcache tag store addressing error on software
|    |    |                                          forced deallocate (Section 15.8.1.3)
|    |    |  otherwise
|    |    +----------------------------------------> Inconsistent status (invalid command)
|    |                                               (Section 15.8.1.22)
|    |  otherwise
|    +---------------------------------------------> Inconsistent status (no BCETSTS error bits set)
|                                                    (Section 15.8.1.22)
|  S_BCETSTS<LOST_ERR>
+--------------------------------------------------> Lost unrecoverable Bcache tag store error
|                                                    (Section 15.8.1.4)
v
1
```

Figure 15–10 Cont'd on next page

Figure 15–10 (Cont.): Cause Parse Tree for Soft Error Interrupts

```
1
v
|  S_BCETSTS<CORR>
+----+ (select one)
|    |
|    |  S_BCETSTS<LOCK>
|    +-------------------------------------------> Lost Bcache tag store correctable error
|    |                                             (Section 15.8.1.6)
|    |  otherwise
|    +----+ (select one)
|         |
|         |  S_BCETSTS<TS_CMD>=DREAD
|         +--------------------------------------> Bcache tag store correctable ECC error on D-stream read
|         |                                        (Section 15.8.1.5)
|         |  S_BCETSTS<TS_CMD>=IREAD
|         +--------------------------------------> Bcache tag store correctable ECC error on I-stream read
|         |                                        (Section 15.8.1.5)
|         |  S_BCETSTS<TS_CMD>=OREAD
|         +--------------------------------------> Bcache tag store correctable ECC error on write or
|         |                                        read-lock (Section 15.8.1.5)
|         |  S_BCETSTS<TS_CMD>=WUNLOCK
|         +--------------------------------------> Bcache tag store correctable ECC error on write-unlock
|         |                                        (done only in ETM) (Section 15.8.1.5)
|         |  S_BCETSTS<TS_CMD>=R_INVAL
|         +--------------------------------------> Bcache tag store correctable ECC error on writeback
|         |                                        request type of NDAL operation (Section 15.8.1.5)
|         |  S_BCETSTS<TS_CMD>=O_INVAL
|         +--------------------------------------> Bcache tag store correctable ECC error on
|         |                                        writeback-and-invalidate type of NDAL operation (Section
|         |  S_BCETSTS<TS_CMD>=IPR_DEALLOCATE
|         +--------------------------------------> Bcache tag store correctable ECC error on software
|         |                                        forced deallocate (Section 15.8.1.5)
|         |  otherwise
|         +--------------------------------------> Inconsistent status (invalid command)
|                                                  (Section 15.8.1.22)
v
1
```

Figure 15–10 Cont'd on next page

Figure 15–10 (Cont.): Cause Parse Tree for Soft Error Interrupts

```
1
v
| S_BCEDSTS<CORR>
+----+ (select one)
|    |
|    |  S_BCEDSTS<LOCK>
|    +--------------------------------------------> Lost Bcache data RAM correctable error
|    |                                              (Section 15.8.1.8)
|    |  otherwise
|    +----+ (select one)
|         |
|         |  S_BCEDSTS<DR_CMD>=DREAD
|         +---------------------------------------> Bcache data RAM correctable error on D-stream read
|         |                                         (Section 15.8.1.7)
|         |  S_BCEDSTS<DR_CMD>=IREAD
|         +---------------------------------------> Bcache data RAM correctable error on I-stream read
|         |                                         (Section 15.8.1.7)
|         |  S_BCEDSTS<DR_CMD>=WRITEBACK
|         +---------------------------------------> Bcache data RAM correctable error on writeback
|         |                                         (Section 15.8.1.7)
|         |  S_BCEDSTS<DR_CMD>=RMW
|         +---------------------------------------> Bcache data RAM correctable error on read-modify-write
|         |                                         for write or write-unlock (Section 15.8.1.7)
|         |  otherwise
|         +---------------------------------------> Inconsistent status (invalid command)
|                                                   (Section 15.8.1.22)
| S_BCEDSTS<LOCK> AND
| NOT S_PCSTS<PTE_ER>
+----+ (select one)
|    |
|    |  S_BCEDSTS<UNCORR>
|    +----+ (select one)
|    |    |
|    |    |  S_BCEDSTS<DR_CMD>=DREAD
|    |    +---------------------------------------> Bcache data RAM uncorrectable ECC error on D-stream read
|    |    |                                         (or Pcache fill for read-lock) (Section 15.8.1.9)
|    |    |  S_BCEDSTS<DR_CMD>=IREAD
|    |    +---------------------------------------> Bcache data RAM uncorrectable ECC error on I-stream read
|    |    |                                         (Section 15.8.1.9)
|    |    |  S_BCEDSTS<DR_CMD>=WRITEBACK
|    |    +---------------------------------------> Bcache data RAM uncorrectable ECC error on writeback
|    |    |                                         (Section 15.8.1.10)
|    |    |  otherwise
|    |    +---------------------------------------> Inconsistent status (all other cases cause hard error
|    |                                              interrupt) (Section 15.8.1.22)
v    v
1    2
```

Figure 15–10 Cont'd on next page

Figure 15–10 (Cont.): Cause Parse Tree for Soft Error Interrupts

```
1    2
v    v
|    |  S_BCEDSTS<BAD_ADDR>
|    +----+ (select one)
|    |    |
|    |    |  S_BCEDSTS<DR_CMD>=DREAD
|    |    +---------------------------------------> Bcache data RAM addressing error on D-stream read
|    |    |                                         (or Pcache fill for read-lock) (Section 15.8.1.9)
|    |    |  S_BCEDSTS<DR_CMD>=IREAD
|    |    +---------------------------------------> Bcache data RAM addressing error on I-stream read
|    |    |                                         (Section 15.8.1.9)
|    |    |  S_BCEDSTS<DR_CMD>=WRITEBACK
|    |    +---------------------------------------> Bcache data RAM addressing error on writeback
|    |    |                                         (Section 15.8.1.10)
|    |    |  otherwise
|    |    +---------------------------------------> Inconsistent status (all other cases cause hard error
|    |                                              interrupt) (Section 15.8.1.22)
|    |  otherwise
|    +--------------------------------------------> Inconsistent Status (no error bits set in BCEDSTS)
|                                                   (Section 15.8.1.22)
|  S_BCEDSTS<LOST_ERR> AND
|  NOT S_PCSTS<PTE_ER>
+----+
|    |
|    |  S_NESTS<BADWDATA> OR S_NESTS<LOST_OERR>
|    +--------------------------------------------> Lost unrecoverable Bcache data RAM error with possible
|    |                                              lost writeback error (Section 15.8.1.11)
|    |  otherwise
|    +--------------------------------------------> Lost unrecoverable Bcache data RAM error
|                                                   (Section 15.8.1.12)
|  S_CEFSTS<LOCK> AND
|  NOT S_PCSTS<PTE_ER>
+----+ (select one)
|    |
|    |  S_CEFSTS<TIMEOUT>
|    +----+ (select one)
|    |    |
|    |    |  S_CEFSTS<OREAD>
|    |    +----+ (select one)
|    |    |    |
|    |    |    |  S_CEFSTS<WRITE> AND
|    |    |    |  NOT S_CEFSTS<TO_MBOX>
|    |    |    +----+ (select one)
|    |    |    |    |
|    |    |    |    |  S_CEFSTS<REQ_FILL_DONE>
|    |    |    |    +-----------------------------> Inconsistent status (should cause hard error interrupt)
|    |    |    |    |                               (Section 15.8.1.22)
|    |    |    |    |  otherwise
|    |    |    |    +-----------------------------> D-stream NDAL ownership read for Mbox write timeout
|    |    |    |                                    error before write data merged with fill data (Section 15
|    |    |    |  S_CEFSTS<TO_MBOX>
|    |    |    +----------------------------------> D-stream NDAL ownership read timeout error
|    |    |    |                                    (modify operand or read-lock) (Section 15.8.1.13)
|    |    |    |  otherwise
|    |    |    +----------------------------------> Inconsistent status (either WRITE or TO_MBOX, but not bo
|    |    |                                         should be set) (Section 15.8.1.22)
v    v    v
1    2    3
```

Figure 15–10 Cont'd on next page

Figure 15–10 (Cont.): Cause Parse Tree for Soft Error Interrupts

```
1     2     3
v     v     v
|     |     |  otherwise
|     |     +-----+ (select one)
|     |           |
|     |           |  S_CEFSTS<IREAD>
|     |           +-----------------------------------------> I-stream NDAL read timeout error (Section 15.8.1.13)
|     |           |
|     |           |  S_CEFSTS<TO_MBOX>
|     |           +-----------------------------------------> D-stream NDAL read timeout error (read only operand)
|     |           |                                           (Section 15.8.1.13)
|     |           |  otherwise
|     |           +-----------------------------------------> Inconsistent status (TO_MBOX should be set)
|     |                                                       (Section 15.8.1.22)
|     |  S_CEFSTS<RDE>
|     +-----+ (select one)
|     |     |
|     |     |  S_CEFSTS<OREAD>
|     |     +-----+ (select one)
|     |     |     |
|     |     |     |  S_CEFSTS<WRITE>
|     |     |     |  AND NOT S_CEFSTS<TO_MBOX>
|     |     |     +-----+ (select one)
|     |     |     |     |
|     |     |     |     |  S_CEFSTS<REQ_FILL_DONE>
|     |     |     |     +-----------------------------------> Inconsistent status (should cause hard error interrupt)
|     |     |     |     |                                     (Section 15.8.1.22)
|     |     |     |     |  otherwise
|     |     |     |     +-----------------------------------> D-stream NDAL ownership read for Mbox write read data
|     |     |     |                                           error before write data merged with fill data (Section 15.8.1.14)
|     |     |     |  S_CEFSTS<TO_MBOX>
|     |     |     +-----------------------------------------> D-stream NDAL ownership read read data error
|     |     |     |                                           (modify operand or read-lock) (Section 15.8.1.14)
|     |     |     |  otherwise
|     |     |     +-----------------------------------------> Inconsistent status (either WRITE or TO_MBOX, but not both,
|     |     |                                                 should be set) (Section 15.8.1.22)
|     |     |  otherwise
|     |     +-----+ (select one)
|     |           |
|     |           |  S_CEFSTS<IREAD>
|     |           +-----------------------------------------> I-stream NDAL read read data error
|     |           |                                           (Section 15.8.1.14)
|     |           |  S_CEFSTS<TO_MBOX>
|     |           +-----------------------------------------> D-stream NDAL read read data error (read only operand)
|     |           |                                           (Section 15.8.1.14)
|     |           |  otherwise
|     |           +-----------------------------------------> Inconsistent status (TO_MBOX should be set)
|     |                                                       (Section 15.8.1.22)
|     |  otherwise
|     +-----------------------------------------------------> Inconsistent status (either CEFSTS<RDE> or CEFSTS<TIMEOUT>
|                                                             should be set or, if CEFSTS<UNEXPECTED_FILL> is set, it
|                                                             should cause a hard error interrupt) (Section 15.8.1.22)
|  S_CEFSTS<LOST_ERR> AND
|  NOT S_PCSTS<PTE_ER>
+-----------------------------------------------------------> Lost Bcache fill error
|                                                             (Section 15.8.1.15)
v
1
```

Figure 15–10 Cont'd on next page

Figure 15–10 (Cont.): Cause Parse Tree for Soft Error Interrupts

```
1
v
|  S_NESTS<NOACK> AND
|  NOT S_PCSTS<PTE_ER>
+----+ (select one)
|    |
|    | S_NEOCMD<CMD>=IREAD
|    +--------------------------------------------> Unacknowledged I-stream NDAL read (Section 15.8.1.16)
|    |
|    | S_NEOCMD<CMD>=DREAD
|    +--------------------------------------------> Unacknowledged D-stream NDAL read (read only operand)
|    |                                              (Section 15.8.1.16)
|    | S_NEOCMD<CMD>=OREAD
|    +--------------------------------------------> Unacknowledged D-stream NDAL read (modify operand or re
|    |                                              (Section 15.8.1.16)
|    | S_NEOCMD<CMD>=WRITE or WDISOWN
|    +--------------------------------------------> Inconsistent status (should cause hard error interrupt)
|    |                                              (Section 15.8.1.22)
|    | otherwise
|    +--------------------------------------------> Inconsistent status (invalid command in NEOCMD<CMD>)
|                                                   (Section 15.8.1.22)
|  S_NESTS<LOST_OERR> AND
|  NOT S_PCSTS<PTE_ER>
+-------------------------------------------------> Lost NDAL output error (Section 15.8.1.17)
|
|  S_BCEDSTS<LOCK> AND
|  S_PCSTS<PTE_ER>[1]
+----+ (select one)
|    |
|    | S_BCEDSTS<UNCORR>
|    +----+ (select one)
|    |    |
|    |    | S_BCEDSTS<DR_CMD>=DREAD
|    |    +---------------------------------------> Bcache data RAM uncorrectable ECC error on PTE read
|    |    |                                         (Section 15.8.1.18.1)
|    |    | S_BCEDSTS<DR_CMD>=IREAD
|    |    +----+ (select one)
|    |    |    |
|    |    |    | S_BCEDSTS<LOST_ERR>
|    |    |    +----------------------------------> Multiple errors in context of PTE read error
|    |    |    |                                    (Section 15.8.1.18.5)
|    |    |    | otherwise
|    |    |    +----------------------------------> Bcache data RAM uncorrectable ECC error on I-stream rea
|    |    |                                         (Section 15.8.1.9)
v    v    v
1    2    3
```

Figure 15–10 Cont'd on next page

[1] At least one potential PTE cause must be found or the status is inconsistent (see Section 15.8.1.22). Some of the outcomes indicate a potential soft error interrupt cause which is not a potential PTE read error cause. These errors should be treated separately.

Figure 15–10 (Cont.): Cause Parse Tree for Soft Error Interrupts

```
1    2    3
v    v    v
|    |    |  S_BCEDSTS<DR_CMD>=WRITEBACK
|    |    +----+ (select one)
|    |    |    |
|    |    |    |  S_BCEDSTS<LOST_ERR>
|    |    |    +----------------------------------> Multiple errors in context of PTE read error
|    |    |    |                                    (Section 15.8.1.18.5)
|    |    |    |  otherwise
|    |    |    +----------------------------------> Bcache data RAM uncorrectable ECC error on writeback
|    |    |                                         (Section 15.8.1.10)
|    |    |  otherwise
|    |    +---------------------------------------> Inconsistent status (all other cases cause hard error
|    |                                              interrupt) (Section 15.8.1.22)
|    |  S_BCEDSTS<BAD_ADDR>
|    +----+ (select one)
|    |    |
|    |    |  S_BCEDSTS<DR_CMD>=DREAD
|    |    +---------------------------------------> Bcache data RAM addressing error on PTE read
|    |    |                                         (Section 15.8.1.18.1)
|    |    |  S_BCEDSTS<DR_CMD>=IREAD
|    |    +----+ (select one)
|    |    |    |
|    |    |    |  S_BCEDSTS<LOST_ERR>
|    |    |    +----------------------------------> Multiple errors in context of PTE read error
|    |    |    |                                    (Section 15.8.1.18.5)
|    |    |    |  otherwise
|    |    |    +----------------------------------> Bcache data RAM addressing error on I-stream read
|    |    |                                         (Section 15.8.1.9)
|    |    |  S_BCEDSTS<DR_CMD>=WRITEBACK
|    |    +----+ (select one)
|    |    |    |
|    |    |    |  S_BCEDSTS<LOST_ERR>
|    |    |    +----------------------------------> Multiple errors in context of PTE read error
|    |    |    |                                    (Section 15.8.1.18.5)
|    |    |    |  otherwise
|    |    |    +----------------------------------> Bcache data RAM addressing error on writeback
|    |    |                                         (Section 15.8.1.10)
|    |    |  otherwise
|    |    +---------------------------------------> Inconsistent status (all other cases cause hard error
|    |                                              interrupt) (Section 15.8.1.22)
|    |  otherwise
|    +--------------------------------------------> Inconsistent Status (no error bits set in BCEDSTS)
|                                                   (Section 15.8.1.22)
v
1
```

Figure 15–10 Cont'd on next page

[1] At least one potential PTE cause must be found or the status is inconsistent (see Section 15.8.1.22). Some of the outcomes indicate a potential soft error interrupt cause which is not a potential PTE read error cause. These errors should be treated separately.

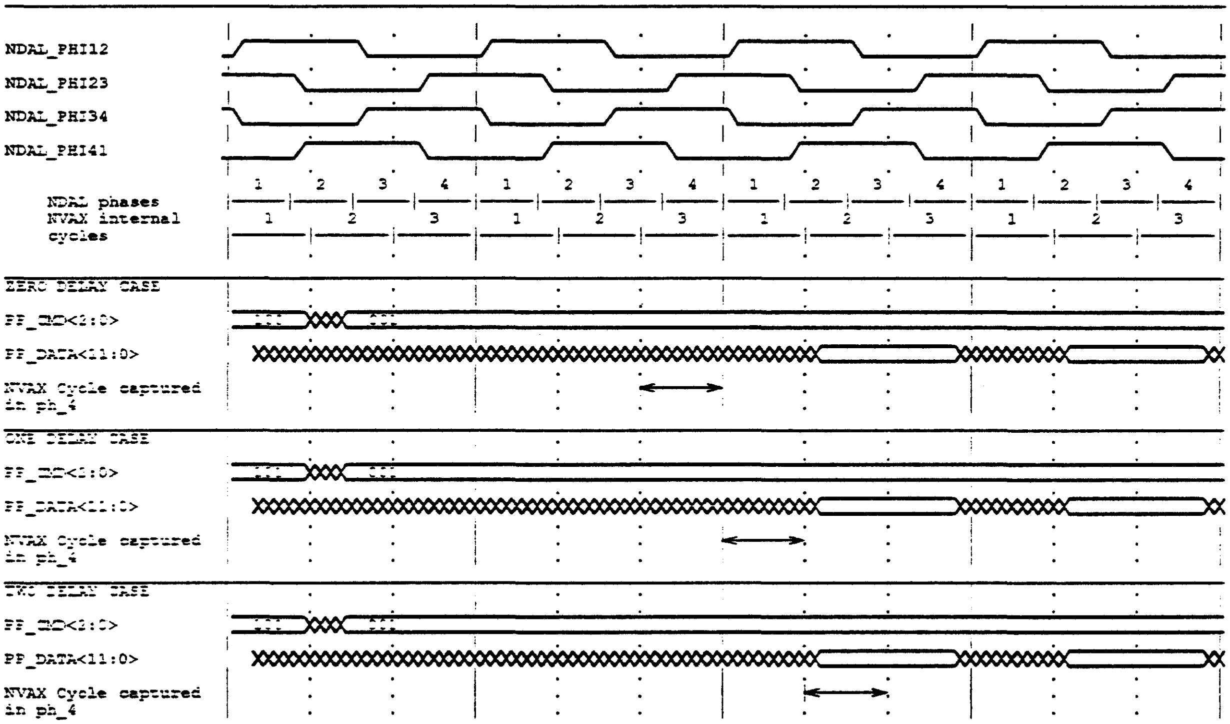
Figure 15–10 (Cont.): Cause Parse Tree for Soft Error Interrupts

```
1
v
|  S_CEFSTS<LOCK> AND
|  S_PCSTS<PTE_ER>1
+----+ (select one)
|    |
|    |  S_CEFSTS<TIMEOUT>
|    +----+ (select one)
|    |    |
|    |    |  S_CEFSTS<OREAD>
|    |    +----+ (select one)
|    |    |    |
|    |    |    |  S_CEFSTS<WRITE> AND
|    |    |    |  NOT S_CEFSTS<TO_MBOX>
|    |    |    +----+ (select one)
|    |    |    |    |
|    |    |    |    |  S_CEFSTS<REQ_FILL_DONE>
|    |    |    |    +-----------------------------> Inconsistent status (should cause hard error interrupt)
|    |    |    |    |                               (Section 15.8.1.22)
|    |    |    |    |  otherwise
|    |    |    |    +----+ (select one)
|    |    |    |         |
|    |    |    |         |  S_CEFSTS<LOST_ERR>
|    |    |    |         +------------------------> Multiple errors in context of PTE read error
|    |    |    |         |                          (Section 15.8.1.18.5)
|    |    |    |         |  otherwise
|    |    |    |         +------------------------> D-stream NDAL ownership read for Mbox write timeout
|    |    |    |                                    error before write data merged with fill data (Section 1
|    |    |    |  S_CEFSTS<TO_MBOX>
|    |    |    +----+ (select one)
|    |    |    |    |
|    |    |    |    |  S_CEFSTS<LOST_ERR>
|    |    |    |    +-----------------------------> Multiple errors in context of PTE read error
|    |    |    |    |                               (Section 15.8.1.18.5)
|    |    |    |    |  otherwise
|    |    |    |    +-----------------------------> D-stream NDAL ownership read timeout error
|    |    |    |                                    (modify operand or read-lock) (Section 15.8.1.13)
|    |    |    |  otherwise
|    |    |    +----------------------------------> Inconsistent status (either WRITE or TO_MBOX, but not b
|    |    |                                         should be set) (Section 15.8.1.22)
|    |    |  otherwise
|    |    +----+ (select one)
|    |         |
|    |         |  S_CEFSTS<IREAD>
|    |         +----+ (select one)
|    |         |    |
|    |         |    |  S_CEFSTS<LOST_ERR>
|    |         |    +-----------------------------> Multiple errors in context of PTE read error
|    |         |    |                               (Section 15.8.1.18.5)
|    |         |    |  otherwise
|    |         |    +-----------------------------> I-stream NDAL read timeout error (Section 15.8.1.13)
|    |         |
|    |         |  S_CEFSTS<TO_MBOX>
|    |         +----------------------------------> D-stream NDAL read timeout error (PTE read)
|    |         |                                    (Section 15.8.1.18.2)
|    |         |  otherwise
|    |         +----------------------------------> Inconsistent status (TO_MBOX should be set)
|    |                                              (Section 15.8.1.22)
v    v
1    2
```

Figure 15–10 Cont'd on next page

[1] At least one potential PTE cause must be found or the status is inconsistent (see Section 15.8.1.22). Some of the outcomes indicate a potential soft error interrupt cause which is not a potential PTE read error cause. These errors should be treated separately.

Figure 19–2: Internal Scan Register Operation Timing



Note that the initial packets of ISR data contain data from before the load event from the last bit on the chain. After one or two samples, this data is all valid sampled data. The bits from the scan chain are serial-to-parallel converted as shown in Table 19–3. Note that for ISR1, 9 bits are always visible. Every third NVAX cycle, they shift up by three bit positions.

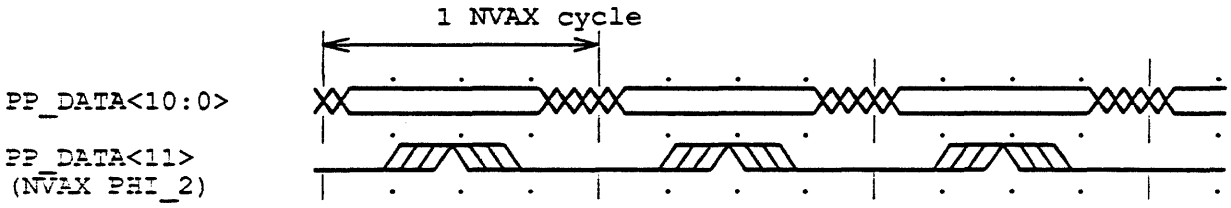
Table 19–3: Serial to Parallel Conversion of Scan Data

	PP_DATA_H Bit	Bit from Scan Chain
ISR1	PP_DATA<5>	Most recently received bit
	PP_DATA_H<4>	Second most recently received bit
	PP_DATA_H<3>	Third most recently received bit
	PP_DATA_H<8:6>	Last PP_DATA_H<5:3> (from 3 NVAX cycles ago)
	PP_DATA_H<11:9>	Last PP_DATA_H<8:6> (from 3 NVAX cycles ago)
ISR2	PP_DATA_H<2>	Most recently received bit
	PP_DATA_H<1>	Second most recently received bit
	PP_DATA_H<0>	Least recently received bit

Observe MAB

For full speed MAB observation, an internal clock is provided which will allow synchronous capture by a DAS in any debug environment. Figure 19–3 shows the the self-relative timing during Observe MAB mode.

Figure 19–3: Self Relative Timing In Observe MAB Mode



Force MAB

During Force MAB mode an internal 11 bit counter forces address on the microaddress bus. The counter is initialized internally by the Ebox. It gets incremented each time FORCE MAB mode is entered, thus allowing it to go through all control store addresses. Refer to the testability sections of Micro-Sequencer chapter for further details of Force MAB operation.

Observe Box Signals

The timing for observing internal signals from boxes follows the basic pattern as that for observing MAB. Note that **PP_DATA_H<11>** may be used for observing box-specific signal. Details of the signals observed may be found in the testability section of each box chapter.

19.5 Test Pads

This port consists of strategic internal nodes brought out to top level of metal in the form of 3x3 micron test pads. These pads will be accessed by probes during chip debug and wafer probe manufacturing tests. The access may primarily provide observability of these nodes, however, controllability may also be provided where appropriate. See the testability sections in box chapters for the list of nodes brought out on the top metal layer.

19.6 System Port

This is simply the normal system I/O of the chip. It is identified as a test access port because of two reason:

- It is used to provide the read/write access to testability features via the VAX architecture's MFPR and MTPR instructions.

- It provides the natural resource for testing the chip via the macro-code based tests.

See the individual box chapters for the list of specific architectural features provided.

19.7 Serial P-cache Port

Instruction stream data may be serially loaded into the P-cache by supplying data on the TEST_DATA_H pin and strobing it with the TEST_STROBE_H pin. Chip microcode collects the bit-serial data, packs it into longwords, and writes the longwords into the P-cache. After loading the P-cache, the microcode passes control to the first MACRO instruction in P-cache.

The serial load follows this flow:

- TEST_STROBE_H is de-asserted while ASYNCH_RESET_L is asserted. TEST_STROBE_H is normally pulled up through on-chip resistors.
- When ASYNCH_RESET_L is de-asserted, the on-chip power-up microcode enters the special burn-in flow.
- When MCHK_H is asserted, TEST_STROBE_H should be de-asserted. The chip is now ready to receive serial data input.
- The first bit of instruction stream data should be placed on TEST_DATA_H, Then TEST_STROBE_H should be asserted.
- TEST_STROBE_H should then be de-asserted. TEST_DATA_H can change on the same edge as the TEST_STROBE de-assertion.
- TEST_STROBE_H may transition at a maximum rate of 1/10 the internal chip clock frequency. There is no minimum rate.
- 32K bits of instruction stream data must be loaded into cache. At this time, MCHK_H will be de-asserted, signifying the cache load is complete. The chip then jumps to the first location in P-cache, attempting to execute an instruction at that location.

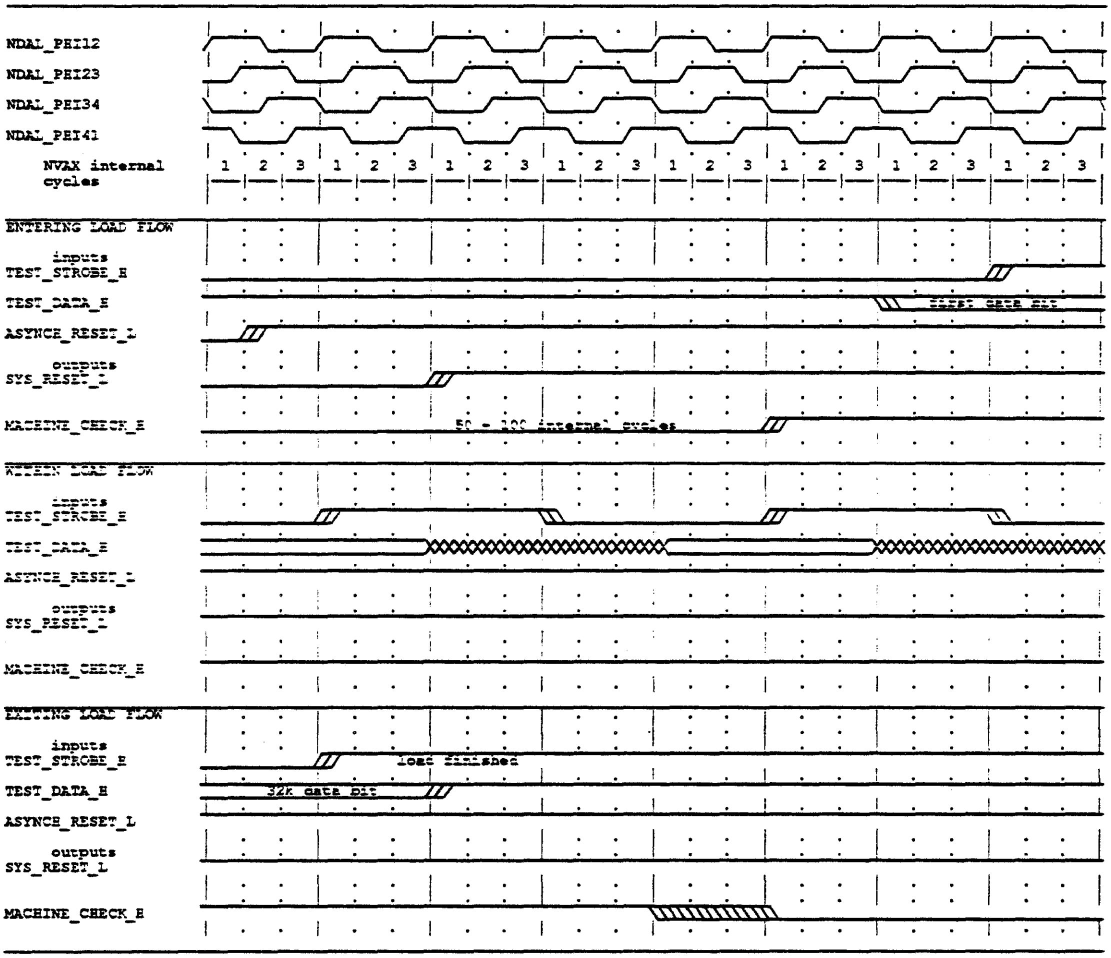
It is difficult to achieve high test coverage in the the burn-in and life-test environments due to limited test pattern bandwidth and the difficulty in synchronizing test equipment to the NVAX chip. Using this serial port, burn-in and life-test programs can load the real "test program" into P-cache, where the chip can perform a self-test.

This scheme minimizes test pattern bandwidth, allows for asynchronous transmission of the serial data, provides a means to stimulate multiple chips under test which are running asynchronously, and supplies a means to achieve high test coverage.

19.8 IEEE 1149.1 (JTAG) Serial Test Port

The Serial Test Port is a 4-pin test access interface based on IEEE 1149.1 standard. (See [2], [3].) In NVAX it is used for accessing and controlling the boundary scan register. The port supports EXTEST, SAMPLE and BYPASS instructions.

Figure 19–4: Serial Port Timing

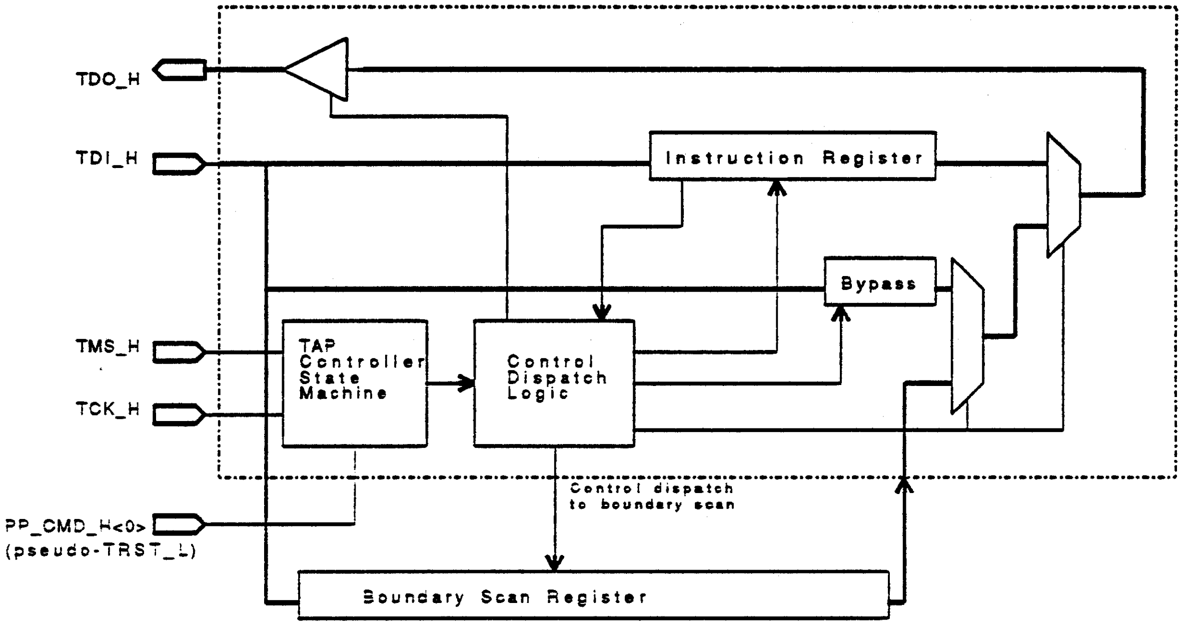


The block diagram of the port logic together with the boundary scan register is shown in Figure 19–5. The port logic shown represents all the logic used in the definition of Common Test Interface (see [2]). It consists of the four-wire Test Access Port (TAP), a TAP controller, an instruction register (IR) and a bypass register (BPR).

The four pins in test access port are TDI_H, TDO_H, TMS_H, and TCK_H. These pins conform to all requirements of the standard. The port also uses PP_CMD_H< 0 > pin as pseudo-TRST_L pin. When asserted low, this pin resets the JTAG test logic. See Section 19.8.5 for more details.

The TAP Controller is a state machine which interprets IEEE 1149.1 protocols received on TMS line and generates appropriate clocks and control signals for the testability features under its jurisdiction.

Figure 19–5: IEEE 1149.1 Serial Port (the Basic CTI)



The Instruction Register resides on a scan path. Its contents are interpreted as test instructions and are used to select the testability modes and features.

The Bypass Register is a one bit shift register which provides a single-bit serial connection through the port (chip) when no other test path is selected.

19.8.1 TAP Controller State Machine

The TAP Controller is a synchronous finite-state state machine that interprets IEEE 1149.1 protocols received on TMS line. The state transitions in the controller are caused by the TMS signal on the rising edge of TCK. In each state, the controller generates appropriate clocks and control signals that control the operation of the testability features. Appropriate actions of the testability features are initiated on the rising edge of TCK following the entry into a state.

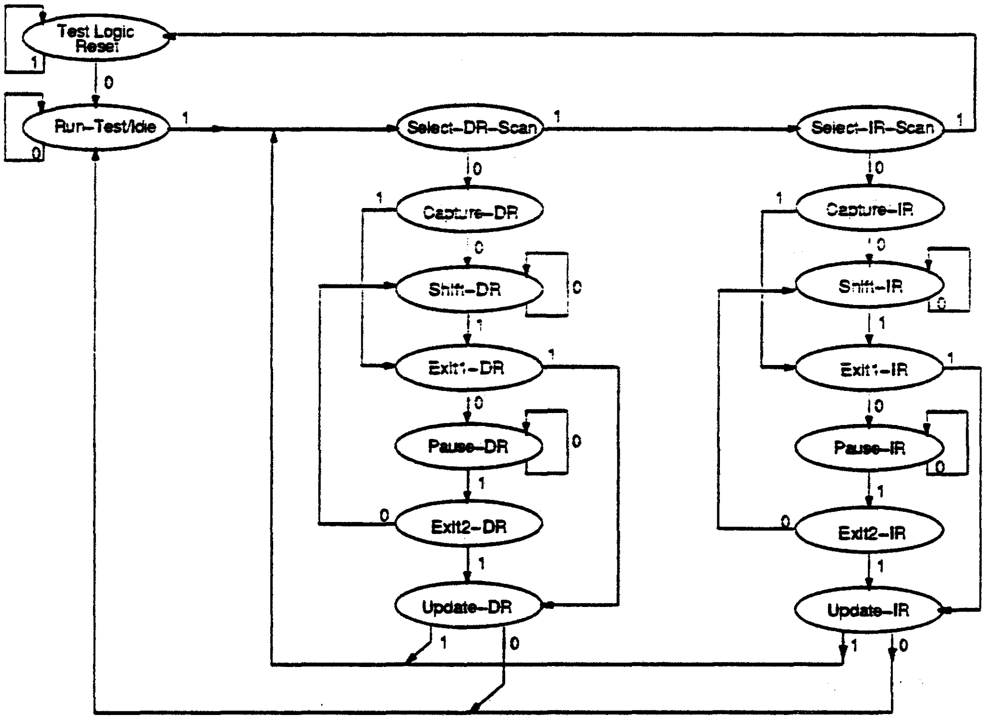
The TAP controller states provide the four basic actions required for testing: transportation of test data (Shift), stimulus application (Update), test execution (Run-Test), and response capture (Capture). Test data are transported generally in the beginning and at the end of a test.

The state diagram for the TAP controller is shown in Figure 19–6. The TAP controller causes appropriate actions to occur only in the testability features selected by the current instruction in the instruction register. All other testability features maintain *status quo*. *Status quo* means that the registers either retain their previous state or continue to operate in their previously selected mode.

A Scan Sequence begins with entry into the Capture State and end with the exit from the Update State. The Scan Sequence entered from Select-DR-Scan controls the instruction register, and the one entered from Select-DR-Scan controls the testability feature selected by the instruction register. The actions caused by the states in the two scan sequences are identical. The following is the brief description of each state.

Figure 19–6: TAP Controller State Machine

Values Shown are TMS



- **Test-Logic-Reset:** This state disables the test logic. The chip performs normal system operation. Testability features are either inactive or are performing normal system operation. The TAP controller is forced into this state at power-up and it continues to remain in this state as long as TMS is held high.
- **Run-Test/Idle:** This is a combined controller state between scan operations when the test logic is either idle or a particular test is running.

  For example, upon entry into this state, an internal test (such as self-test or macrocode test involving data reducers etc.) selected by the current instruction is executed. All other testability features (not involved in the current test) maintain *status quo*.

- **Select-DR-Scan:** This is a temporary controller state in which all test registers (instruction register as well as testability features) maintain *status quo*. If TMS is held low when the controller is in this state, then a scan sequence for the selected test feature is initiated.
- **Select-IR-Scan:** This is a temporary controller state in which all test registers maintain *status quo*. If TMS is held low when the controller is in this state, then a scan sequence for the Instruction Register is initiated.
- **Capture:** In this controller state, the chip data is parallel loaded into the selected test register (Instruction Register or testability feature). This is the state in which the observe action takes place.
- **Shift:** In this state the selected test register shifts data one stage towards its serial output on each rising edge of TCK.
- **Exitl:** This is a temporary controller state where all test registers maintain *status quo*.
- **Pause:** This controller state allows shifting of the selected test register to be temporarily halted. All test registers maintain *status quo*.
- **Exit2:** This is a temporary controller state. All test registers maintain *status quo*.
- **Update:** The selected test register updates its outputs by transferring data from the shifter-stage into parallel output stage. This update action is initiated on the first falling edge of TCK upon entry into the state. All other registers maintain *status quo*.

19.8.2 Instruction Register

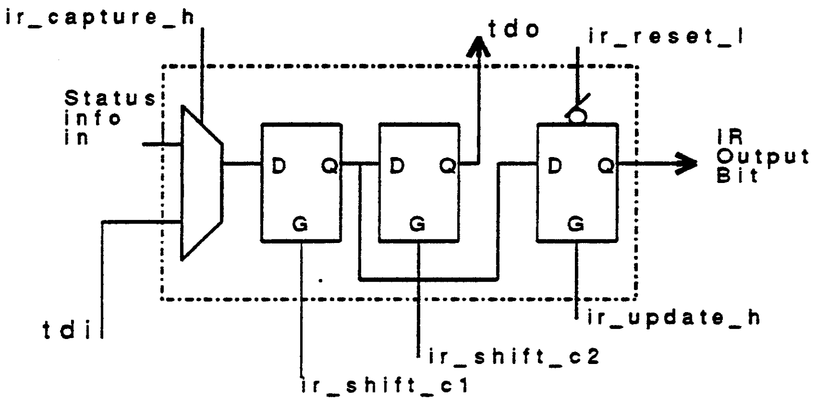
The JTAG Instruction Register on NVAX CPU consists of 2 bits. The two bits are interpreted as per Table 19–4 to select and control the operation of boundary scan register. During Caoture-IR state, the shift register stage of IR is loaded with data '01'. This automatic load feature is useful for testing the integrity of the JTAG scan chain on module.

Table 19–4: Instruction Register

IR< 1:0 >	Test Register Selected	Test Instruction/ Operation
00	Boundary Scan Register	EXTEST. Also forces reset to internal chip logic.
01	Boundary Scan Register	SAMPLE
10	Bypass Register	BYPASS
11	Bypass Register	BYPASS. Default

A cell used in the instruction register is shown in Figure 19–7. The ir_cell operations are controlled by **IR_CAPTURE_H**, **IR_SHIFT_C1**, **IR_SHIFT_C2**, **IR_UPDATE_H** and **IR_RESET_L** signals. These signals are described later.

Figure 19–7: JTAG Instruction Register Cell



19.8.3 Bypass Register

The bypass register provides a one bit scan route though the NVAX chip during a scan-shift operations. It provides a means for effectively bypassing the NVAX CPU chip's test logic during testing at module and system levels.

When the bypass register is selected, a CAPTURE-DR controller state loads a '0' in the bypass register. When the JTAG instruction selects the Bypass operation, Bypass register is selected for the scan operation.

19.8.4 Control Dispatch Logic

Dispatch logic generates signals to control operations of JTAG circuitry, including the the instruction register and the driver on TDO_H pin. It decodes the current instruction in the IR and the current TAP controller state information and dispatches the control signals to the bypass and boundary scan registers. The control signals dispatched are described below..

Dispatch to Boundary Scan Register

- BSR_EXTEST_H: Asserted high when the instruction selects EXTEST instruction. This allows boundary scan cells to drive data on output and I/O pins. BSR_EXTEST_H also forces an internal reset to chip logic. This makes chip's internal logic insensitive to test patterns used for interconenction test.
- BSR_CAPTURE_H: The signal is asserted when TAP controller enters the CAPTURE-DR state and deasserted when the TAP Controller exits CAPTURE-DR state. The signal causes data to be observed into the boundary scan register,
- BSR_SHIFT_C1: Issues a pulse with the falling edge of TCK_H during CAPTURE-DR and SHIFT-DR states.
- BSR_SHIFT_C2: Unconditionally issues a pulse with each rising edge of TCK_H.
- BSR_UPDATE_H: Issues a pulse with the falling edge of TCK_H during UPDATE-DR state. This pulse loads new data into the parallel output latch in *md_bcells* described later.

Dispatch to Bypass Register

Dispatch to Bypass Register consists of **BSR_CAPTURE_H**, **BSR_SHIFT_C1**, and **BSR_SHIFT_C2** signals. Note that these are subset of signals dispatched to the boundary scan register.

Dispatch to Instruction Register

- **IR_CAPTURE_H**: The signal is asserted when TAP controller enters the CAPTURE-IR state and deasserted when the TAP Controller exits CAPTURE-IR state. The signal causes status data ('01') to be observed into IR,
- **IR_SHIFT_C1**: Issues a pulse with the falling edge of TCK_H during CAPTURE-IR and SHIFT-IR state.
- **IR_SHIFT_C2**: Unconditionally issues a shift pulse with each rising edge of TCK_H.

  Note that the data shifts from the most significant bit to least significant bit. The least significant bit is at **TDO_H**.
- **IR_UPDATE_H**: Issues a pulse with the falling edge of TCK_H during UPDATE-IR state. This pulse loads new instruction into the parallel output latches of IR.
- **IR_RESET_L**: This signal initializes the instruction register's output latches. When asserted low, all IR output latches are set high to force BYPASS instruction. **IR_RESET_L** is asserted low when the TAP Controller enters the Test-Logic-Reset state.

Dispatch to TDO Multiplexers and Driver

Multiplexer control is dispatched by decoding the instruction register as per Table 19–4. **ENABLE_TDO_H** is generated as follows.

- **ENABLE_TDO_H**: This signal is asserted high when the TAP controller is in SHIFT-IR or SHIFT-DR states. The signal enables TDO_H pin driver whenever a shift operation is in progress and keeps it disabled all other times.

Figure 19–8 and Figure 19–9 show the timing diagram of the signals dispatched by the Control Dispatch Logic and the behavior of the Boundary Scan Register and the Instruction Register during the *IR-Scan* and *DR-Scan* sequences.

Notice that the implementation must meet the standard's requirement that the changes on TDO_H occur with falling edge of TCK_H signal. In NVAX CPU this requirement is met by including a timing latch at the TDO_H pin. The latch opens when the TCK_H is low and closes when TCK_H is high.

Figure 19–8: IEEE 1149.1 Logic Timing during IR-Scan Sequence

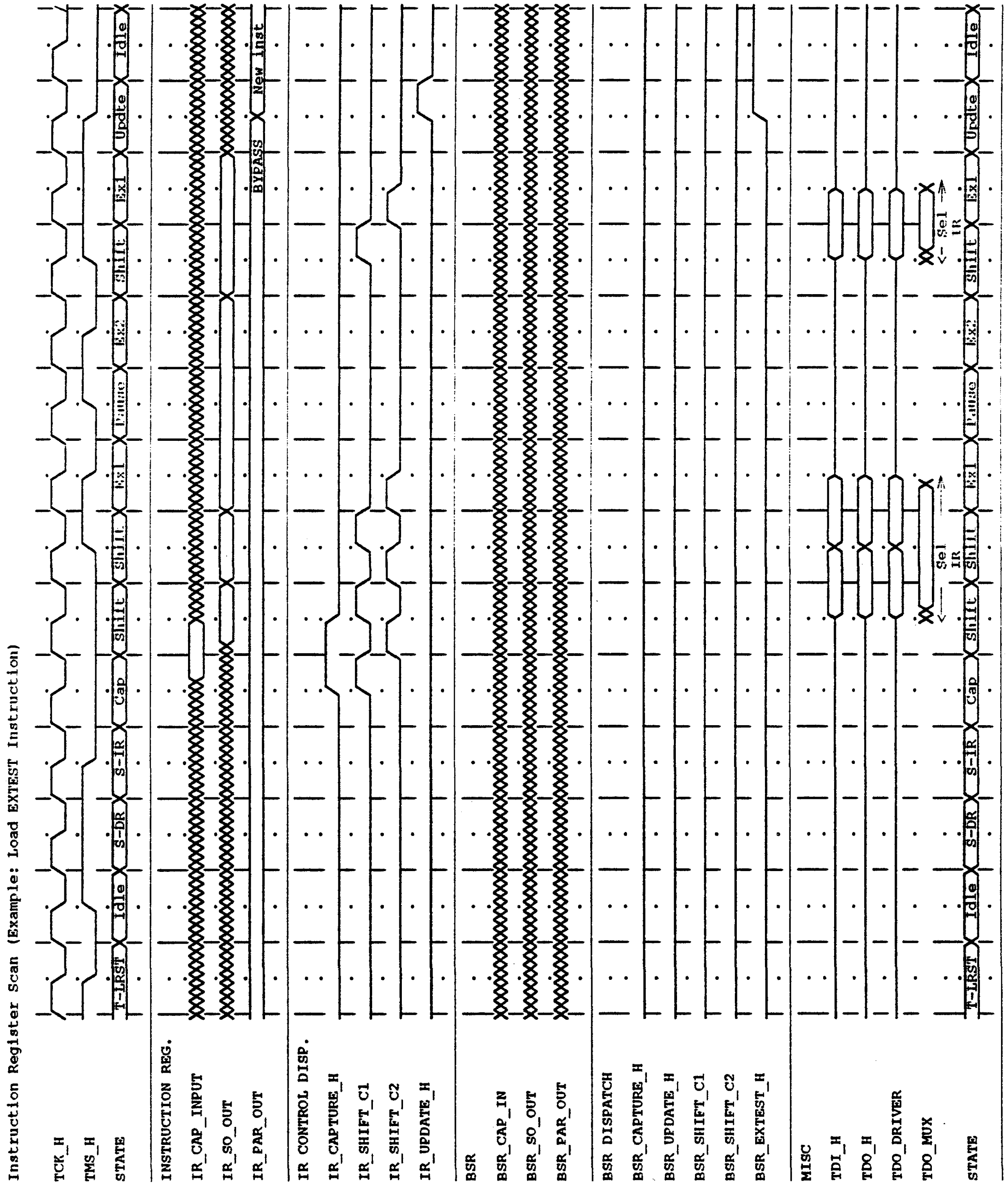
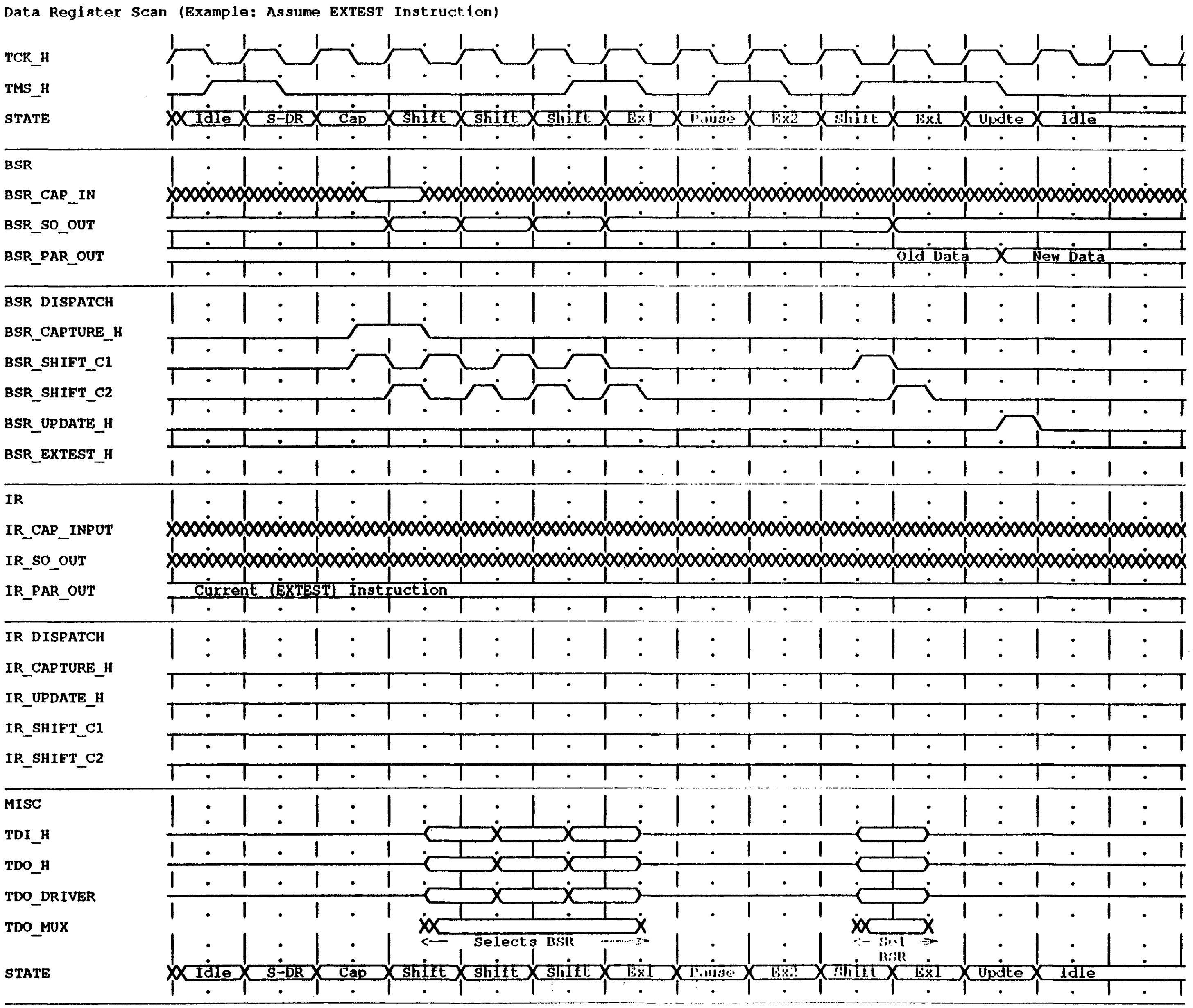


Figure 19–9: IEEE 1149.1 Logic Timing during DR-Scan Sequence



19.8.5 Initialization

The TAP Controller and the Instruction Register's output latches are initialized by **PP_CMD_H<0>**. When **PP_CMD_H<0>** pin is asserted low, the TAP controller is forced to enter the Test-Logic-Rest state and the IR is forced to BYPASS instruction.

During Test-Logic-Reset state, all JTAG logic, including boundary scan register, is in inactive state. That is, the chip performs normal system functions. The boundary scan logic is set to a passive sample (observe) mode. TAP controller leaves this state only when a JTAG test operation is desired and appropriate sequence is sent on TMS_H and TCK_H pins.

NOTE

> Note that PP_CMD_H< 0 > pin on NVAX CPU acts like a pseudo-TRST_L pin. Since this pin is internally pulled-up, **a system designer must make provision to assert the pin low, at least during the power-up operation**. This will keep all JTAG circuits inactive and allow system to wake up normally in system mode.

19.9 Boundary Scan Registers

The NVAX CPU chip's boundary scan register primarily facilitates interconnection test on module during module manufacturing and field service. Uses during other life cycle testing phases may also be possible.

The boundary scan register is a single shift register formed by boundary scan cells placed at most of the chip's signal pins. The register is accessed via the JTAG port's TDI_H and TDO_H pins. Its operation is controlled by the control dispatch received from the JTAG Port.

19.9.1 Boundary Scan Register Cells

The NVAX chip uses four main types of boundary scan cells.

in_bcell: Used on input-only pins. Figure 19–10 shows the block diagram. The bcell basically consists of 1-bit shift register. The cell supports Sample and Shift functions. The cell is used at input-only pins.

out_bcell: Used on output-only pins. Figure 19–11 show the block diagram. Besides the shift register, the cell has an output multiplexer. The cell supports the following functions: Sample, Shift, Drive outputs. The cell is used at miscellaneous output-only pins.

io_bcell: Used on bi-directional pins. Figure 19–12 show the block diagram. The cell is identical to the out_bcell cell except that it captures test data from the incoming data line. The cell supports Sample, Shift, Drive output functions. It is used at all I/O pins.

md_bcell: Used on certain special pins and internal signals. For example, this cell is used on **TS_WE_L, TS_OE_L, DR_WE_L, DR_OE_L** pins and on internal driver enable signals for bi-directional busses. Figure 19–13 show the block diagram of an md_bcell. The cell builds upon the out_bcell. It has a third output latch which holds data at output steady while a shift operation is in progress. The cell supports Sample, Shift, Drive output, and Hold output functions.

Figure 19–10: in_bcell Boundary Scan Cell

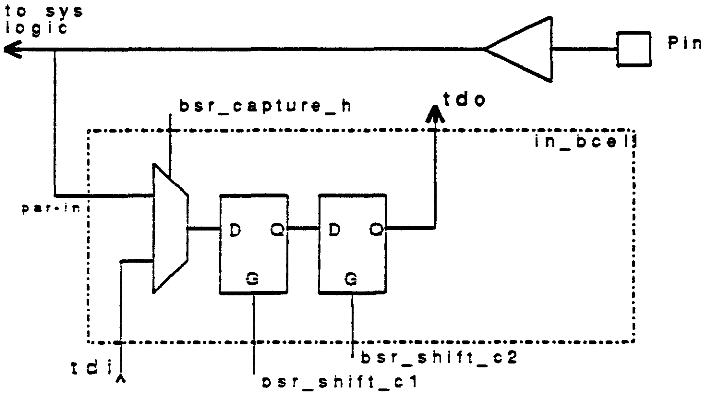


Figure 19–11: out_bcell Boundary Scan Cell

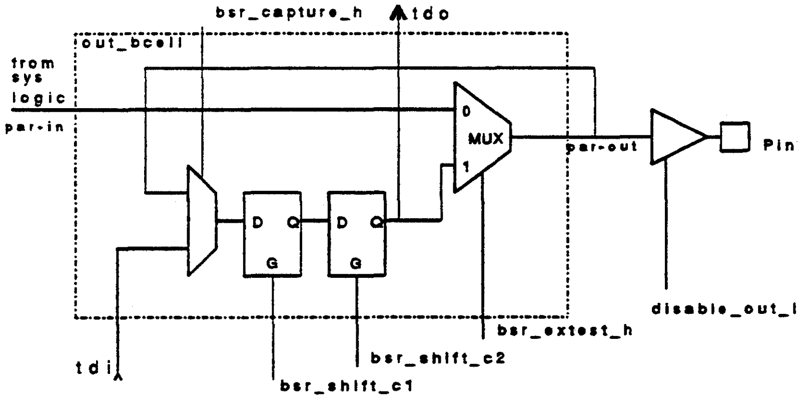


Figure 19–12: io_bcell Boundary Scan Cell

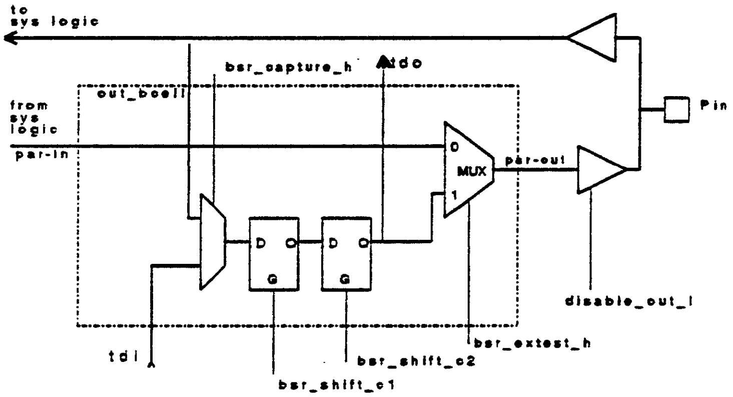
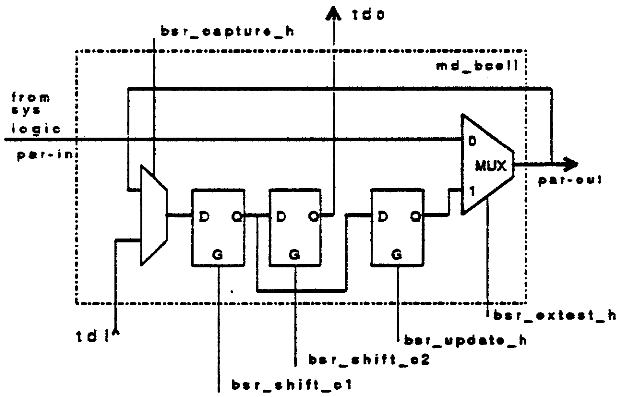


Figure 19–13: md_bcell Boundary Scan Cell



NOTE

Caution: In NVAX CPU chip, when Boundary Scan Register is shifting data in EXTEST mode (that is, when bsr_extest_h is asserted) the shifting of data is transferred to the pins and is visible to the other components connected to the pins.

Since the back-up cache interface pins are connected to RAMs which do not have boundary scan on them, the protection is provided by extra logic in the bcells on R/W bits. This is explained later.

19.9.2 Boundary Scan Register Organization

The boundary scan register on NVAX CPU chip is 243 bits long. Table 19–5 lists all the signal pins and the associated boundary scan register cell type. The pins are listed in the order of their connection from TDI_H pin to to TDO_H pin. Thus, cell on the internal signal on signal C_PAD_N%NDAL_OUT_DRV_H is closest to TDI_H pin and the cell on pin **CHIP_ID_H<11>** is farthest from the TDI_H pin. In an entry with more than one pin, the cell on the first pin is closer to the TDI_pin. On-chip fuses provide a means to program each die with a unique ID number which can be used to trace a packaged part back to the lot, wafer, and die location of origin for yield analysis. Although it is not part of the chip boundary, the twelve bit **CHIP_ID_H<11>** is connected to boundary scan chain so that the ID can be easily accessed through the JTAG port.

Table 19–5: Boundary Scan Register Organization

Signal Name	Count	Pin type	BSR Cell Type	Remarks
C_PAD_N%NDAL_OUT_DRV_H	1	Int signal	md_bcell	Int Signal
NDAL_H< 32:63 >	32	I/O, tri, 4 pts	io_bcell	
OSC_H, OSC_L	2	In	none	
OSC_TEST_H	1	In	none	
OSC_TC1_H, OSC_TC2_H	2	In	none	
PHI12_OUT_H, PHI23_OUT_H	2	Out, 1D, 4R	none	
PHI41_OUT_H, PHI34_OUT_H	2	Out, 1D, 4R	none	
SYS_RESET_L	1	Out	md_bcell	
ASYNC_RESET_L	1	In, 1D, 3R	in_bcell	
DISABLE_OUT_L	1	In, 1D, 3R	in_bcell	
TEST_STROBE_H	1	In, ptp	in_bcell	
TEST_DATA_H	1	In, ptp	in_bcell	
IRQ_L< 0:3 >	4	In, Op dr, 3D, 1R	in_bcell	
H_ERR_L, S_ERR_L	2	In, Op dr, 3D, 1R	in_bcell	
INT_TIM_L	1	In, ptp	in_bcell	
PWRFL_L, HALT_L	2	In, ptp	in_bcell	
MACHINE_CHECK_H	1	Out, ptp	out_bcell	
TEMP_H	1	Out	none	
PP_CMD_H< 0:2 >	3	In, pull-up	none	
PP_DATA_H< 0:11 >	12	Out	none	
TS_TAG_H< 17:31 >	15	I/O, tri 7 pts	io_bcell	
TS_ECC_H< 0:5 >	6	I/O, tri 7 pts	io_bcell	
TS_OWNED_H, TS_VALID_H	2	I/O, tri, 7 pts	io_bcell	
C_PAD_T%EN_TS_DRV_H	1	Int. signal	md_bcell	Int signal
TS_INDEX_H< 5:20 >	16	Out, 6 pts	out_bcell	

Table 19–5 (Cont.): Boundary Scan Register Organization

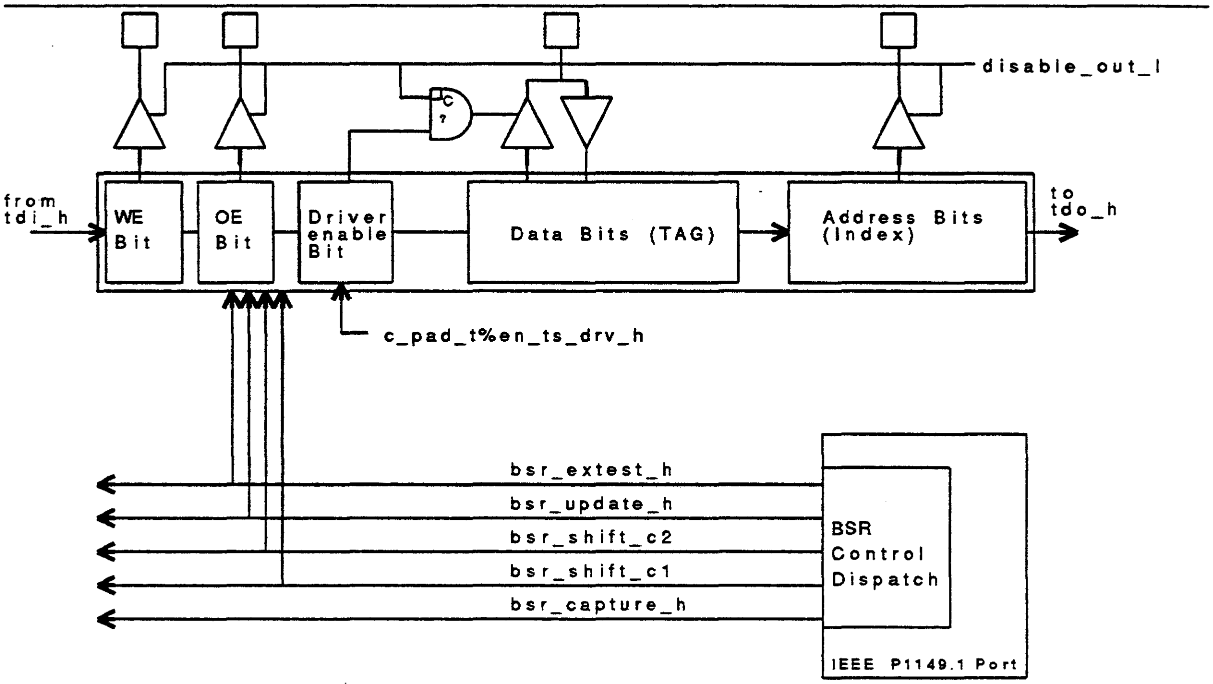
Signal Name	Count	Pin type	BSR Cell Type	Remarks
TS_OE_L, TS_WE_L	2	Out, 6 pts	md_bcell	
DR_INDEX_H< 3:20 >	18	Out, 8 pts	out_bcell	
DR_OE_L, DR_WE_L	2	Out, 8 pts	md_bcell	
C_PAD_D%EN_DR_DRV_H	1	Internal sig	md_bcell	Int Signal
DR_DATA_H< 0:23 >	24	I/O, tri, 19 pts	io_bcell	
DR_ECC_H< 0:7 >	8	I/O, tri, 19 pts	io_bcell	
DR_DATA_H< 24:63>	40	I/O, tri, 19 pts	io_bcell	
CPU_WB_ONLY_L	1	In, Op dr	in_bcell	
ACK_L	1	I/O, Op dr, 4 pts	io_bcell	
CPU_SUPRESS_L	1	Out, ptp	out_bcell	
CPU_HOLD_L	1	Out, ptp	out_bcell	
CPU_REQ_L	1	Out, ptp	out_bcell	
CPU_GRANT_L	1	In, ptp	in_bcell	
CMD_H< 0:3 >	4	I/O, tri, 4 pts	io_bcell	
ID_H< 0:2 >	3	I/O, tri, 4 pts	io_bcell	
PARITY_H< 0:2 >	3	I/O, tri, 4 pts	io_bcell	
NDAL_H< 0:31 >	32	I/O, tri, 4 pts	io_bcell	
CHIP_ID_H<0:11>	12	Int signal	in_bcell	Int Signal
PHI12_IN_H, PHI23_IN_H	2	In, 1D, 4R	none	
PHI41_IN_H, PHI34_IN_H	2	In, 1D, 4R	none	
TMS_H	1	In, pull-up	none	
TCK_H	1	In, pull-down	none	
TDO_H	1	Out,tri, 2D	none	
TDI_H	1	In, ptp, pull-up	none	

Some of the boundary scan register cells in NVAX are grouped together to form sections. A section is simply a collection of pins that are identical in nature and have identical boundary scan cells on them. A section is generally controlled and operated identically during certain test modes. The pins in a section may also be logically related and may be located physically together. Some such sections are described below.

BSR at TAG Store Interface

The boundary scan register at TAG Store interface consists of 4 sections: WE/OE bits, a driver enable bit on C_PAD_T%EN_TS_DRV_H signal, 23 data bits (tag, ECC and own), and 16 address (index) bits. Figure 19–14 shows the block diagram. The boundary scan cell type used in each segment is listed in Table 19–5. *(The figure does not show the actual order of connection.)*

Figure 19–14: Boundary Scan Register at TAG Store Interface



The following are some specific requirements.

WE/OE Bits: The WE/OE Bits use md_bcells with additional logic to allow proper operation of RAMs during interconnection testing. When bsr_extest_h is asserted, the test data is injected on pins is as follows:

- **TS_WE_L bit**: Data injected is the logical OR of the value stored in the md_bcell's output latch and the complement of bsr_update_h signal.
- **TS_OE_L bit**: Data injected is the logical OR of the value stored in the md_bcell's output latch and the complement of bsr_capture_h signal.

Idea is to assert these two signals appropriately in a non-overlapping manner and only when the boundary scan is not shifting the data. This enhancement allows the test operation to meet the timing constraints in accessing RAMs. (See reference [4].) It also protects RAM interface from the shifting data pattern.

BSR at Data RAM Interface

The BSR section at Data RAM interface also consists of 4 segments: WE/OE bits, a driver enable bit on C_PAD_D%EN_DR_DRV_H) signal, 72 Data bits, and 18 Index bits. The block diagram and operation of BSR at Data RAM interface are exactly same as the BSR at TAG Store interface.

BSR at NDAL Interface

The BSR section at NDAL data interface has a driver enable bit on the internal signal C_PAD_N%NDAL_OUT_DRV_H. It allows the drivers on bi-directional NDAL pins to be controlled by JTAG during testing.

19.10 Internal Scan Register and LFSR Reducer

NVAX CPU chip has several internal nodes observable via internal scan registers. This observability facilitates chip debug. Some internal scan register sections turned into LFSR Reducers to enhance fault coverage and reduce test vectors during chip manufacturing tests.

19.10.1 Internal Scan Register Cells

Figure 19–15 shows the block diagrams of two types of cells used in NVAX. ISR cell is used for Scan-only registers and ISL is used for implementing Scan-cum-LFSR registers.

Figure 19–15: Cells for Internal Scan Registers

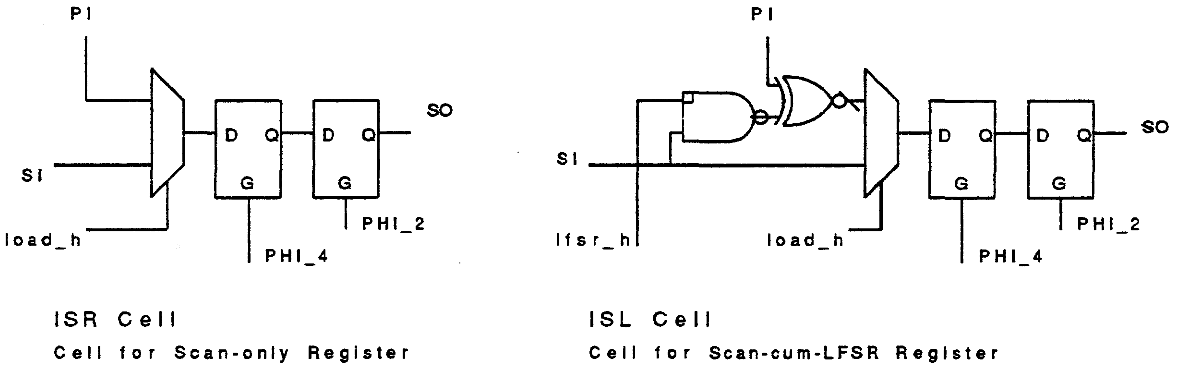
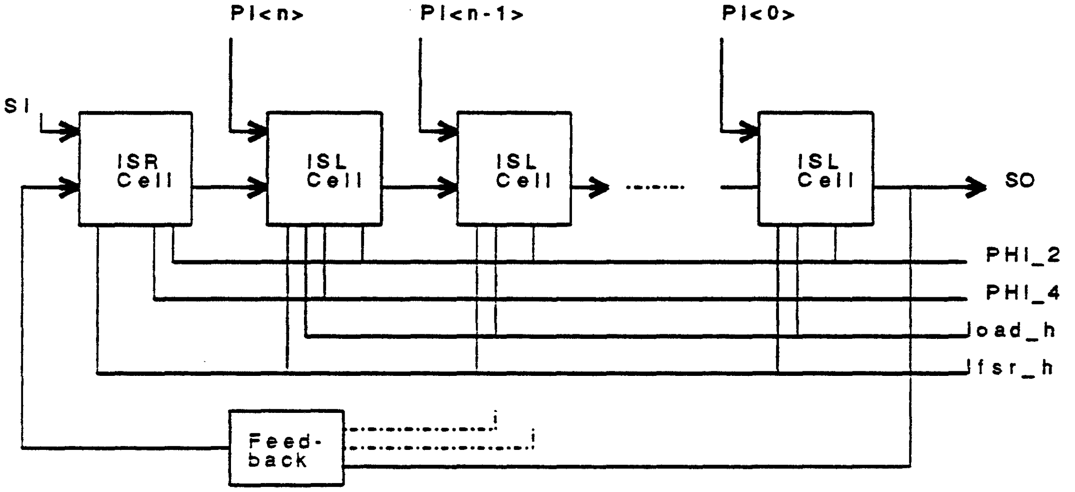


Figure 19–16 shows how an LFSR is constructed by using ISL cells and an ISR. The ISR cell used in the left-most bit position represents a dummy bit. The cell provides the multiplexer function required to enable feedback during LFSR operation. (Note that this cell can be replaced by an ordinary multiplexer. The feedback taps for the LFSRs are based on *primitive characteristic polynomial*. (The actual taps used will be documented in respective box chapters when LFSR size and other constraints are known.)

Internal Scan register's operations are controlled by internal NVAX clocks and by two signals received from the parallel port as follows:

— **PHI_4_H** and **PHI_2_H** are internal NVAX clocks. The **PHI_4_H** loads the master and **PHI_2_H** loads the slave.

Figure 19–16: An ISR section turned into LFSR



- When ISR_LOAD_H is asserted high the master latches in ISR and ISL cells capture/observe data from internal signals. When ISR_LOAD_H is asserted low the internal scan register shifts data. Note that the shift occurs independent of assertion on ISR_LFSR_H. ISR_LOAD_H is latched in phase PHI_3 before using it to control the ISRs.
- When both ISR_LFSR_H and ISR_LOAD_H are asserted high, the internal scan register sections containing ISL cells operate as LFSRs to and compress data. ISR_LFSR_H is a;so latched in phase PHI_3 before using it to control the internal LFSRs.

19.10.2 Internal Scan Register Organization

The Internal Scan Registers are divided into 2 groups: ISR1 and ISR2. The ISR1 consists of the scan register on the control store. It is used for patching the control store as well as reading out the control store during testing.

ISR2 consists of all the other internal scan registers. Specific nodes included on the internal scan registers are listed in individual box chapters under their testability sections. The individual box scan registers are chained together, and are shifted out in the following order: Ibox, Ebox, Mbox, Cbox.

Both ISR1 and ISR2 operate at the internal clock rate. However, they are read out at the parallel port at NDAL clock rate. See Section 19.4.1 for details of ISR1 and ISR2 operation.

19.11 Output Pin Tri-state Control

NVAX CPU chip has a dedicated pin **disable_out_l**. When asserted low, the CPU chip tri-states output drivers on all output-only and bi-directional pins, except those listed below. When asserted, the pin also forces internally a reset to the NVAX CPU chip.

The only exceptions are the **TDO_H** pin and NDAL clock output pins which are not tristated by the **disable_out_l** pin. Not tristating clock output pins has been approved by the stage-1 module test engineers.

Leaving out the **TDO_H** pin allows the JTAG circuits to operate while chip tristate is in effect. This affords additional flexibility for the module manufacturing test. For example, during the interconnection test, the NVAX outputs may be allowed to drive only during the CAPTURE-DR state and kept in tristate in all other states. This can eliminates the effect of shifting patterns, as well as drastically reduces the duration of time for which the drivers may see an interconnect short fault.

The single pin tristate function is used only during testing.

Note that the drivers on bi-directional I/O pins are also tristated by internal Cbox logic during RESET and by the boundary scan register during the interconnection test (EXTEST mode).. The order of precedence is as follows: DISABLE_OUT_L, Boundary scan register, and the Cbox logic.

19.12 Operating Speed of Test Logic

The IEEE 1149.1 Port and the boundary scan register are designed to be operable in the range 0 to 10 MHz at least. Internal scan registers operate at internal clock rate. A higher speed of 10 MHz (instead of 5 MHz) has been set to make the boundary scan register usable during the wafer probe testing.

NOTE

The JTAG circuitry design must account for the fact that TCK_H will not be driven in the running system.

References

1. "NTUG's Testability Document V1.0," NVAX Testability User's Group, December 1988.
2. "Common Test Architecture: Adaptations and Compatible Applications of IEEE P1149.1 Specification, Revision 1.0," Semiconductor Design & Engineering/Advanced Test Technology Group, February 1990.
3. IEEE Standard 1149.1-1990: "IEEE Standard Test Access Port and Boundary-Scan Architecture draft D3." January 1989,
4. "Testing connections to non-JTAG Static RAMs with JTAG Boundary Scan," D. K. Bhavsar, *DEC Internal Report*, December 1989.

19.13 Revision History

Table 19–6: Revision History

Who	When	Description of change
Dilip Bhavsar	06-Mar-1989	Release for external review.
Dilip Bhavsar	18-Jul-1989	First Update of specific details.
Dilip Bhavsar	15-Jan-1990	3rd Release.
Dilip Bhavsar	16-Mar-1990	Spec error in WE and OE bcells corrected.
Dilip Bhavsar	21-May-1990	(3.2) bcell on SYS_REST pin changed to md_bcell. IR and speed spec updated. Clock tristating removed. Serial port for PCache introduced.
Dilip Bhavsar	14-June-1990	(3.3) Parallel port modes changed. JTAG Reset added. Box controllability removed.
Dilip Bhavsar	03-July-1990	(3.4) JTAG Reset finalized. bcell and other figures updated to reflect actual implementation. Timing on JTAG control signals changed to be consistent with edge-trigger design. Pins listed in order of their connection in BSR.
John F. Brown	03-Jul-1990	Serial PCache Port details added.
Dilip Bhavsar	30-Jul-1990	Reset actions by JTAG EXTEST instruction and DISABLE_OUT_L pin added. Timing diagrams added. Parallel Port operation details added. ISR/ISL clocking changed to PHI_4 (master) and Phi_2 (slave). Final Edits for Rev 3.4. (See NITS 314, 330, 337, 351, 360)
John F. Brown	23-Aug-90	Timing diagram for Serial P-cache port added
Dilip Bhavsar	28-Sep-90	Rev 3.5. PPort timing changed (NITS # 385). More description for PPort operation.Also, the boundary scan order updated to reflect implementation.
John F. Brown	20-Feb-91	Rev 3.6. Updates for spec release: PPort fields & scan chain order

3

3.465
.175
3.640
3.640
.050
3.590 = 3.6

3.465
.175

3.640

175 mV p-p [illegible]
10% decoupling
15 - nanofarad
101 - nanofarad

Steve T. - doesn't know where sin is

Figure 15–10 (Cont.): Cause Parse Tree for Soft Error Interrupts

```
1    2
v    v
|    |  S_CEFSTS<RDE>
|    +----+ (select one)
|    |    |
|    |    |  S_CEFSTS<OREAD>
|    |    +----+ (select one)
|    |    |    |
|    |    |    |  S_CEFSTS<WRITE>
|    |    |    |  AND NOT S_CEFSTS<TO_MBOX>
|    |    |    +----+ (select one)
|    |    |    |    |
|    |    |    |    |  S_CEFSTS<REQ_FILL_DONE>
|    |    |    |    +-----------------------------> Inconsistent status (should cause hard error interrupt)
|    |    |    |    |                               (Section 15.8.1.22)
|    |    |    |    |  otherwise
|    |    |    |    +----+ (select one)
|    |    |    |         |
|    |    |    |         |  S_CEFSTS<LOST_ERR>
|    |    |    |         +------------------------> Multiple errors in context of PTE read error
|    |    |    |         |                          (Section 15.8.1.18.5)
|    |    |    |         |  otherwise
|    |    |    |         +------------------------> D-stream NDAL ownership read for Mbox write read data
|    |    |    |                                    error before write data merged with fill data (Section15.8.1.14)
|    |    |    |  S_CEFSTS<TO_MBOX>
|    |    |    +----+ (select one)
|    |    |    |    |
|    |    |    |    |  S_CEFSTS<LOST_ERR>
|    |    |    |    +-----------------------------> Multiple errors in context of PTE read error
|    |    |    |    |                               (Section 15.8.1.18.5)
|    |    |    |    |  otherwise
|    |    |    |    +-----------------------------> D-stream NDAL ownership read read data error
|    |    |    |                                    (modify operand or read-lock) (Section 15.8.1.14)
|    |    |    |  otherwise
|    |    |    +----------------------------------> Inconsistent status (either WRITE or TO_MBOX, but not both,
|    |    |                                         should be set) (Section 15.8.1.22)
|    |    |  otherwise
|    |    +----+
|    |         |
|    |         |  S_CEFSTS<IREAD>
|    |         +----+ (select one)
|    |         |    |
|    |         |    |  S_CEFSTS<LOST_ERR>
|    |         |    +-----------------------------> Multiple errors in context of PTE read error
|    |         |    |                               (Section 15.8.1.18.5)
|    |         |    |  otherwise
|    |         |    +-----------------------------> I-stream NDAL read read data error
|    |         |                                    (Section 15.8.1.14)
|    |         |  S_CEFSTS<TO_MBOX>
|    |         +----------------------------------> D-stream NDAL read read data error (PTE read)
|    |         |                                    (Section 15.8.1.18.3)
|    |         |  otherwise
|    |         +----------------------------------> Inconsistent status (TO_MBOX should be set)
|    |                                              (Section 15.8.1.22)
|    |  otherwise
|    +--------------------------------------------> Inconsistent status (either CEFSTS<RDE> or CEFSTS<TIMEOUT>
|                                                   should be set or, if CEFSTS<UNEXPECTED_FILL> is set, it
|                                                   should cause a hard error interrupt) (Section 15.8.1.22)
v
1
```

Figure 15–10 Cont'd on next page

[1] At least one potential PTE cause must be found or the status is inconsistent (see Section 15.8.1.22). Some of the outcomes indicate a potential soft error interrupt cause which is not a potential PTE read error cause. These errors should be treated separately.

Figure 15–10 (Cont.): Cause Parse Tree for Soft Error Interrupts

```
1
v
|  S_NESTS<NOACK> AND
|  S_PCSTS<PTE_ER>[1]
+----+ (select one)
|    |
|    |  S_NEOCMD<CMD>=IREAD
|    +----+ (select one)
|    |    |
|    |    |  S_NESTS<LOST_OERR>
|    |    +---------------------------------------> Multiple errors in context of PTE read error
|    |    |                                         (Section 15.8.1.18.5)
|    |    |  otherwise
|    |    +---------------------------------------> Unacknowledged I-stream NDAL read (Section 15.8.1.16)
|    |
|    |  S_NEOCMD<CMD>=DREAD
|    +--------------------------------------------> Unacknowledged D-stream NDAL read (PTE read)
|    |                                              (Section 15.8.1.18.4)
|    |  S_NEOCMD<CMD>=OREAD
|    +----+ (select one)
|    |    |
|    |    |  S_NESTS<LOST_OERR>
|    |    +---------------------------------------> Multiple errors in context of PTE read error
|    |    |                                         (Section 15.8.1.18.5)
|    |    |  otherwise
|    |    +---------------------------------------> Unacknowledged D-stream NDAL read (modify operand or re
|    |                                              (Section 15.8.1.16)
|    |  S_NEOCMD<CMD>=WRITE or WDISOWN
|    +--------------------------------------------> Inconsistent status (should cause hard error interrupt)
|    |                                              (Section 15.8.1.22)
|    |  otherwise
|    +--------------------------------------------> Inconsistent status (invalid command in NEOCMD<CMD>)
|                                                   (Section 15.8.1.22)
|  S_NESTS<PERR>
+----+ (select one)
|    |
|    |  S_NESTS<INCON_PERR>
|    +--------------------------------------------> NDAL inconsistent parity error
|    |                                              (Section 15.8.1.19)
|    |  otherwise
|    +--------------------------------------------> NDAL parity error (Section 15.8.1.19)
|
|  S_NESTS<LOST_PERR>
+-------------------------------------------------> Lost NDAL parity error or inconsistent parity error
|                                                   (Section 15.8.1.20)
|  (status consistent with soft error interrupt
|  in system environment error registers)
+-------------------------------------------------> Soft error interrupt from system environment
|                                                   (Section 15.8.1.21)
|  none of the above
+-------------------------------------------------> Inconsistent status (possible machine check or hard err
                                                    interrupt during soft error interrupt processing)
                                                    (Section 15.8.1.22)
```

Figure 15–10 Cont'd on next page

[1] At least one potential PTE cause must be found or the status is inconsistent (see Section 15.8.1.22). Some of the outcome: indicate a potential soft error interrupt cause which is not a potential PTE read error cause. These errors should b treated separately.

Figure 15–10 (Cont.): Cause Parse Tree for Soft Error Interrupts

```
Notation:
    (select one)              - Exactly one case must be true. If zero or more than one is
                                true, the status is inconsistent.
    (select all)              - More than one case may be true.
    (select all, at least one) - All the cases are possible causes of a soft error interrupt.
                                More than one may be true. At least one must be true or the status
                                is inconsistent.  A case is not considered true if it evaluates to
                                "Not a soft error interrupt cause".
    otherwise                 - fall-through case for (select one) if no other case is true.
    none of the above         - fall-through case for (select all) or (select all, at least one)
                                if no other case is true.
```

15.8.1.1 VIC Parity Errors

Description: A parity error was detected in the VIC tag or data store in the Ibox.

VIC Data Parity Errors: A parity error occurred in the data portion of the VIC.

VIC Tag Parity Errors: A parity error occurred in the tag portion of the VIC.

In all cases, the quadword virtual address of the error is in S_VMAR.

Recovery procedures: To recover, disable and flush the VIC by re-writing all the tags (using the procedure in Section 15.3.3.1.1.1). Also, clear ICSR<LOCK>.

15.8.1.2 Pcache Parity Errors

Description: A parity error was detected in the Pcache. Either a tag parity error or a data parity error is reported, though tag parity errors in both the left and right banks may be reported simultaneously. The reference, whether it was a read or write, was passed to the Cbox as if the Pcache had missed. No data is lost. The Pcache is disabled because PCSTS<LOCK> is set.

S_PCADR contains the physical address of operation incurring the error. The address should not be in IO space. If it is, it is an inconsistent status (see Section 15.8.1.22).

Recovery procedures: Clear PCSTS<LOCK>. Flush the Pcache and initialize the Pcache tag store (see Section 15.3.3.1.1.1.2).

15.8.1.3 Bcache Tag Store Uncorrectable Errors

Description: An uncorrectable ECC error or an addressing error resulted from reading the Bcache tag store. The Bcache is in ETM. The hexaword physical address of the transaction incurring the error is in S_BCETIDX. (If the physical address is found to be in IO space, it is an inconsistent status. See Section 15.8.1.22.) S_BCETAG contains the actual tag data and check bits read during the failing access. Software may use the routine TAG_ECC_CHECK in Section 15.10 to check the tag data and determine the syndrome. The result of this check should give the result expected from S_BCETSTS<UNCORR,BAD_ADDR>.

It should never be the case that both S_BCETSTS<BAD_ADDR> and S_BCETSTS<UNCORR> are set. If they are, it is an inconsistent status (see Section 15.8.1.22).

For any normal Mbox command (i.e., not BCFLUSH), this error leads to a fill of the block whose tag had the error. This is because the Cbox converts uncorrectable tag store errors into misses and sends the associated reference to memory. For reads, the reference sent out is a read or an ownership read, and when the data returns it is loaded in the Bcache. For writes, an ownership read is sent, and when the data returns the write is merged with it and it is loaded in the Bcache. When the fill finishes successfully, the tag is updated (overwriting the bad tag). If the fill times out, the tag is not overwritten.

In some cases, this error leads to an NVAX CPU read timeout and/or a write timeout in memory. This occurs when the block was VALID-OWNED in the Bcache and is the same block that is being accessed by the failing operation. Errors resulting from these lost blocks are handled separately.

Write-unlocks are a special case. No tag lookup is done for write-unlocks unless the Bcache is in ETM. If the Bcache is in ETM, and the tag store error occurs for that transaction, the write-unlock is sent to memory.

Recovery procedure (all cases): Clear BCETSTS<LOCK>. If it is an addressing error, clear BCETSTS<BAD_ADDR>. Otherwise, clear BCETSTS<UNCORR>.

15.8.1.3.1 Case: BCETSTS<TS_CMD>=WUNLOCK

Recovery procedure: Write a INVALID tag with good ECC to the tag with the error (using the BCTAG access path). Then flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache). Software should prepare for another tag error during the Bcache flush by clearing BCETSTS of unrecoverable errors.

Restart conditions:

The Bcache was in ETM at the time the write-unlock arrived. The data is in memory may be corrupt and memory's ownership bit was cleared. Memory is corrupted at the location indicated by S_BCETIDX. Software must determine if the error is fatal to one process or the whole system and take appropriate action.

15.8.1.3.2 Case: BCETSTS<TS_CMD>=DREAD,IREAD,OREAD

Recovery procedure: Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache). Software should prepare for another tag error during the Bcache flush by clearing BCETSTS of unrecoverable errors. After flushing the Bcache, it is necessary to determine if any block is "lost". If a block's memory ownership bit is set and no writeback cache in the system has it owned, then the block is said to be lost. Use the procedure in Section 15.3.3.1.2.5. This procedure can result in finding no lost blocks, one lost block, or multiple lost blocks.

Restart conditions: If there is one lost block, it is not recoverable. Software must if the lost data was fatal to one process or the whole system and take appropriate action.

If multiple blocks are lost (this isn't expected), crash the system.

15.8.1.3.3 Case: BCETSTS<TS_CMD>=R_INVAL,O_INVAL,IPR_DEALLOCATE

Recovery procedure: Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache). Software should prepare for another tag error during the Bcache flush by clearing BCETSTS of unrecoverable errors. After flushing the Bcache, it is necessary to determine if any block is "lost". If a block's memory ownership bit is set and no writeback cache in the system has it owned, then the block is said to be lost. Use the procedure in Section 15.3.3.1.2.5. This procedure can result in finding no lost blocks, one lost block, or multiple lost blocks.

If exactly one block is lost, memory's owner ID information indicates this CPU, write a VALID-OWNED tag with the address of the lost block into the tag which had the error (using the BCTAG access means). Then flush this location to memory. An error could occur with this flush, in which case the data is not recoverable.

NOTE

> If memory does not store an owner ID with each block in a particular system, then this recovery method is not recommended. Instead, the data should be considered lost.

Restart conditions: If there is one lost block, and the repair procedure didn't incur an error, restart.

If the repair procedure was not successful, the data is not recoverable. Software must if the lost data was fatal to one process or the whole system and take appropriate action.

If multiple blocks are lost (this shouldn't result from one tag store error), crash the system.

15.8.1.4 Lost Bcache Tag Store Errors

Some number of unrecoverable Bcache tag store errors occurred and were not latched because BCETSTS already contained a report of an unrecoverable error. All unrecoverable tag store errors cause soft error interrupt, so this is definitely a cause of the soft error interrupt.

- Lost Bcache tag store errors may be caused by more than one operand prefetch to the same cache block.

The Bcache is in ETM.

Unrecoverable tag store errors can cause lost data by overwriting blocks in the Bcache.

Unrecoverable tag store errors in ETM on write-unlocks can cause corrupted memory data.

Recovery procedure: Clear BCETSTS<LOST_ERR>. Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache). Software should prepare for another tag error during the Bcache flush by clearing BCETSTS of unrecoverable errors.

Restart conditions: Lost write-unlock errors may have corrupted memory. Crash the system.

15.8.1.5 Bcache Tag Store Correctable ECC errors

Description: A correctable error occurred in accessing the Bcache tag store. The Bcache is not in ETM. S_BCETIDX contains the physical address of the error. (If the physical address is found to be in IO space, it is an inconsistent status. See Section 15.8.1.22.) (The index portion of S_BCETIDX indicates which tag store entry had the error.) S_BCETAG contains the actual tag data and check bits read during the failing access. Software may use the routine

TAG_ECC_CHECK in Section 15.10 to check the tag data and determine the syndrome. The result of this check should be a correctable single-bit error.

Recovery procedures: Clear BCETSTS<CORR>.

If the operation was anything but a tag lookup for an explicit IPR deallocate operation (i.e., BCFLUSH), software should flush that one location by writing the BCFLUSH IPR.

```
TBS (MTPR to (BCFLUSH + (S_BCETIDX & INDEX_MASK)))
```

This effectively scrubs the Bcache tag store location by invalidating it and forcing it to be written back if it is owned. This may be done without putting the Bcache in software ETM.

15.8.1.6 Lost Bcache Tag Store Correctable ECC errors

Description: A correctable error occurred in accessing the Bcache tag store, but it is lost because of an uncorrectable tag store error which also occurred.

Recovery procedures: Clear BCETSTS<CORR>.

The Bcache should be flushed (and it would be because of the uncorrectable error in any case). This effectively scrubs the Bcache tag store location by invalidating it.

15.8.1.7 Bcache Data RAM Correctable ECC Errors

Description: A correctable error occurred in accessing the Bcache data RAM. The Bcache is not in ETM. S_BCEDIDX contains the cache index of the error, and S_BCEDECC contains the syndrome calculated by the ECC logic. It is not possible to reliably determine the physical address of the error, since the Bcache is not in ETM and therefore the block can be overwritten at any time after the error.

Recovery procedures: Clear BCEDSTS<CORR>.

If the operation was a read (S_BCEDSTS<DR_CMD>=DREAD or IREAD), software should flush that one location using the BCFLUSH IPR.

```
TBS (MTPR to (BCFLUSH_BASE + (BCEDIDX & INDEX_MASK)))
```

This effectively scrubs the Bcache data RAM location by invalidating it and forcing it to be written back if it is owned. This may be done without putting the Bcache in software ETM.

15.8.1.8 Lost Bcache Data RAM Correctable ECC Errors

Description: A correctable error occurred in accessing the Bcache data RAM, but it is lost because of an uncorrectable data RAM error which also occurred. The address and syndrome of the error are not known.

Recovery procedures: Clear BCEDSTS<CORR>.

The Bcache should be flushed (and it would be because of the uncorrectable error in any case). This effectively scrubs the Bcache data RAM location by invalidating it and forcing it to be written back if it is owned.

15.8.1.9 Bcache Data RAM Uncorrectable ECC Errors and Addressing Errors on I-Stream or D-Stream Reads

Description (addressing error): A Bcache addressing error was detected by the Cbox in an I-stream or D-stream read during a Bcache hit. Addressing errors are the result of a mismatch between the address the Cbox drives to the RAMs for a read access and the address used to write that location. A multiple bit data error can appear to be addressing error, though it is extremely unlikely.

Description (uncorrectable ECC error): A Bcache uncorrectable ECC error was detected by the Cbox in an I-stream or D-stream read during a Bcache hit. Uncorrectable data errors are the result of a multiple bit error in the data read from the Bcache. An addressing error with a single bit data error will appear as an uncorrectable data error.

Description (both cases): The Bcache in in ETM. S_BCEDIDX contains the cache index of the error, and S_BCEDECC contains the syndrome calculated by the ECC logic. The physical address of the reference can be found by reading the tag for the data block (using the procedure in Section 15.3.3.1.2.4). (If the physical address is found to be in IO space, it is an inconsistent status. See Section 15.8.1.22.)

If the block's tag is found to contain an ECC error, then the address can not be determined.

It should never be the case that both S_BCEDSTS<BAD_ADDR> and S_BCEDSTS<UNCORR> are set. If they are, it is an inconsistent status (see Section 15.8.1.22).

Recovery procedures: To recover, clear BCEDSTS<LOCK>. Also, if it is an addressing error, clear BCEDSTS<BAD_ADDR>. Otherwise, clear BCEDSTS<UNCORR>.

Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache). If the data is owned by the Bcache and if the error repeats itself (is not transient), then a writeback error will result from the flush procedure. Software should prepare for this by clearing NESTS and BCEDSTS errors.

Restart Conditions: If a writeback error occurs in the Bcache flush, then the data is presumed to be unrecoverable. See the next section for a description of handling an error in a writeback. Software must determine if the error is fatal to one process or the whole system and take appropriate action.

If the address of the error in the flush is not the same as that of the original error, this is a multiple error case in the data RAMs and is a serious failure. Crash the system.

15.8.1.10 Bcache Data RAM Uncorrectable ECC Errors and Addressing Errors on Writebacks

Description (addressing error): A Bcache addressing error was detected by the Cbox in an writeback. Addressing errors are the result of a mismatch between the address the Cbox drives to the RAMs for a read access and the address used to write that location. A multiple bit data error can appear to be addressing error, though it is extremely unlikely. The NDAL WDATA cycle was converted to a BADWDATA cycle. Memory should have tagged the location as bad and unreadable by an implementation specific mechanism.

Description (uncorrectable ECC error): A Bcache uncorrectable ECC error was detected by the Cbox in an writeback. Uncorrectable data errors are the result of a multiple bit error in the data read from the Bcache. An addressing error with a single bit data error will appear as an uncorrectable data error. The NDAL WDATA cycle was converted to a BADWDATA cycle.

Memory should have tagged the location as bad and unreadable by an implementation specific mechanism.

Description (both cases): The Bcache in in ETM. S_NESTS<BADWDATA> should be set. If it isn't, and S_NESTS<LOST_OERR> and S_NESTS<NOACK> aren't set, then the writeback which incurred the error is still in the writeback queue in the BIU. Software should force the writeback queue to be drained (causing the second error event to occur) by reading from the CWB register.

```
MFPR  #PR19$_CWB,R0
```

After this, NESTS, NEOADR, and NEOCMD should be captured again.

If S_NESTS<BADWDATA> is set, then S_NEOADR contains the physical address of the lost writeback data. (If the physical address is found to be in IO space, it is an inconsistent status. See Section 15.8.1.22.)

If S_NESTS<BADWDATA> isn't set but S_NESTS<LOST_OERR> is, then the address of the lost writeback data is not available.

If after draining the writeback queue, S_NESTS<BADWDATA> isn't set, then an inconsistency exists (see Section 15.8.1.22).

It should never be the case that both S_BCEDSTS<BAD_ADDR> and S_BCEDSTS<UNCORR> are set. If they are, it is an inconsistent status (see Section 15.8.1.22).

Recovery procedures: To recover, clear BCEDSTS<LOCK> and NESTS<BADWDATA>, if it is set. If it is an addressing error, clear BCEDSTS<BAD_ADDR>, otherwise clear BCEDSTS<UNCORR>. Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache). Then use the system specific memory repair procedure to undo the tagged-bad data in memory (see Section 15.3.3.1.2.2.3).

NOTE

When clearing the tagged-bad data state of memory, software must first ensure that no more accesses to the block can occur. Otherwise there is the danger that some process on some other processor or a DMA IO device will see incorrect data and not detect an error.

Restart Conditions: The data is lost, software must determine if the error is fatal to one process or the whole system and take appropriate action. If the address of the lost data could not be determined, crash the system.

15.8.1.11 Lost Bcache Data RAM Errors With Possible Lost Writebacks

Description: Lost Bcache data RAM errors which cause only a soft error interrupt (when S_NESTS indicates the possibility of a lost writeback error) indicate that data errors occurred on reads or writebacks, but no new write data was lost. S_NESTS reports the writeback error, unless multiple NDAL output errors have occurred.

The Bcache in in ETM.

Lost Bcache data RAM errors of this kind can be caused by an operand prefetch from a Bcache block followed by a write to the same block.

If S_NESTS<BADWDATA> is set, then S_NEOADR contains the physical address of a writeback. (If the physical address is found to be in IO space, it is an inconsistent status. See Section 15.8.1.22.)

Recovery procedures: To recover, clear BCEDSTS<LOST_OERR>. Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache). Writeback errors may occur during the flush. Software should prepare for this by clearing NESTS and BCEDSTS errors.

If S_NESTS<BADWDATA> is set, clear NESTS<BADWDATA>. Use the system specific memory repair procedure to undo the tagged-bad data in memory (see Section 15.3.3.1.2.2.3) (the Bcache must be flushed before this repair procedure).

NOTE

> When clearing the tagged-bad data state of memory, software must first ensure that no more accesses to the block can occur. Otherwise there is the danger that some process on some other processor or a DMA IO device will see incorrect data and not detect an error.

Restart condition (S_NESTS<LOST_OERR> set): There is no way to determine how many writebacks failed. They all should have gone to memory with BADWDATA cycles, where memory would have them marked as tagged-bad data. So an unknown block may be tagged-bad in memory. If so, the next access to that block could come from the system itself, even if it "belonged" only to one process. This will cause the system to crash. But there is a chance that the next access will come from a user process. This would allow the system to stay up, though that process would have to be deleted.

If the system's implementation of tagged-bad data is not reliable (see Section 15.11.1, Note On Tagged-Bad Data Mechanisms), software should crash the system. If it is reliable, restart.

Restart condition (S_NESTS<LOST_OERR> not set):

The writeback data is lost but the address is known. Software must determine if the error is fatal to one process or the whole system and take appropriate action.

15.8.1.12 Lost Bcache Data RAM Errors Without Lost Writebacks

Description: Lost Bcache data RAM errors which cause only a soft error interrupt (when S_NESTS indicates no possibility of writeback error) indicate that data errors occurred on reads. No write data was lost.

Lost Bcache data RAM errors may be caused by more than one operand prefetch to the same cache block.

The Bcache in in ETM.

Recovery procedures: To recover, clear BCEDSTS<LOST_OERR>. Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache). Writeback errors may occur during the flush. Software should prepare for this by clearing NESTS and BCEDSTS errors.

Restart condition: Only reads from the Bcache failed. Restart is possible unless any error encountered during Bcache flush is fatal.

15.8.1.13 NDAL I-Stream or D-Stream Read or D-Stream Ownership Read Timeout Errors

Description: An I-stream or D-stream read or D-stream ownership read timed out before all the fill quadwords were received. This is not an accepted means for a system environment to notify the NVAX CPU of "non-existent memory or IO location". The error could be caused by an error in the system environment or an NDAL parity error on the returned data. It also could be caused by some previous error in the system environment or this CPU which leaves a cache block marked as owned in memory and not marked as owned in any cache in the system. S_CEFSTS<COUNT> indicates the number of quadwords received before the error. (S_CEFSTS<COUNT> should always be 11 (binary) if the address is in IO space. If the address is in memory space, S_CEFSTS<COUNT> indicates the number of quadwords received.) The physical address is in S_CEFADR.

I-stream or D-stream read: The Bcache is not in ETM.

D-stream ownership read: The Bcache is in ETM. No write data has been merged with the returning fills.

The address should not be in IO space. If it is, it is an inconsistent status (see Section 15.8.1.22).

If the ownership read was for an Mbox write, the write was sent on the NDAL after the OREAD timed out.

If the ownership read was for a read-lock, the corresponding write-unlock should have been received from the Ebox. The write-unlock is sent as a quadword WDISOWN by the Cbox, so no memory location is left owned. (If the error was on the requested quadword, a machine check would definitely have resulted. If a separate error prevents the write-unlock, that will be reported either in other error registers.)

Recovery procedures (all cases): Clear CEFSTS<LOCK, TIMEOUT>.

Additional Recovery procedures for D-stream ownership read (S_CEFSTS<WRITE> set): Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache).

Depending on the system environment, memory may have set its ownership bit for this block. If so the write data must have been lost, and a hard error interrupt is expected. Use the system dependent procedure for reseting the ownership bit in memory.

If memory would not have set its ownership bit for this block, memory's state may be correct and up to date.

Additional Recovery procedures for D-stream ownership read (S_CEFSTS<WRITE> not set): Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache).

Depending on the system environment, memory may have set its ownership bit for this block. The data in memory is presumably still good. The Bcache block is marked invalid in the Bcache tag store. However, if the error occurred on a read-lock, the corresponding write-unlock should have occurred and it will have cleared the ownership bit for this block.

If S_CEFSTS<COUNT> is greater than 0, then part of the data also is in the Bcache. In general, it is not possible to determine which quadwords are valid. However, if S_CEFSTS<REQ_FILL_DONE> is set, then the quadword in the Bcache block pointed to by S_CEFADR is valid (except in the case of a read-lock, but the data shouldn't be needed for memory repair in that case).

If S_CEFSTS<COUNT> is greater than 0, and the address in S_CEFADR is not in IO space, then the block was not owned before the operation began. In this case, use the system dependent procedures (see Section 15.3.3.1.2.2.1) to determine if memory's ownership bit is set and this CPU owns the block. If so, use the system specific procedure (see Section 15.3.3.1.2.2.2) to reset it. In some systems (the XMI2 for example) this may require a quadword of correct data be written to memory to reset the ownership bit. Section 15.3.3.1.2.3 describes procedures for extracting data from the Bcache data RAMs in this case.

If memory's ownership bit was left set as a result of this error and no non-destructive procedure exists for restoring it, then the hexaword block is lost.

Restart condition: Restart if the memory state repair procedure is successful or no repair is called for, no data is lost, and the address is not in IO space. If the hexaword block could not be repaired or data is lost, software must determine if the error is fatal to one process or the whole system and take appropriate action.

Post Restart Recovery: If the same fill error recurs on restart, then the block is probably "lost".[1] Software must determine if the error is fatal to one process or the whole system and take appropriate action. (If it is fatal only to one process, use the system dependent procedure for reseting memory's ownership bit.)

NOTE

> It may be appropriate in this case to first cause each CPU in the system to flush its Bcache, and then restart once more.

NOTE

> It may be that another error (such as an uncorrectable tag store error on a coherence request) will be repaired by the soft error interrupt handler before the restart actually occurs, fortuitously repairing the cause of the fill error.

15.8.1.14 NDAL I-Stream or D-Stream Read or D-Stream Ownership Read Data Errors

Description: An I-stream or D-stream read or D-stream ownership read terminated with an RDE (read data error) NDAL cycle before all the fill quadwords were received. If S_CEFSTS<COUNT> is 0 or the address is an IO space address, this is an accepted means for a system environment to notify the NVAX CPU of "non-existent memory or IO location". Otherwise, the error could be caused by an error in the system environment. It also could be caused by some previous error in the system environment or this CPU which leaves a cache block marked as owned in memory and not marked as owned in any cache in the system.

S_CEFSTS<COUNT> indicates the number of quadwords received before the error. (S_CEFSTS<COUNT> should always be 11 (binary) if the address is in IO space.)

In any case, the physical address is in S_CEFADR.

I-stream or D-stream read: The Bcache is not in ETM.

D-stream ownership read: The Bcache is in ETM. No write data has been merged with the returning fills.

[1] In this case the more general sense of "lost" is implied. That is, memory's ownership bit is set but no cache writes the data back when a read is done to that location. In some systems, it may be possible to identify which CPU memory "thinks" owns the data, but it is often not possible to determine which error caused this situation to arise.

The address should not be in IO space. If it is, it is an inconsistent status (see Section 15.8.1.22).

If the ownership read was for an Mbox write, the write was sent on the NDAL after the OREAD was aborted.

If the ownership read was for a read-lock, the corresponding write-unlock should have been received from the Ebox. The write-unlock is sent as a quadword WDISOWN by the Cbox, so no memory location is left owned. (If the error was on the requested quadword, a machine check would definitely have resulted. If a separate error causes prevent the write-unlock, that will be reported either in other error registers.)

Recovery procedures (all cases): Clear CEFSTS<LOCK, RDE>.

Additional Recovery procedures for D-stream ownership read (S_CEFSTS<WRITE> set): Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache).

Depending on the system environment, memory may have set its ownership bit for this block. If so the write data must have been lost, and a hard error interrupt is expected. Use the system dependent procedure for reseting the ownership bit in memory.

If memory would not have set its ownership bit for this block, memory's state may be correct and up to date.

Additional Recovery procedures for D-stream ownership read (S_CEFSTS<WRITE> not set): Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache).

Depending on the system environment, memory may have set its ownership bit for this block. The data in memory could still be good. The Bcache block is marked invalid in the Bcache tag store. However, if the error occurred on a read-lock, the corresponding write-unlock should have occurred and it will have cleared the ownership bit for this block.

If S_CEFSTS<COUNT> is greater than 0, then part of the data also is in the Bcache. In general, it is not possible to determine which quadwords are valid. However, if S_CEFSTS<REQ_FILL_DONE> is set, then the quadword in the Bcache block pointed to by S_CEFADR is valid (except in the case of a read-lock, but the data shouldn't be needed for memory repair in that case).

If S_CEFSTS<COUNT> is greater than 0, and the address in S_CEFADR is not in IO space, then the block was not owned before the operation began. In this case, use the procedures in Section 15.3.3.1.2.2 to determine if memory's ownership bit is set. If so, use the system specific procedure (see Section 15.3.3.1.2.2.2) to reset it. In some systems (the XMI2 for example) this may require a quadword of correct data be written to memory to reset the ownership bit. Section 15.3.3.1.2.3 describes procedures for extracting data from the Bcache data RAMs in this case.

If memory's ownership bit was left set as a result of this error and no non-destructive procedure exists for restoring it, then the hexaword block is lost.

Restart condition: Restart if the memory state repair procedure is successful or no repair is called for, no data is lost, and the address is not in IO space. If the hexaword block could not be repaired or data is lost, software must determine if the error is fatal to one process or the whole system and take appropriate action.

Post Restart Recovery: If the same fill error recurs on restart, then the block is probably "lost".[1] Software must determine if the error is fatal to one process or the whole system and take appropriate action. (If it is fatal only to one process, use the system dependent procedure for reseting memory's ownership bit.)

NOTE

It may be appropriate in this case to first cause each CPU in the system to flush its Bcache, and then restart once more.

NOTE

It may be that another error (such as an uncorrectable tag store error on a coherence request) will be repaired by the soft error interrupt handler before the restart actually occurs, fortuitously repairing the cause of the fill error.

15.8.1.15 Lost Bcache Fill Error

Description: Some number of fill errors occurred and were not latched because CEFSTS and CEFADR already contained a report of an unrecoverable error. Lost Bcache fill errors which do not cause hard error interrupts are always read errors.

Lost Bcache fill errors may be caused by more than one operand prefetch to the same cache block.

Lost Bcache fill errors may leave blocks marked owned by this CPU in memory without the Bcache actually owning the block.

The Bcache may be in ETM. Read S_CCTL<HW_ETM> to find out.

Recovery procedures: Clear CEFSTS<LOST_ERR>. If the Bcache is in ETM, flush the Bcache and clear CCTL<HW_ETM> (in that order).

Restart condition: Lost Bcache fill errors may leave blocks marked owned by this CPU in memory without the Bcache actually owning the block. In systems where the ownership bits are very reliably maintained (see Section 15.11.2, Note On Ownership Mechanism), restart.

In systems where the ownership bits are not very reliably maintained, crash the system.

15.8.1.16 Unacknowledged NDAL I-Stream or D-Stream Read or D-Stream Ownership Read

Description: An I-stream or D-stream read or D-stream ownership read was no-ACKed by the system environment. This could be because the external component(s) received bad NDAL parity or it could be due to a system-specific notification of "non-existent memory or IO location". The physical address is in S_CEFADR.

I-stream or D-stream read: The Bcache is not in ETM.

D-stream ownership read: The Bcache is in ETM.

The address should not be in IO space. If it is, it is an inconsistent status (see Section 15.8.1.22).

[1] In this case the more general sense of "lost" is implied. That is, memory's ownership bit is set but no cache writes the data back when a read is done to that location. In some systems, it may be possible to identify which CPU memory "thinks" owns the data, but it is often not possible to determine which error caused this situation to arise.

If the ownership read was for an Mbox write, the write was sent on the NDAL after the OREAD timed out. If the write was also no-ACKed, a hard error interrupt would have been posted. That is handled as a separate error.

Recovery procedures (all cases): Clear NESTS<NOACK>.

Additional Recovery procedure for D-stream ownership read: Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache). No error is expected during the Bcache flush.

15.8.1.17 Lost NDAL Output Error

Description: Some number of NDAL output errors occurred. Some number of read no-ACKs and/or BADWDATAs were missed. Hard error interrupt would have occurred if a write or writeback was no-ACKed.

Lost NDAL output errors may be caused by more than one operand prefetch to the same cache block.

The Bcache may be in ETM. read S_CCTL<HW_ETM> to find out.

Recovery procedure: Clear NESTS<LOST_OERR>. If CCTL<HW_ETM> is set, flush the Bcache and clear CCTL<HW_ETM> (in that order).

Restart conditions: Lost NDAL output errors may leave tagged bad locations in memory. In systems where the method of implementing tagged-bad data is reliable (see Section 15.11.1, Note On Tagged-Bad Data Mechanisms), restart.

If a tagged-bad block is not reliable in the particular system, crash the system.

15.8.1.18 PTE read errors

The following sections describe error handling for PTE read errors. PTE read errors are read errors which happen in reads issued by the Mbox in handling a TB miss. Handling of these errors is different from handling the same underlying error (Bcache data RAM error, Bcache fill error, or NDAL no-ACK error) when PTE read isn't the cause.

If S_PCSTS<PTE_ER> is set, then a PTE read issued by the Mbox in processing a TB miss had an unrecoverable error. The TB miss sequence was aborted because of the error. The original reference can be any I-stream or D-stream read or write.

PTE read errors are difficult to analyze, partly because the read error report in the Cbox does not directly indicate that the failing read was a PTE read. Because of this and because PTE read errors should be rare (a very small percentage of the reads issued by the Mbox are PTE reads), multiple errors which interfere with the analysis of the PTE error are not considered recoverable.

If the reference which incurs the PTE read error is a write, S_PCSTS<PTE_ER_WR> will be set. In this case the original write is lost. No retry is possible partly because the instruction which took the machine check may be subsequent to the one which issued the failing write. Also, PTE read errors on write transactions can cause a machine check at a practically arbitrary time in a microcode flow, and core machine state may not be consistent.

15.8.1.18.1 Bcache Data RAM Uncorrectable ECC Errors and Addressing Errors on PTE Reads

Description (addressing errors): A Bcache addressing error was detected by the Cbox in a PTE read during a Bcache hit. Addressing errors are the result of a mismatch between the address the Cbox drives to the RAMs for a read access and the address used to write that location. A multiple bit data error can appear to be addressing error, though it is extremely unlikely.

Description (uncorrectable ECC errors): A Bcache uncorrectable data error was detected by the Cbox in a PTE read during a Bcache hit. Uncorrectable data errors are the result of a multiple bit error in the data read from the Bcache. An addressing error with a single bit data error will appear as an uncorrectable data error.

Description (all cases): The Bcache in in ETM. S_BCEDIDX contains the cache index of the error, and S_BCEDECC contains the syndrome calculated by the ECC logic. The physical address of the PTE read can be found by reading the tag for the data block (using the procedure in Section 15.3.3.1.2.4). (If the physical address is found to be in IO space, it is an inconsistent status. See Section 15.8.1.22.)

If the block's tag is found to contain an ECC error, then the address can not be determined.

S_BCEDSTS<LOST_ERR> may be set. This lost error is probably due to the same PTE error occurring more than once. This is an acceptable assumption unless a hard error interrupt occurs after handling this error.

It should never be the case that both S_BCEDSTS<BAD_ADDR> and S_BCEDSTS<UNCORR> are set. If they are, it is an inconsistent status (Section 15.5.2.7).

Recovery procedures (addressing errors): To recover, clear BCEDSTS<LOCK, BAD_ADDR>.

Recovery procedures (uncorrectable ECC errors): To recover, clear BCEDSTS<LOCK, UNCORR>.

Recovery procedures (both cases): Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache). Clear PCSTS<PTE_ER>. If the data is owned by the Bcache and if the error repeats itself (is not transient), then a writeback error will result from the flush procedure. Software should prepare for this by clearing NESTS and BCEDSTS errors.

Restart condition: If no writeback error occurs in the Bcache flush, restart if:

(S_PCSTS<PTE_ER_WR> = 0).

If

(S_PCSTS<PTE_ER_WR> = 1),

crash the system.

If a writeback error occurs in the Bcache flush, then the data is presumed to be unrecoverable. See Section 15.8.1.10 for a description of handling an error in a writeback (software must determine if the error is fatal to one process or the whole system and take appropriate action).

15.8.1.18.2 NDAL PTE Read Timeout Errors

Description: A PTE read timed out before any fill quadword was received. This is not an accepted means for a system environment to notify the NVAX CPU of "non-existent memory or IO location". The error could be caused by an error in the system environment or an NDAL parity error on the returned data. It also could be caused by some previous error in the system environment or this CPU which leaves a cache block marked as owned in memory and not marked as owned in any cache in the system.

S_CEFSTS<COUNT> indicates the number of quadwords received before the error. (S_CEFSTS<COUNT> should always be 11 (binary) if the address is in IO space.) The physical address is in S_CEFADR.

CEFSTS<WRITE> should not be set. If it is, it is an inconsistent status (see Section 15.5.2.7).

The physical address of the PTE is in S_CEFADR. The Bcache is not in ETM. The read could not have been an ownership read, so this error can not have caused the ownership bits in memory to be left in the wrong state.

S_CEFSTS<LOST_ERR> may be set. This error is probably due to the same PTE error occurring more than once. This is an acceptable assumption unless a hard error interrupt occurs after handling this error.

Recovery procedures: Clear CEFSTS<LOCK, TIMEOUT>. Clear PCSTS<PTE_ER>.

Restart condition: Restart if:

(S_PCSTS<PTE_ER_WR> = 0).

Otherwise, crash the system.

Post Restart Recovery: If the same fill error recurs on restart, then the block is probably "lost".[1] Software must determine if the error is fatal to one process or the whole system and take appropriate action. (If it is fatal only to one process, use the system dependent procedure for reseting memory's ownership bit.)

NOTE

> It may be appropriate in this case to first cause each CPU in the system to flush its Bcache, and then restart once more.

NOTE

> It may be that another error (such as an uncorrectable tag store error on a coherence request) will be repaired by the soft error interrupt handler before the restart actually occurs, fortuitously repairing the cause of the fill error.

[1] In this case the more general sense of "lost" is implied. That is, memory's ownership bit is set but no cache writes the data back when a read is done to that location. In some systems, it may be possible to identify which CPU memory "thinks" owns the data, but it is often not possible to determine which error caused this situation to arise.

15.8.1.18.3 NDAL PTE Read Data Errors

Description: A PTE read ended with an RDE (read data error) NDAL cycle before any the fill quadwords were received. This is an accepted means for a system environment to notify the NVAX CPU of "non-existent memory or IO location". Otherwise, the error could be caused by an error in the system environment. It also could be caused by some previous error in the system environment or this CPU which leaves a cache block marked as owned in memory and not marked as owned in any cache in the system.

S_CEFSTS<COUNT> indicates the number of quadwords received before the error. (S_CEFSTS<COUNT> should always be 11 (binary) if the address is in IO space.) The physical address is in S_CEFADR.

CEFSTS<WRITE> should not be set. If it is, it is an inconsistent status (see Section 15.5.2.7).

The physical address of the PTE is in S_CEFADR. The Bcache is not in ETM. The read could not have been an ownership read, so this error can not have caused the ownership bits in memory to be left in the wrong state.

S_CEFSTS<LOST_ERR> may be set. This error is probably due to the same PTE error occurring more than once. This is an acceptable assumption unless a hard error interrupt occurs after handling this error.

Recovery procedures: Clear CEFSTS<LOCK, RDE>. Clear PCSTS<PTE_ER>.

Restart condition: Restart if:

(S_PCSTS<PTE_ER_WR> = 0).

Otherwise, crash the system.

Post Restart Recovery: If the same fill error recurs on restart, then the block is probably "lost".[1] Software must determine if the error is fatal to one process or the whole system and take appropriate action. (If it is fatal only to one process, use the system dependent procedure for reseting memory's ownership bit.)

NOTE

It may be appropriate in this case to first cause each CPU in the system to flush its Bcache, and then restart once more.

NOTE

It may be that another error (such as an uncorrectable tag store error on a coherence request) will be repaired by the soft error interrupt handler before the restart actually occurs, fortuitously repairing the cause of the fill error.

[1] In this case the more general sense of "lost" is implied. That is, memory's ownership bit is set but no cache writes the data back when a read is done to that location. In some systems, it may be possible to identify which CPU memory "thinks" owns the data, but it is often not possible to determine which error caused this situation to arise.

15.8.1.18.4 Unacknowledged NDAL PTE Read

Description: A PTE read was no-ACKed by the system environment. This could be because the external component(s) received bad NDAL parity or it could be due to a system-specific notification of "non-existent memory or IO location".

The physical address of the PTE is in S_NEOADR. The Bcache is not in ETM.

S_CEFSTS<LOST_OERR> may be set. This error is probably due to the same PTE error occurring more than once. This is an acceptable assumption unless a hard error interrupt occurs after handling this error.

Recovery procedures: Clear NESTS<NOACK>. Clear PCSTS<PTE_ER>.

Restart condition: Restart if:

(PCSTS<PTE_ER_WR> = 0).

Otherwise, crash the system.

15.8.1.18.5 Multiple Errors Which Interfere with Analysis of PTE Read Error

Because PTE read errors lead to several unusual cases, restart is not recommended in the event that other errors cloud the analysis of the PTE read error.

Pending Interrupts: A hard or soft error interrupt should be pending, or possibly both.

Recovery procedures: No specific recovery action is called for.

Restart condition: No restart is possible. Crash the system.

15.8.1.19 NDAL Parity Errors

Description: A cycle with a paritv error was received by the NVAX CPU chip from the NDAL. If it is an inconsistent parity error, another node acknowledged the transaction despite the parity error seen by the NVAX chip. The Bcache is in ETM. The Bcache is coherent with memory because it only accesses VALID-OWNED locations in the Bcache data RAMs once in ETM. Some other node's request may timeout because the Cbox missed a coherency request for writeback. The Pcache may now be incoherent since an NDAL write to a Bcache VALID-UNOWNED location may have been missed.

In some systems (e.g., OMEGA), a no-ACK on an NDAL command implies no effect from that command took place. This makes NDAL parity errors very recoverable. In other systems (e.g., XMI2), a no-ACK on an NDAL command does not imply this (for invalidates forwarded from the XMI2 bus), and all parity errors imply possible lost invalidates and incoherent Pcache.

Recovery procedure: Clear NESTS<PERR> and NESTS<INCON_PERR>. Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache).

Restart condition: If no-ACK in the specific system implies a command was not effective, and if the error was not an inconsistent parity error, restart. Otherwise, It isn't possible to determine whether the interrupted instruction stream may have seem the effect of out of order writes because of the Pcache missing an invalidate. Crash the system.

15.8.1.20 Lost Parity Errors

Description: Some number of cycles with parity errors were received by the NVAX CPU chip from the NDAL. Some may have been inconsistent parity errors. The Bcache is in ETM. The Bcache is coherent with memory because it only accesses VALID-OWNED locations in the Bcache data RAMs once in ETM. Some other node may timeout because the Cbox missed a coherency request for writeback. The Pcache may now be incoherent since an NDAL write to a Bcache VALID-UNOWNED location may have been missed.

Recovery procedure: Clear NESTS<LOST_PERR>. Flush the Bcache. Clear CCTL<HW_ETM> (after flushing the Bcache).

Restart condition: It isn't possible to determine whether the interrupted instruction stream may have seem the effect of out of order writes because of the Pcache missing an invalidate. Crash the system.

15.8.1.21 System Environment Soft Error Interrupts

Description: Errors which occur in the system environment and do not result in loss of data or which can notify the NVAX CPU by returning RDE also notify the CPU of the error by asserting S_ERR_L (e.g., read errors). Errors which are corrected automatically by hardware and do not result in loss of data should use soft error interrupt notification.

NOTE

> It is important that components in the system environment which assert S_ERR_L have a CPU accessible register which reports the S_ERR_L assertion.
>
> Attention should be given to the robustness tagged-bad data schemes. If error detection for these schemes is good enough, then error recovery may be able to ignore lost soft errors. Lost soft errors are very possible in NVAX systems because the first error doesn't normally prevent NVAX from continuing to issue new requests (sue to macropipelining).
>
> Similarly, good error detection schemes on the ownership bits in memory may facilitate recovery from lost soft errors.
>
> It is also recommended that an address be stored where applicable. They allow software to do improve the systems chance of surviving an error event without crashing by cleaning up tagged-bad locations and the like. For example, a write timeout clearing a page in the VMS page handler may be unrecoverable, while clearing that tagged-bad data location before it ever got to the page handler might be quite recoverable.

Recovery procedures: Clear the error status bits in the system registers and perform any necessary system dependent recovery procedure.

Restart condition: Typically, restart is possible, though in cases where data is lost software may have to kill one process or crash the system.

15.8.1.22 Inconsistent Status in Soft Error Interrupt Analysis

Description: A presumed impossible error report was found in the error registers. This could be due to a hardware failure or bug.

Recovery procedures: No specific recovery action is called for.

Restart condition: No restart is possible. The integrity of the entire system is questionable. Crash the system.

NOTE

This status can result if machine check occurs. Software may employ some mechanism for determining that this occurred, but it must be sure that mechanism can't ever falsely indicate that an inconsistent status is acceptable. Inconsistent status is a serious problem and should not be ignored.

15.9 Kernel Stack Not Valid Exception

A Kernel Stack Not Valid Exception occurs when a memory management exception is detected while attempting to push information on the kernel stack during microcode processing of another exception. Note that a console halt with an error code of ERR_INTSTK is taken if a memory management exception is encountered while attempting to push information on the interrupt stack.

The Kernel Stack Not Valid exception is dispatched through SCB vector 08 (hex) with the stack frame shown in Figure 15–11.

Figure 15–11: Kernel Stack Not Valid Stack Frame

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              PC                                               | :(SP)
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              PSL                                              |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

15.10 Error Recovery Coding Examples

To be supplied.

15.11 Miscellaneous Background Information

This section contains miscellaneous background information relevant to this error handling chapter.

15.11.1 Note On Tagged-Bad Data Mechanisms

Writebacks which are sent as BADWDATA are supposed to appear as tagged-bad data in memory, and further reads to that block should fail. In some systems, tagged bad data is implemented by a mechanism as reliable as that used to store data. In at least one system (OMEGA), tagged-bad data is implemented by altering the ECC code of the data as it is written. Some single-bit and many double-bit errors in this data can make it appear to be correctable or correct when read. This is less protection from error than valid data has. In such a system, an error which results in a lost tagged-bad-data block is reason to crash the system. In systems with reliable storage of "tagged-bad-data", operation can continue after such an error because it is essentially certain that any process which accesses that data will see an RDE error for that block and will machine check before it uses the bad data.

The Bcache data RAMs in NVAX use the above relatively unreliable mechanism for tagged-bad data. Three ECC check bits are flipped in the stored value. This mechanism would often prevent a subsequent read from succeeding, but it is not sufficiently reliable to allow missing tagged-bad blocks in the Bcache to be tolerated. As a result, all errors which may have left a tagged-bad block in the Bcache without some error address register pointing it out are cause to crash the system.

15.11.2 Note On Ownership Mechanism

In the absence of additional errors, the memory/cache ownership mechanism ensures that no other process can access the block whose ownership bit is set in memory and is not owned by any cache. Cache coherence in the system depends on this mechanism. In some systems, memory error detection and correction for ownership bits is as reliable as for data. This is true of XMI2 based systems. However, in some systems the mechanism is less reliable. One example is the OMEGA system, where the ownership bits are stored with a single-bit-error-detect-and-correct scheme which can not detect most double bit errors and therefore interprets most double bit errors as correctable single bit errors. In such a system, error situations in which unknown blocks in memory may be owned should be taken as a system crash.

In OMEGA, there is a proposal make up for the non-robust ownership bit error detection scheme by flushing the cache on every "correctable" ownership bit error in the NMC. If the "correctable" error really is an uncorrectable error, this may be detected by a WDISOWN to an unowned memory location. This is because some uncorrectable errors are seen as correctable errors, so one ownership bit is flipped by memory's error correction hardware and at least two bits were wrong to start with. There is a chance that the "correction" flips one of the bad bits, but it could also flip one of the remaining correct bits. This leaves the memory with one or three incorrect ownership bits after an uncorrectable "correctable" error. If every cache is flushed immediately after a "correctable" error, then writebacks to apparently unowned locations may result if the error is inadvertently made worse by the correction scheme. These are detectable protocol errors and should lead to a system crash. If the effect of the error correction was to mark block(s) as owned when no cache owns them, then eventually some process will attempt to access that data and time out. If the error was successfully corrected, then flushing the caches causes a pause in processing and no bad effects. If these errors are infrequent, this seems an acceptable loss in performance in exchange for increased reliability.

15.12 Revision History

Table 15–6: Revision History

Who	When	Description of change
Mike Uhler	06-Mar-1989	Release for external review.
Mike Uhler	19-Dec-1989	Update for second-pass release.
John Edmondson	12-Feb-1990	Update with error handling information.
John Edmondson	30-Jun-1990	Update further after internal review and resolution of many issues.
John Edmondson	31-May-1991	Minor updates for pass 2 changes.

Chapter 16

Chip Initialization

16.1 Overview

This chapter describes the hardware initialization process for the NVAX CPU chip. The hardware and microcode start the initialization, and then pass control to the console macrocode at address E0040000 for further initialization.

Much of the job of initialization involves setting the NVAX internal processor registers (IPRs) to a known state, or using NVAX IPRs to perform functions such as cache initialization. See Chapter 2 for a list of the NVAX IPRs. Also, see the individual box chapters for a more in depth definition of many of the IPRs.

16.2 Hardware/Microcode initialization

The NVAX Chip hardware initializes to the following state on powerup or the assertion of chip reset:

1. The VIC, Pcache, and Bcache are disabled.
2. The RLOG is cleared.
3. The Fbox and vector unit are disabled.
4. The microstack is cleared.
5. The Mbox and Cbox are reset, and all previous operations are flushed.
6. The Fbox is reset.
7. The Ibox is stopped, waiting for a LOAD PC.
8. All instruction and operand queues are flushed.
9. All MD valid bits are cleared, and all Wn valid bits are set.
10. A powerup microtrap is initiated which starts the Ebox at the label IE.POWERUP..

The NVAX Chip microcode then does the following:

1. Hardware interrupt requests are cleared.
2. ICCS<6> is set to 0.
3. SISR<15:1> is set to 0.
4. ASTLVL is set to 4.

5. The Mbox PAMODE IPR is set to 30-bit physical address mode.
6. CPUID is set to 0.
7. The BPCR branch history algorithm is reset to the default value.
8. Backup PC is retrieved from the Ibox and saved in SAVPC.
9. PME is cleared.
10. The current PSL, halt code, and value of MAPEN are saved in SAVPSL.
11. MAPEN is cleared (memory management is disabled).
12. All state flags are cleared.
13. PSL is loaded with 041F0000.
14. PC is loaded with E0040000 (the address of the start of the console code).

16.3 Console initialization

The console macrocode has the job of filling the gap between the initialized state described above and the initial state needed for the operating system. To that end, the console code does the following:

1. Set CPUID to the correct value from the system environment.
2. Set ECR (Ebox Control Register) as follows:
    1. Set FBOX_ENABLE to enable the Fbox.
    2. Set S3_TIMEOUT_EXT as required by the system environment.
    3. Set FBOX_ST4_BYPASS_ENABLE to enable Fbox stage 4 bypass.
    4. Write one to S3_STALL_TIMEOUT to clear any error.
    5. Set ICCS_EXT as required by the system environment.
3. Set ICSR (Ibox Control Status Register) as follows:
    1. Clear ENABLE to leave the VIC disabled.
    2. Write one to LOCK to clear any error.
4. Set the PAMODE register MODE bit as required by the system.
5. Write one to clear the LOCK bit in TBSTS (Translation Buffer Status).
6. Initialize the PCSTS (Pcache Status) Register:
    1. Write one to clear the LOCK bit.
    2. Write one to clear PTE_ER_WR.
    3. Write one to clear PTE_ER.
7. Set CCTL (Cbox Control) as follows:
    1. Clear ENABLE to leave the Bcache disabled.
    2. Set TAG_SPEED, DATA_SPEED, and SIZE to reflect the Bcache RAM configuration in the system.
    3. Clear FORCE_HIT.
    4. Clear DISABLE_ERRORS.
    5. Clear SW_ECC.
    6. Clear TIMEOUT_TEST.
    7. Clear DISABLE_PACK to allow the write packing feature.

    8. Clear SW_ETM.
    9. Write one to clear HW_ETM.
8. Clear the various Cbox error registers:
    1. BCETSTS (Bcache Error Tag Status): Write one to LOCK, CORR, UNCORR, BAD_ADDR, and LOST_ERR to clear any errors.
    2. BCEDSTS (Bcache Error Data Status): Write one to LOCK, CORR, UNCORR, BAD_ADDR, and LOST_ERR to clear any errors.
    3. CEFSTS (Cbox Error Fill Status): Write one to RDLK, LOCK, TIMEOUT, RDE, and LOST_ERR to clear any errors.
    4. NESTS (NDAL Error Status): Write one to NOACK, BADWDATA, LOST_OERR, PERR, INCON_PERR, and LOST_PERR to clear any errors.

16.4 Cache initialization

Either the console code or the operating system will do the following final initialization steps (code examples are given):

1. Initialize the VIC

```
;       This code initializes the VIC by writing all 128 tags with
;       good parity and all valid bits clear.
;

        movl    #^x00000020, r0         ; tag index increment = 1 hexaword block
        movl    #0, r1                  ; tag init value
        movl    #0, r2                  ; VIC tag starting address
        movl    #^x00000800, r3         ; VIC tag ending address + 1 block
        movl    #PR19$_VMAR, r4         ; VIC memory address register (VMAR)
        movl    #PR19$_VTAG, r5         ; VIC tag register (VTAG)

vic_loop:
        mtpr    r2, r4                  ; write current index to VMAR
        mtpr    r1, r5                  ; write the tag via VTAG
        addl2   r0, r2                  ; increment index by the block size
        cmpl    r3, r2                  ; check if done
        bneq    vic_loop                ;
```

2. Enable the VIC

```
        mtpr    #<icsr$m_enable+icsr$m_lock>, #PR19$_ICSR
```

3. Initialize the Bcache tags

```
;       This code initializes the Bcache by writing all tags with good
;       ECC and all valid and owned bits clear.  This example initializes
;       a 512Kb Bcache.  This code can be changed to init the other legal
;       Bcache sizes by changing the value in R3. SW_ECC in CCTL is clear,
;       so the CBOX will generate correct ECC for the tag/valid/owned bits.

        movl    #^x00000020, r0         ; tag index increment = 1 hexaword block
        movl    #0, r1                  ; tag init value
        movl    #^x01000000, r2         ; Bcache tag starting address
        movl    #^x01080000, r3         ; Bcache tag ending address + 1 block
                                        ; for 512Kb Bcache

bcache_loop:
        mtpr    r1, r2                  ; write tag to current tag address
        addl2   r0, r2                  ; increment index by the block size
        cmpl    r3, r2                  ; check if done
        bneq    bcache_loop             ;
```

4. Initialize the Bcache data

```
 .SBTTL ZERO_BCACHE_DATA
;++
; ZERO_BCACHE_DATA - Write zero data and good ECC to the BCACHE data rams
;--

BYTES_PER_QUADWORD = 8

BYTES_PER_PAGE = 512

QUADWORDS_PER_PAGE = BYTES_PER_PAGE/BYTES_PER_QUADWORD

ZERO_BCACHE_DATA:
 PUSHR #^M<R0,R1,R2,R3,R4,R5,R6> ; Save registers
 MFPR #PR$_CPUID,R5  ; XMI node id
 MOVL SYSL$L_BACKUP_CACHE_CONSTANT[R5],R1 ; Formative cache constant
 MTPR R1,#PR13$_CCTL  ; Set cache with default constant
 EXTZV #PR13_CCTL$V_SIZE,#PR13_CCTL$S_SIZE,R1,R2 ; Extract backup cache size
 MOVL SYSL$L_BCACHE_PAGE_CONSTANT[R2],R5 ; Cache page count
 CLRL R6    ; "AOB" index
 CLRQ R1    ; Quadword data to be written to BCACHE rams
10$:
 MULL3 #BYTES_PER_PAGE,R6,R3 ; BCACHE page index to write
 BSBW MAP_PHYSICAL_ADDRESS ; Map R3 PA to R4 VA
 CLRL R3    ; "AOB" index
20$:
 JSB @IO_WRITE_BCACHE_DATA ; Write BCACHE data
 ADDL2 #BYTES_PER_QUADWORD,R4 ; Update VA
 AOBLSS #QUADWORDS_PER_PAGE,R3,20$ ; Loop 'til done
 AOBLSS R5,R6,10$  ; Loop 'til done
 MFPR #PR$_CPUID,R5  ; XMI node id
 MOVL SYSL$L_BACKUP_CACHE_CONSTANT[R5],R1 ; Formative cache constant
 MTPR R1,#PR13$_CCTL  ; Set cache with default constant
 POPR #^M<R0,R1,R2,R3,R4,R5,R6> ; Restore registers
 RSB    ; Return


 .SBTTL MAP_PHYSICAL_ADDRESS

;++
; MAP_PHYSICAL_ADDRESS - Map a physical address with a system VA
;
; INPUTS:
; R3  =  Physical address to map to system VA
;
; OUTPUTS:
; R4  =  System VA of physical address in R3
;--

MAP_PHYSICAL_ADDRESS:
 PUSHR #^M<R0,R1,R9>  ; Save registers
 BSBW GET_XNP_NUMBER  ; CPU number to R9
 MOVAL @SYSLOA_SPTE[R9],R0 ; Address of this CPU's SPTE
 BICL2 #PTE$M_VALID,(R0) ; Invalidate SPTE

 INVALIDATE_TB ENVIRON=UNMAPPED ; Invalidate this TB

 MOVL (R0),R1   ; SPTE
 EXTZV #VA$V_VPG,#PTE$S_PFN,R3,R4 ; Address PFN
 INSV R4,#PTE$V_PFN,#PTE$S_PFN,R1 ; Insert the PFN
 BISL3 #<PTE$M_VALID!PTE$M_MODIFY!PTE$C_KW>,R1,(R0) ; Map PFN
 EXTZV #VA$V_BYTE,#VA$S_BYTE,R3,R0 ; Address byte offset
 MULL3 #512,R9,R1  ; This CPU's page offset
 MOVAB @SYSLOA_SPTE_VA[R1],R4 ; VA that this CPU's SPTE maps
 INSV R0,#VA$V_BYTE,#VA$S_BYTE,R4 ; A VA that maps physical address
 POPR #^M<R0,R1,R9>  ; Restore registers
 RSB    ; Return
```

```
;++
; INPUTS:
; R1 = Lo longword data to be written to BCACHE
; R2 = Hi longword data to be written to BCACHE
; R4 = Virtual address that maps physical address corresponding
;      to secondary cache index to be written.
;
; OUTPUTS:
; R0 LBS indicates BCACHE data written, otherwise clear
;--

;********************************************************************************
;
; This routine cannot be stepped through using XDELTA.  The FORCEHIT bit
; in the backup cache control is set and will cause erroneous hits to
; occur in the secondary cache.
;
;********************************************************************************
 .ALIGN LONG

IO_WRITE_BCACHE_DATA_ROUTINE:
 MOVL R3,IO_SAVED_REGISTER ; Save register
 MTPR #0,#PR$_TBIA  ; Reset TB allocation pointer
 CLRL R0   ; Signal failure
 TSTL 10$   ; Ensure TB hit
 TSTL 30$   ; Ensure TB hit
 TSTL (R4)   ; Ensure TB hit
 TSTL B^4(R4)   ; Ensure TB hit
 MOVAB 10$,R3   ; Address to check
 MTPR R3,#PR$_TBCHK  ; In TB
 BVC 20$   ; If VC no
 MOVAB 30$,R3   ; Address to check
 MTPR R3,#PR$_TBCHK  ; In TB
 BVC 20$   ; If VC no
 MOVAL (R4),R3   ; Address to check
 MTPR R3,#PR$_TBCHK  ; In TB
 BVC 20$   ; If VC no
 MOVAL B^4(R4),R3  ; Address to check
 MTPR R3,#PR$_TBCHK  ; In TB
 BVC 20$   ; If VC no
10$:
 MFPR #PR13$_CCTL,R3  ; Read CCTL
 BICL2 #<-   ; Form a mask
  <1@PR13_CCTL$V_FORCE_HIT>!- ; Force hit mode
  <1@PR13_CCTL$V_DISABLE_ERRORS>!- ; Disable errors
  <0>-   ;
  >,R3   ; Local copy control register
 BISL3 #<-   ; Form a mask
  <1@PR13_CCTL$V_ENABLE>!- ; Enable BCACHE
  <1@PR13_CCTL$V_FORCE_HIT>!- ; Force hit mode
  <1@PR13_CCTL$V_DISABLE_ERRORS>!- ; Disable errors
  <0>-   ;
  >,R3,R0   ; Local copy control register
 MTPR R0,#PR13$_CCTL  ; Enable Bcache - FORCE HIT, DISABLE ERRORS
 MFPR #PR13$_CCTL,R0  ; Allow the dust to settle...
 MOVQ R1,(R4)   ; Write BCACHE data
 MTPR R3,#PR13$_CCTL  ; BCACHE off
 MFPR #PR13$_CCTL,R3  ; Allow the dust to settle...
 MOVL #SS$_NORMAL,R0  ; Signal success
20$:
 MOVL IO_SAVED_REGISTER,R3 ; Restore register
 RSB   ; Return
30$:
```

5. Initialize the Pcache

```
;       This code initializes the Pcache by writing all 256 tags with
;       good parity and all valid bits clear.

        movl    #^x00000020, r0         ; tag index increment = 1 hexaword block
        movl    #0, r1                  ; tag init value
        movl    #^x01800000, r2         ; Pcache tag starting address
        movl    #^x01802000, r3         ; Pcache tag ending address + 1 block
```

```
pcache_loop:
        mtpr    r1, r2                  ; write tag to current tag address
        addl2   r0, r2                  ; increment index by the block size
        cmpl    r3, r2                  ; check if done
        bneq    pcache_loop             ;
```

6. Enable the Bcache and the Pcache

   NVAX cache coherency requires that the Pcache is always a subset of the Bcache. This code to enable the caches is arranged to insure that this is true. Thus, the Bcache is enabled first, and an REI is executed between the Bcache enable and the Pcache enable. The purpose of the REI is to synchronize data prefetching such that the Pcache will not perform any fills to addresses that were not also filled in the Bcache.

```
        mfpr    #PR19$_CCTL, r6         ; get current value in Cbox CTL IPR
        bisl2   #<cctl$m_enable>, r6    ; set the Bcache enable bit
        mtpr    r6, #PR19$_CCTL         ; write the new Cbox CTL IPR

        movpsl  -(sp)                   ; push the psl
        moval   init_cont,-(sp)         ; and the next PC
        rei                             ; branch to the next PC
                                        ; flushing the VIC
                                        ; and aborting all
                                        ; previous IREADS

                                        ; Now that state is synchronized, enable
                                        ; the Pcache
init_cont:
        mtpr    #<pcctl$m_d_enable+pcctl$m_i_enable+pcctl$m_p_enable>, #PR19$_PCCTL
```

16.5 Miscellaneous Information

There is no need to explictly initialize the Translation Buffer as the NVAX microcode performs an internal TBIA on any MTPR to the MAPEN IPR.

There is no need to explictly initialize the data portions of the VIC or Pcache as long as the tags are initialized with all valid bits clear. Both Bcache tags and Bcache data must be initialized before the cache is enabled.

16.6 Revision History

Table 16–1: Revision History

Who	When	Description of change
Debra Bernstein	9-May-1990	Initial edit
Debra Bernstein	19-Nov-1990	Add Miscellaneous Information section. Add true code examples for cache init. Add information on the ordering of cache enable.
Debra Bernstein	11-Mar-1991	Update to pcsts, tbsts
Rebecca Stamm	9-Oct-1991	Bcache data must be initialized as well as the Bcache tags.

Chapter 17

Chip Clocking

17.1 Overview of the NVAX Clocking System

The NVAX CPU generates all the clock signals required to operate the CPU and the NDAL interface. The clocks are derived from a high frequency oscillator signal that is supplied to the chip. To allow for flexible logic design the chip implements a four phase clocking system. The four internal NVAX clock phases are generated on-chip by dividing the frequency of the external oscillator by four.

The NVAX chip generates and drives the NDAL clocks which are used to clock the peripheral chips on the NDAL bus. The NDAL also uses a four phase clock scheme, but runs three times slower than the internal NVAX clocks.

17.2 Receiving the NVAX External Oscillator Signal

The NVAX chip can receive the external clock from one of two sources depending on the state of the **OSC_TEST_H** pin. When **OSC_TEST_H** is asserted the clock is received by the **OSC_TC1_H** and **OSC_TC2_H** pins. These pins are configured to use standard 3V CMOS signal levels. When **OSC_TEST_H** is deasserted the clock is received by the **OSC_H** and **OSC_L** pins. These pins use a differential amplifier circuit to receive the clock signal from an ECL oscillator. Figure 17–1 shows the NVAX clock interface circuitry.

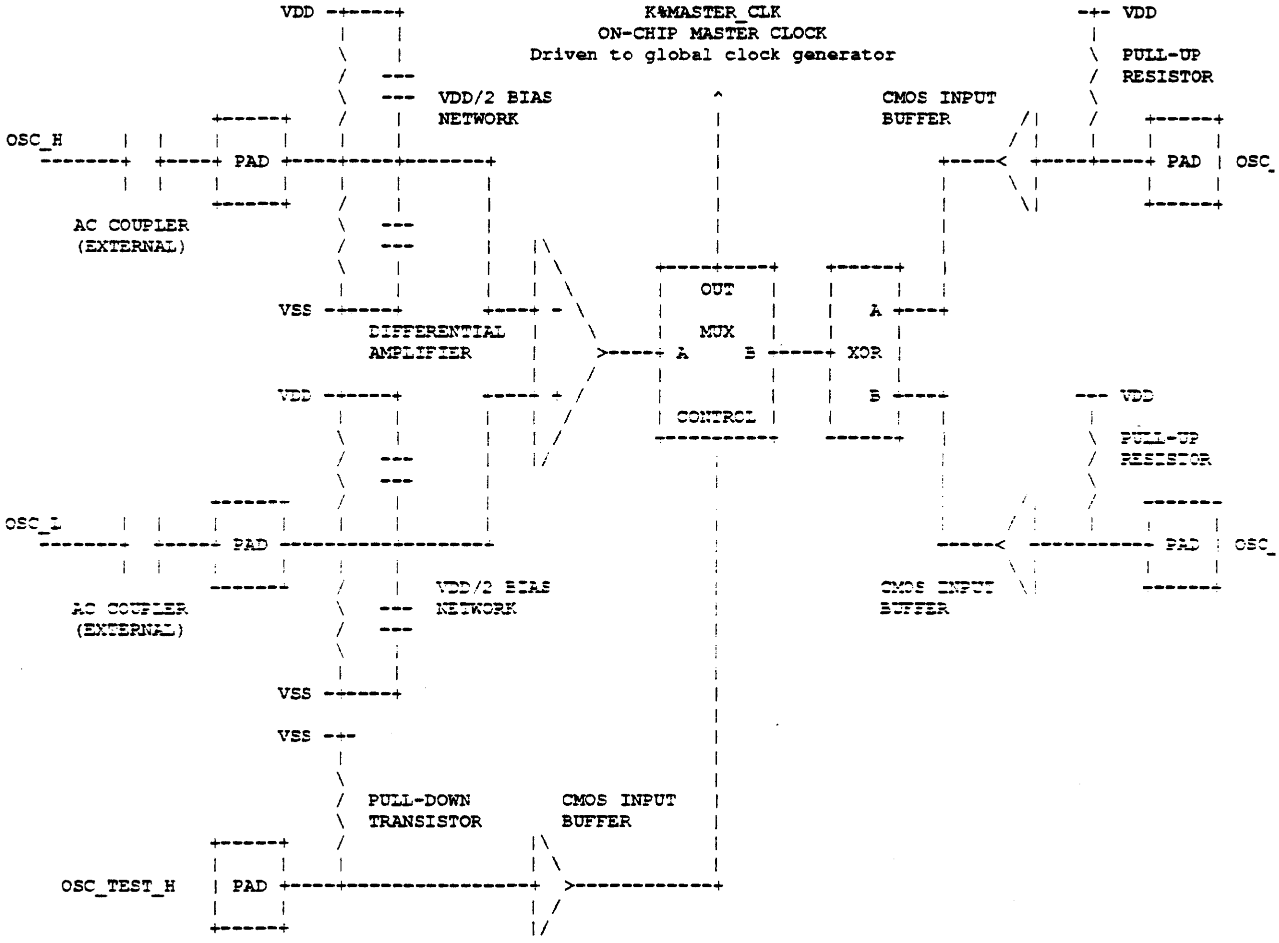
EXTERNAL OSCILLATOR

Detailed information concerning the design of the external oscillator can be found in the NVAX Signal Integrity Specification.

17.2.1 The System Environment

During normal system operation **OSC_TEST_H** is tied low and the **OSC_H** and **OSC_L** pins are used to receive the external clock source. The NVAX CPU is designed to operate at a maximum internal clock speed of 100 MHz. This requires the external oscillator to deliver a 400 MHz clock. At these frequencies the generation and interconnection of signals is extremely complex and specialized circuitry must be used.

Figure 17–1: NVAX CPU Interface Circuitry



The NVAX oscillator generates a pair of clock signals that are 180 degrees out of phase. The oscillator does not supply standard CMOS logic levels. The signals have a peak to peak voltage swing of .5 volts centered at 3.5 volts - therefore, a standard CMOS input buffer cannot be used on the chip to receive the signals. Instead, a differential amplifier is used and the signals are AC coupled and level shifted before they are received by the amplifier.

17.2.2 The Chip Test Environment

The chip tester, used during chip manufacturing to functionally verify the part, cannot supply a 400 MHz clock. The two pins **OSC_TC1_H** and **OSC_TC2_H** offer an alternative method for supplying the chip with clocks. The pin **OSC_TEST_H** is used to select between the system and test clocking modes. When the pin is asserted the test clock pins supply the clock to the chip.

The test clock pins are supplied with two clock signals that are 90 degrees out of phase. They are XORed on-chip to generate the internal 2X clock signal K_PAD_CK1%ZZ. Figure 17–2 shows the relationship between K_PAD_CK1%ZZ and the test clock input signals.

Figure 17–2: On-Chip XOR Test Functionality Waveforms

```
              ------------------------------------------------------------------------------------------------
              +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |
OSC_TC1_H     /~~~~~~~\_______/~~~~~~~\_______/~~~~~~~\_______/~~~~~~~\_______/~~~~~~~\_______/~~~~~~~\_______
              +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |
OSC_TC2_H     ____/~~~~~~~\_______/~~~~~~~\_______/~~~~~~~\_______/~~~~~~~\_______/~~~~~~~\_______/~~~~~~~\___
              +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |
K_PAD_CK1%ZZ  /~~~\___/~~~\___/~~~\___/~~~\___/~~~\___/~~~\___/~~~\___/~~~\___/~~~\___/~~~\___/~~~\___/~~~\___
              +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |
              +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |
              ------------------------------------------------------------------------------------------------
```

In addition to the frequency doubling feature of the test clock input circuitry, the pins use CMOS differential amplifiers to receive the clock signals. Hence, the test oscillator clock inputs can be used to drive the chip at slower than maximum speeds using standard 3 volt CMOS logic levels.

17.3 On-Chip Clocks

17.3.1 Clock Generation/Distribution Overview

Figure 17–3 illustrates the overall structure of the clock generation/distribution system. The clocks are distributed across the chip in two stages. The global clock generator receives the master clock signal and generates the following global clocks that are driven to various sections of the nvax chip:

- eight single phase matched clocks (true and complement) **K%PHI_1:4_H & L**
- four double phase matched clocks **K%PHI_12:41_H**
- four NDAL matched clocks **K_GLB%PHI12:41_OUT_H**
- two specially tuned single phase clocks **K_P%PHI_3E** and **K_V%PHI_3E**

For purposes of defining clock specifications in different parts of the NVAX chip, *clock section* (or simply, *section*) is defined for the remainder of the chapter to be one of the chip sections shown in Table 17–1.

Table 17–1: NVAX CPU Clock Sections

Section Name	Symbol
Cbox	MCB
Ebox	IEB
Fbox	F
Ibox	I
Mbox	M
Pcache	P

Table 17–1 (Cont.): NVAX CPU Clock Sections

Section Name	Symbol
VIC	V
Upper I/O Pad Logic	PAD
Lower I/O Pad Logic	PADL
Global Reset Logic	K

Figure 17–3: On-Chip Clock Distribution

```
                                                        To
                                                    all sections
                                                             /
                                                        ~                  ----------------------
                                                        |                  |   Section X1       |
                                                        |                  | Local Clock Drive  |
                                                        ------------------>+                     -----> K_X%PHI_
                                        ----------      |    -----------   | Outputs 4 single   |
                                        | Fbox   |<-------    | Sect X1 |   | phase clks         |
                                        | Dummy  |      ---->+ Dummy   |   ---------------------+
K_F%PHI_1:4_H       ------------------  | Loads  |<--   |    | Loads   |
K_F%PHI_1:4_L       |     Fbox       |  ---------- |    |    -----------
K_F%PHI_12:41_H <--- Local Clk Drv   |<------------------+
                    |   8 sgl phase  |             |    |
                    |   4 dbl phase  |<------------+    |
                    +----------------+             |    |
                                                   |    |
                                                   |    |
                        DISTRIBUTION TO FBOX ONLY >|    |< GLOBAL DISTRIBUTION - K%PHI_1:4_H
                          K%PHI_1:4_L              |    |  Matched IC and Load
                          K%PHI_12:41_H            |    |
                        Matched IC & Load   +------+----+----------+
                                            | Global Clock Gen     +------> K_V%PHI_3E_H \
                                            |                      +------> K_P%PHI_3E_H / Special Phase 3 Cloc
                               400 MHZ      | Divide by 12         |
                               master  ---->+                      +------> NDAL clocks to I/O Pad Drivers
                               clock        | Outputs:             |          K_GLB%PHI12:41_OUT_H
                                            | 8 single phase clks  |        Matched IC and Load
                                            | 4 double phase clks  |
                                            | 4 NDAL clks          |
                                            +----------------------+
```

The global clock signals are received and driven into each section by local clock buffers. It is these local clocks that are used to control logic sequencing throughout the chip. Note that the active high single phases are used by all sections, while the double phases and active low single phases are used only in the Fbox. NDAL clocks are driven to the pads where they are buffered and driven off chip.

[1] where X is a clock section symbol.

17.3.2 Global Clock Distribution

In this two stage distribution scheme, clock generation and distribution are very tightly controlled at the global level. Delays seen by each section are minimized and equalized to reduce global skew. Global clock signals have matched buffer delays from the generator, matched interconnect, and matched section loads.

Load matching on global clock signals is implemented using *dummy loads* - MOSFET capacitors added to global distribution lines to balance section driver loads seen by the global clock generator. Dummy loads are added to global clock signals at each section input, matching the section load on that signal with the most heavily loaded clock signal at that section input. Global routing of the clock signals is carefully controlled to both minimize RC delays and to match the delays of all signals arriving at a common receiver.

To provide flexibility in global clock distribution, global clock signals are organized into four groups. Interconnect and loads are matched between the signals within each group. The four groups are designed to have very similar edge rates and delay characteristics. These groups consist of:

1. K%PHI_1:4_H - active high CPU clocks
2. K%PHI_1:4_L and K%PHI_12:41_H - active low and double phase CPU clocks
3. K_GLB%PHI12:41_OUT_H - double phase NDAL clocks
4. K_P%PHI_3E_H and K_V%PHI_3E_H - special CPU clocks

17.3.3 Section Clock Distribution

Section clock distribution rules are more flexible than global rules to allow for stringent routing requirements at the section level. Primary requirements for section-level distribution are 1) maximum 125 pS RC delay between section drivers and any receiver, and 2) adherence to NVAX methodology which specifies the use of only fully complementary receivers. A detailed description of the rules relating to the use of the NVAX on-chip clocks can be found in the NVAX CPU Chip Design Methodology document.

17.3.4 Global Clock Waveforms

Eight single phase and four double phase clock signals are globally distributed on the chip. Four NDAL clocks are driven to the pads where they are buffered and driven off chip. The single and double phase CPU clocks have a period of one NVAX cycle. The NDAL clock cycle is three NVAX cycles in length. Both rising and falling clock transitions occur at the boundaries of each of the four phases of an NVAX clock cycle. Waveforms for the globally-distributed clock signals are shown in Figure 17–4. The use of these global clock signals is RESTRICTED to interconnecting the section clock drivers.

Clock signals K_P%PHI_3E_L and K_V%PHI_3E_L are used for sense amplifier timing within the Pcache and VIC, respectively. These signals are "early" versions of K%PHI_3_H and are carefully tuned in relation to other clock signals. For this reason, waveforms for these clocks are not depicted in Figure 17–4. These signals are discussed further in the next section.

Figure 17–4: Global Clock Waveforms

```
                     NVAX        NVAX       NVAX
                     Cycle       Cycle      Cycle
                 |           |           |           |
                 |--|--|--|--|--|--|--|--|--|--|--|--|
                 |           |           |           |

    K%PHI_1_H    /~~\________/~~\________/~~\________/
    K%PHI_2_H    ___/~~\________/~~\________/~~\______
    K%PHI_3_H    ______/~~\________/~~\________/~~\___
    K%PHI_4_H    \________/~~\________/~~\________/~~\
    K%PHI_1_L    \__/~~~~~~~~\__/~~~~~~~~\__/~~~~~~~~\
    K%PHI_2_L    ~~~\__/~~~~~~~~\__/~~~~~~~~\__/~~~~~~
    K%PHI_3_L    ~~~~~~\__/~~~~~~~~\__/~~~~~~~~\__/~~~
    K%PHI_4_L    /~~~~~~~~\__/~~~~~~~~\__/~~~~~~~~\__/
    K%PHI_12_H   /~~~~~\_____/~~~~~\_____/~~~~~\_____/
    K%PHI_23_H   ___/~~~~~\_____/~~~~~\_____/~~~~~\___
    K%PHI_34_H   \_____/~~~~~\_____/~~~~~\_____/~~~~~\
    K%PHI_41_H   ~~~\_____/~~~~~\_____/~~~~~\_____/~~~
K_GLB%PHI_12_H   /~~~~~~~~~~~~~~~~~\_________________/
K_GLB%PHI_23_H   _________/~~~~~~~~~~~~~~~~~\_________
K_GLB%PHI_34_H   \_________________/~~~~~~~~~~~~~~~~~\
K_GLB%PHI_41_H   ~~~~~~~~~\_________________/~~~~~~~~~
```

17.3.5 Section Clock Waveforms

The section clocks are buffered versions of the globally distributed clock signals. Ten sections on the chip receive clocks **K%PHI_1:4_H**, while the Fbox is the only receiver of **K%PHI_1:4_L** and **K%PHI_12:41_H**. NDAL clocks are received only at the pads and are driven off chip as PHI12:41_OUT_H.

Clock signals **K_P%PHI_3E_L** and **K_V%PHI_3E_L** are received only in the Pcache and VIC sections, respectively. These clocks are used to trigger sense amplifiers and must be tuned such that their buffered, section level edges precede the normal section level phase 3 edges (e.g. K_P%PHI_3_H and K_V%PHI_3_H) by approximately 1.2 nS.

All section buffers have identical internal delays. To insure this, standard clock drivers are used in each section (except the Fbox). The standard clock driver is designed to be used in a distributed fashion: multiple identical parallel drivers are used, with inputs, outputs, and primary internal nodes being individually strapped together within each section.

17.3.6 Clock Skews and Rise/Fall Times of the Section Clocks

Because of the tightly controlled delays in the first stage of the distribution network, clock skew specifications are the same in most sections of the chip. The only exception to this is in the Fbox, where leverage of the layout from the Rigel Fbox necessitates specification of a lower-tolerance skew. This higher skew figure is due to larger allowable RC delays in the Fbox section level clock distribution network.

Table 17–2 specifies the skews and rise/fall times for the edges of the single phase clock signals. These values are for a TT part running at 100°C and 3.0 volts. *Clock Skew* is the uncertainty in time from when any clock edge crosses the 50% Vdd point to when any other clock edge crosses the 50% Vdd point. The rise and fall times are measured from the 10% to 90% points of the full voltage transition of the clock signal. Adjacent clock phases can overlap or underlap due to clock skew.

Table 17–2: Skews and Rise/Fall Times[1]

Skew Within Any Section[2]	Skew Within Fbox	Skew Between Any Two Sections[2]	Skew Between Fbox and Any Section	Rise/Fall Times
0.5 nS	1.0 nS	0.5 nS	1.0 nS	0.5 nS

1 2

17.4 The NDAL interface timing system

17.4.1 NDAL Clocks

The NVAX CPU provides four double phase low skew clocks that are used by the memory interface to communicate with the CPU via the NDAL. The NDAL runs at one third the speed of the internal CPU cycle. The NDAL clocks are generated by dividing the internal clock frequency by three. The interconnect used for these signals must be well controlled to avoid excessive delay, ringing, and skew.

The relationship of the four clocks to the internal CPU clock cycle is shown in Figure 17–5. The timing diagram also indicates the timing of the NDAL signals. The NDAL changes in Φ_{12}, is valid during Φ_3, and goes tristate in Φ_4. All NDAL signal transitions are referenced to the RISING transitions of the clocks.

[1] These skews are not valid for the NDAL clocks. See Section 1.4 for specific NDAL clock skew information.

[2] Excluding the Fbox.

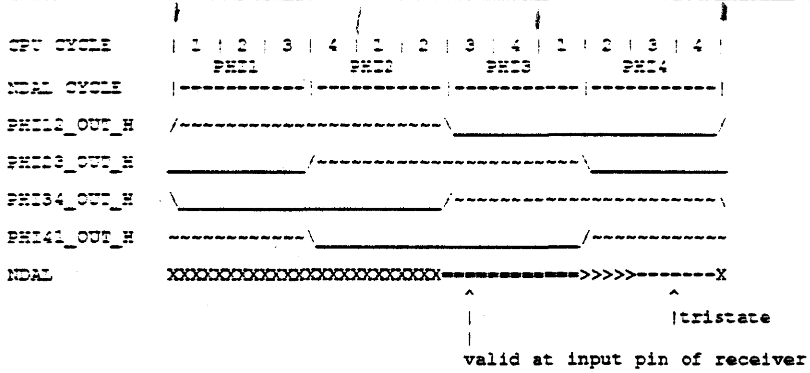
17.4.2 Controlling Inter-Chip Clock Skew

The distribution of the NDAL clocks across the module is critical to the performance and functionality of the CPU. At the specified operating frequencies of the CPU, the module interconnect acts as a transmission line. It has a characteristic impedance and delay. The interconnect used for the clock signals must be carefully matched to avoid skew. Note that skews and signal delays are measured from the point where the waveform reaches VDD/2 (nominally 1.65V).

MODULE INTERCONNECT

Detailed information concerning the design of the module interconnectivity can be found in the NVAX Module Signal Integrity Handbook.

Figure 17–5: Relationship of Internal and NDAL Clock Cycles



17.4.2.1 Self Skew

Each NDAL clock is distributed to a number of receivers on the CPU board. In a perfect electrical environment each chip would receive the clocks at exactly the same time. Unfortunately, due to mismatched interconnect lengths and variations in the electrical properties of the interconnect, a clock signal will not arrive at the different receivers at the same time. For example, refer to Figure 17–6. The clock signal is driven from the NVAX CPU to four clock receivers. Due to interconnect length mismatches it will be received at points A, B, C, and D at different times.

Figure 17–6: Self Skew

```
+----------+      PHI12         B +-----------+
|       out+-----------+---------->|           |
|NVAX       |           |          | receiver  |
|           | A         |          |     1     |
|         in|<----------+          |           |
+----------+            |          +-----------+
                        |
                        |
                        |
+----------+            |          +-----------+
|           | D         |        C |           |
|receiver  |<-----------+---------->| receiver  |
|   2       |                      |     3     |
+----------+                       +-----------+
```

The maximum difference in the arrival time of a particular clock transition at different locations is defined as the *self-skew* of the clock. Self-skew is the maximum possible difference between the actual clock transition and the specified clock transition. For the NVAX CPU to operate at its maximum performance the following rules must be obeyed.

1. The rising transition of each NDAL clock occurs at any receiver within 1.0ns of when it occurs at any other receiver. For example, refer to the diagram above. The Φ_{12} rising transition occurs at point A, point B, point C and point D, and the transitions at each separate point occur within 1.0ns of the transitions at every other point.
2. Rule 1 must also hold for NDAL falling edge transitions.

These rules imply that if a clock transition appears at one receiver 0.5ns before the specified time, the same clock transition cannot appear at another receiver more than 0.5ns after the specified time: this would violate rule 1.

17.4.2.2 Inter-Clock Skew

At the clock receivers, each NDAL clock transition is specified to appear at some time relative to any of the other NDAL clock transitions. In an ideal design, all clock transitions would occur at the specified time. Unfortunately, due to device, processing, and interconnect mismatches, the clock signals will arrive at times different from those specified. The uncertainty in arrival times is defined as the inter-clock skew. For the NVAX CPU to operate at its maximum performance the following inter-clock skew rules must be obeyed.

1. The skew between any two rising NDAL clock transitions at any two receivers is +/-0.5ns. For example, if the transitions are defined to be 15ns apart, the clock design guarantees that they are between 14.5 and 15.5 ns apart.
2. Skew between falling clock transitions is +/-0.5ns.
3. The skew between a rising transition and a falling transition is +/-0.75ns.

17.4.3 Driving and Receiving NDAL signals

Detailed information regarding NDAL clocking and NDAL skew considerations can be found in Chapter 3.

17.4.4 Information Transfer between the NDAL clock system and the on-chip clock system

Detailed information regarding information transfer between the NDAL clock system and the on-chip clock system can be found in Chapter 13.

17.5 Initializing the NVAX system.

ASYNC_RESET_L is an asynchronous input to the NVAX chip. It is used to force the NVAX CPU into a known state. The assertion of **ASYNC_RESET_L** occurs during NVAX system initialization. **ASYNC_RESET_L** must be asserted for a minimum of 7 NDAL cycles.

SYS_RESET_L is both an asynchronous and synchronous output. **SYS_RESET_L** is asynchronously asserted whenever **ASYNC_RESET_L** is asserted. When asserted, it places the NVAX system chips in their initial power-up states.

SYS_RESET_L is asserted for a minimum of 7 NDAL cycles. The deassertion of the signal is synchronized to the NDAL clocks. It is deasserted on the rising edge of **PHI12_OUT_H** and is valid at the NDAL receivers in time to be latched in NDAL Φ_4. Figure 17–7 shows the relationship between **ASYNC_RESET_L** and **SYS_RESET_L** signals.

Figure 17–7: System Reset Timing

```
                                                             *************************************************
                                                             |  NDAL CYCLE   |  NDAL CYCLE   |  NDAL CYCLE   |
                                                             |               |               |               |
                                                             | P1| P2| P3| P4| P1| P2| P3| P4| P1| P2| P3| P4|
                                                             |---|---|---|---|---|---|---|---|---|---|---|---|
                                                             +   |   |   |   +   |   |   |   +   |   |   |   +
ASYNC_RESET_L  --\\\_________________   ..   ..  __________________/////--------------------------------------
                    <-- Asserted for a minimum of 7 NDAL Cycles ---> |   |   +   |   |   |   +   |   |   |   +
SYS_RESET_L    --\\\\\\\\\\\_________   ..   ..  ____________________________________________//////////-------
                                                             +   |   |   |   +   |   |   |   +   |   |   |   +
                 ^                               PHI12_OUT_H /-------\_______/-------\_______/-------\_______/
                 | ASYNC_RESET_L asynchronous                +   |   |   |   +   |   |   |   +   |   |   |   +
                 | assertion causes asynchronous PHI23_OUT_H ____/-------\_______/-------\_______/-------\____
                 | assertion of SYS_RESET_L.                 +   |   |   |   +   |   |   |   +   |   |   |   +
                                                 PHI34_OUT_H \_______/-------\_______/-------\_______/-------\
                                                             +   |   |   |   +   |   |   |   +   |   |   |   +
                                                 PHI41_OUT_H ----\_______/-------\_______/-------\_______/----
                                                             +   |   |   |   +   |   |   |   +   |   |   |   +
                                                             *************************************************
```

17.5.1 Internal NVAX Reset

The **ASYNC_RESET_L** pin is used to generate several internal reset signals which reset various parts of the NVAX chip. **ASYNC_RESET_L** is synchronized with NDAL Φ_3, then latched after settling with NDAL Φ_1. This synchronized signal is piped to NVAX Φ_4 to produce **K_PAD%SYNC_RESET**. The internal, buffered version of **ASYNC_RESET_L** is **K_PAD%ASYNC_RESET**.

To satisfy various logic timing constraints, several reset signals are produced and distributed throughout the NVAX chip. The primary internal NVAX reset is **K%RESET**. This signal is asserted asynchronously and deasserted synchronously following assertion/deassertion of **ASYNC_RESET_L** or **DISABLE_OUT_L**, or during the BSR External Test. Buffered versions of **K%RESET** are used by the Ebox, Ibox, VIC, and Fbox to reset local logic. Detailed information regarding the functions of **DISABLE_OUT_L** and the BSR External Test can be found in the NVAX Testability Specification.

The Mbox, Pcache, and Cbox (excluding BIU) receive buffered versions of **K_MC%RESET**. This signal functions the same as **K%RESET**, except it is also asserted following an Ebox S3 timeout (see individual box chapters of this specification for detailed information).

The I/O Pad logic receives buffered versions of **K%EXT_RESET**. This signal is the same as **K%RESET**, except it is not asserted with **DISABLE_OUT_L** or during BSR External Test as **K%RESET** is.

K_CE%RESET is asserted during NDAL Φ_3 and piped to NVAX internal Φ_1. A buffered version of this is used to reset BIU logic in the Cbox to the proper NDAL sequencing state during the reset sequence (see CBOX chapter for detailed information).

17.5.2 Generation of Clocks During Power-up

The NVAX chip generates its internal clocks and the NDAL clocks by dividing down a high frequency external oscillator signal. The external system oscillator is powered from the module 5 volt power supply. Its clock signals must be valid before 3 volt power is supplied to the NVAX chip. The oscillator takes a maximum of 10 mS of initialization time before its clocks can be considered free running. Hence, the module power supply must be designed to guarantee that the 3 volt supply is not valid until 10 mS after the 5 volt supply is stable.

The NVAX clock generator derives free running clocks from the external oscillator clock. The clock generator is self-initializing and is not affected by the assertion of **ASYNC_RESET_L**, except for clock generator reset test features (see CLOCK GENERATOR RESET, next section). The clock generator requires a maximum of 3 oscillator clock cycles to initialize itself after the 3 volt module power supply has become valid. Figure 17–8 shows the NVAX Chip and external oscillator power-up sequence.

Figure 17–8: Clock State During Initial Power-up

```
*******************************************************************************************************************
                                                    |   NVAX CLOCK  |   NVAX CLOCK  |   NVAX CLOCK  |   NVAX CLOCK
                                                    | P1| P2| P3| P4| P1| P2| P3| P4| P1| P2| P3| P4| P1| P2| P3| P
                                                     ---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|--
                                                    +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |
5V_POWER __//////~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                 <---10 mS min--->                  +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |
3V_POWER ____________________/////~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                                    +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |
OSC_TEST_H XXXXXXXXXXXXXXXXXXXXXXX_________________________________________________________________________________
            <-OSC indeterminate->                   +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |
OSC_H    XXXXXXXXXXXXXXXXXXXXXXXX/~\_/~\   ...  ~\_/~\_/~\_/~\_/~\_/~\_/~\_/~\_/~\_/~\_/~\_/~\_/~\_/~\_/~\_/~\_/~\_
                                 <3 OSC Cycles max.>+   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |
OSC_L    XXXXXXXXXXXXXXXXXXXXXXXX\_/~\_/ ...    _/~\_/~\_/~\_/~\_/~\_/~\_/~\_/~\_/~\_/~\_/~\_/~\_/~\_/~\_/~\_/~\_/~
                                 <3 OSC Cycles max.>+   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |
K%PHI_12_H  (Internal Chip Clock) XXXXXXXXXXXXXXXXXX~~~~~~~~\_______/~~~~~~~\_______/~~~~~~~\_______/~~~~~~~\______
                                                    +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |
K%PHI_23_H  (Internal Chip Clock) XXXXXXXXXXXXXXXXXX____/~~~~~~~\_______/~~~~~~~\_______/~~~~~~~\_______/~~~~~~~\__
                                                    +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |
K%PHI_34_H  (Internal Chip Clock) XXXXXXXXXXXXXXXXXX________/~~~~~~~\_______/~~~~~~~\_______/~~~~~~~\_______/~~~~~~
                                                    +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |
K%PHI_41_H  (Internal Chip Clock) XXXXXXXXXXXXXXXXXX~~~~\_______/~~~~~~~\_______/~~~~~~~\_______/~~~~~~~\_______/~~
                                                    +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |
PHI_12_OUT (NDAL System Clock)    XXXXXXXXXXXXXXXXXX/~~~~~~~~~~~~~~~~~~~~~~~\_______________________/~~~~~~~~~~~~~~
                                                    +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |
PHI_23_OUT (NDAL System Clock)    XXXXXXXXXXXXXXXXXX____________/~~~~~~~~~~~~~~~~~~~~~~~\_______________________/~~
                                                    +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |
PHI_34_OUT (NDAL System Clock)    XXXXXXXXXXXXXXXXXX________________________/~~~~~~~~~~~~~~~~~~~~~~~\______________
                                                    +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |
PHI_41_OUT (NDAL System Clock)    XXXXXXXXXXXXXXXXXX~~~~~~~~~~~~\_______________________/~~~~~~~~~~~~~~~~~~~~~~~\__
                                                    +   |   |   |   +   |   |   |   +   |   |   |   +   |   |   |
*******************************************************************************************************************
```

17.5.3 Clock Generator Reset

The NVAX chip incorporates a clock generator reset feature for use in verifying chip timing. The generator can be reset to a known cycle and phase in order to verify various signals against their specified timing.

WARNING

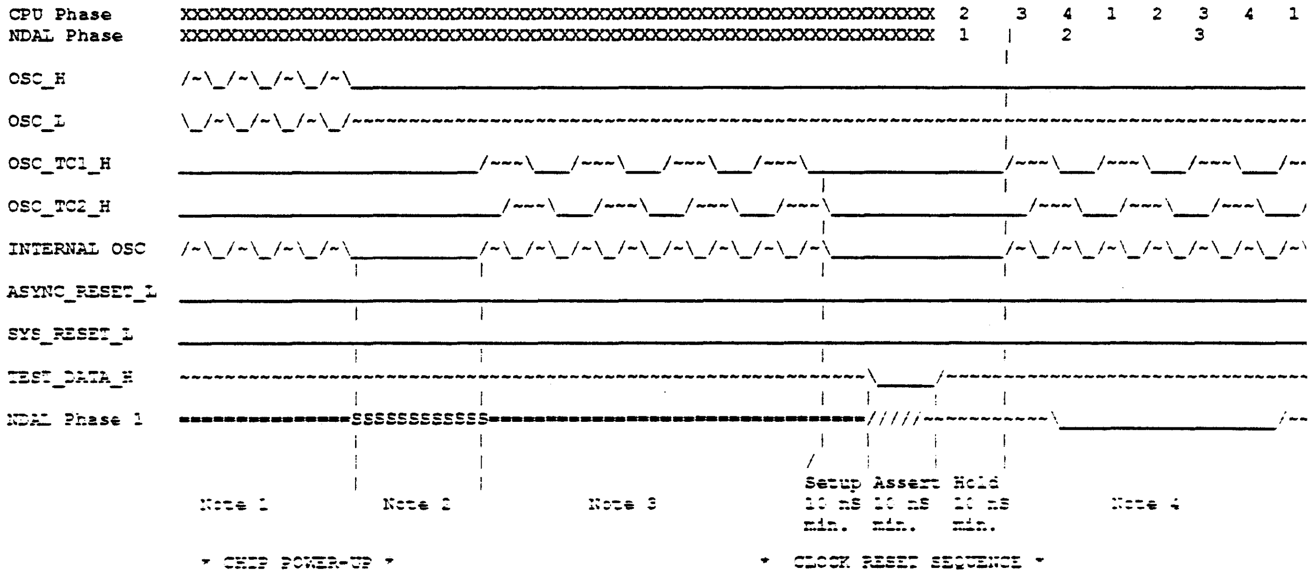
> Use of the clock generator reset feature must follow these specific sequencing and timing constraints. Deviation from these specifications will have undesirable results, and *can result in physical damage to the NVAX chip*. Contact a member of the NVAX clock design team for further information about this feature.

Figure 17–9 shows the proper signal timing for effecting a reset of the clock generator. To begin the clock generator reset sequence, the chip is powered up using normal high speed oscillator inputs supplied through **OSC_H** and **OSC_L**. This is the normal powerup mode, and allows the internals of the chip to reach a deterministic operating state.

Following a normal powerup reset sequence, the oscillator input is turned off briefly (1-2 mS) to switch the oscillator input to the test clocks. Following the switch to the test clocks, the chip is again reset to restore any internal state lost during the test clock switch. Note that **ASYNC_RESET_L** is held asserted through the duration of the clock generator reset sequence.

Following this second chip reset sequence, the test clocks are stopped briefly (500 nS MAX). The states of test clocks **OSC_TC1_H** and **OSC_TC2_H** when stopped must be the same, either both high or both low (as shown). **TEST_DATA_H** should be driven low as shown in Figure 17–9 to effect the clock generator reset. This immediately places the clock generator into NVAX Φ_2 and NDAL Φ_1. **TEST_DATA_H** is then driven high and clocking of the chip is resumed. On the first oscillator cycle following resumption of clocking, the generator will transition into NVAX Φ_3 and begin normal sequencing. **AYSYNC_RESET_L** must remain asserted for at least 7 NDAL cycles following resumption of clocking.

Figure 17–9: Clock Generator Reset Timing



- **K_SEC4OSC1_H** is the internal master clock produced from either the **OSC_H** and **OSC_L** inputs, or the **OSC_TC1_H** and **OSC_TC2_H** inputs. **OSC_TEST_H** is used to select the clock source as described in this clock specification.

- S indicates a static (non-changing) NDAL Φ_1.

Timing Notes:

1. ECL pin inputs **OSC_H** and **OSC_L** must be used to supply clocks to chip prior to and during power-up. Inputs OSC_TC1_H and OSC_TC2_H must be held low in order to prevent latch-up.
2. Switch to test clocks OSC_TC1_H and OSC_TC2_H. Start measure out 1pat on chip tester.
3. Clocks restarted to restore internal chip signals prior to clock-reset sequence.
4. **ASYNC_RESET_L** must remain asserted for a minimum of 7 NDAL cycles following restart of clocks.

17.6 NVAX Clock Section Signal/Pin Dictionary

17.6.1 Schematic - Behavioral Translation

Schematic Name[1]	Behavioral Model Name[1]
- Signals	
K%EXT_RESET	K%EXT_RESET
K%PHI_1_H	K%PHI_1_H
K%PHI_2_H	K%PHI_2_H
K%PHI_3_H	K%PHI_3_H
K%PHI_4_H	K%PHI_4_H
K%PHI_1_L	K%PHI_1_L
K%PHI_2_L	K%PHI_2_L
K%PHI_3_L	K%PHI_3_L
K%PHI_4_L	K%PHI_4_L
K%PHI_12_H	K%PHI_12_H
K%PHI_23_H	K%PHI_23_H
K%PHI_34_H	K%PHI_34_H
K%PHI_41_H	K%PHI_41_H
K%RESET	K%RESET
K_CE%RESET	K_CE%RESET
K_GLB%PHI12_OUT_H	K%NDAL_PHI_12_H
K_GLB%PHI23_OUT_H	K%NDAL_PHI_23_H
K_GLB%PHI34_OUT_H	K%NDAL_PHI_34_H
K_GLB%PHI41_OUT_H	K%NDAL_PHI_41_H
K_MC%RESET	K_MC%RESET
K_P%PHI_3E	non-existent[2]
K_PAD%ASYNC_RESET	K_PAD%ASYNC_RESET
K_PAD%SYNC_RESET	K_PAD%SYNC_RESET
K_SEC%OSC1_H	module call[3]
K_V%PHI_3E	non-existent[2]
- Pins	
ASYNC_RESET_L	P%ASYNC_RESET_L
DISABLE_OUT_L	P%DISABLE_OUT_L
OSC_TEST_H	P%OSC_TEST_H

[1] Signals without specified assertion levels may exist in _H and/or _L versions.

[2] These signals are not modeled in the behavioral code.

[3] Any transition is represented in behavioral model by a call to routine n_%master_clock_transition.

Schematic Name[1]	Behavioral Model Name[1]
OSC_H	P%OSC_H
OSC_L	P%OSC_L
OSC_TC1_H	P%OSC_TC1_H
OSC_TC2_H	P%OSC_TC2_H
PHI12_OUT_H	P%PHI12_OUT_H
PHI23_OUT_H	P%PHI23_OUT_H
PHI34_OUT_H	P%PHI34_OUT_H
PHI41_OUT_H	P%PHI41_OUT_H
SYS_RESET_L	P%SYS_RESET_L
TEST_DATA_H	P%TEST_DATA_H

[1] Signals without specified assertion levels may exist in _H and/or _L versions.

17.6.2 Behavioral - Schematic Translation

Behavioral Model Name[4]	Schematic Name[4]
- Signals	
K%EXT_RESET	K%EXT_RESET
K%PHI_1_H	K%PHI_1_H
K%PHI_2_H	K%PHI_2_H
K%PHI_3_H	K%PHI_3_H
K%PHI_4_H	K%PHI_4_H
K%PHI_1_L	K%PHI_1_L
K%PHI_2_L	K%PHI_2_L
K%PHI_3_L	K%PHI_3_L
K%PHI_4_L	K%PHI_4_L
K%PHI_12_H	K%PHI_12_H
K%PHI_23_H	K%PHI_23_H
K%PHI_34_H	K%PHI_34_H
K%PHI_41_H	K%PHI_41_H
K%RESET	K%RESET
K_CE%RESET	K_CE%RESET
K%NDAL_PHI_12_H	K_GLB%PHI12_OUT_H
K%NDAL_PHI_23_H	K_GLB%PHI23_OUT_H
K%NDAL_PHI_34_H	K_GLB%PHI34_OUT_H

[4] Signals without specified assertion levels may exist in _H and/or _L versions.

Behavioral Model Name[4]	Schematic Name[4]
K%NDAL_PHI_41_H	K_GLB%PHI41_OUT_H
K_MC%RESET	K_MC%RESET
K_PAD%ASYNC_RESET	K_PAD%ASYNC_RESET
K_PAD%SYNC_RESET	K_PAD%SYNC_RESET
- Pins	
P%ASYNC_RESET_L	ASYNC_RESET_L
P%DISABLE_OUT_L	DISABLE_OUT_L
P%OSC_TEST_H	OSC_TEST_H
P%OSC_H	OSC_H
P%OSC_L	OSC_L
P%OSC_TC1_H	OSC_TC1_H
P%OSC_TC2_H	OSC_TC2_H
P%PHI12_OUT_H	PHI12_OUT_H
P%PHI23_OUT_H	PHI23_OUT_H
P%PHI34_OUT_H	PHI34_OUT_H
P%PHI41_OUT_H	PHI41_OUT_H
P%SYS_RESET_L	SYS_RESET_L
P%TEST_DATA_H	TEST_DATA_H

[4] Signals without specified assertion levels may exist in _H and/or _L versions.

17.7 Revision History

Table 17–3: Revision History

Who	When	Description of change
Bill Bowhill	28-Jan-1990	Initial Release
Tim Fischer	28-Jan-1991	Pass 1 Updates Complete

Chapter 18

Performance Monitoring Facility

18.1 Overview

The NVAX CPU chip contains a facility by which privileged software may obtain performance information about the dynamic behavior of the CPU. The facility is implemented with a combination of hardware and microcode, and controlled by software using privileged instructions.

Two 64-bit performance counters called PMCTR0 and PMCTR1 are maintained in memory for each CPU in the system. The lower 16 bits of each counter are implemented in hardware in the CPU, and at specified points, microcode updates the quadwords in memory with the contents of the hardware counters.

The performance monitoring facility may be configured by privileged software to count a number of events in the system, from which performance analysis data such as cache and TB hit rates, cycles-per-instruction, and stall frequencies may be calculated.

18.2 Software Interface to the Performance Monitoring Facility

The performance monitoring facility makes use of a data structure in memory, and must be configured and enabled via a location in the System Control Block, processor register references, and the LDPCTX instruction.

18.2.1 Memory Data Structure

The two 64-bit performance counters for each CPU are maintained in a data structure in memory. This data structure consists of a pair of quadwords for every CPU in the system. The physical address of the base of the data structure is obtained from offset 58 (hex) in the System Control Block. The format of this location is shown in Figure 18–1.

Figure 18–1: Performance Monitoring Data Structure Base Address

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|          Physical Address of Performance Monitoring Data Structure           |SBZ 0  1  1| :SCB+58(he
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

NOTE

> A quadword-aligned physical base address is constructed by clearing the lower 3 bits of the longword fetched from offset 58 (hex) in the SCB. Microcode will not update the block in memory unless bits <2:0> of this longword contain 011 (binary). If these bits are found to contain another value, a machine check with code MCHK_PMF_CONFIG is performed to notify software that the performance monitoring facility was incorrectly configured. If is strongly suggested that the physical address be at least octaword aligned, and preferably page aligned.

The address of the pair of quadwords for an individual CPU is computed by shifting the CPUID value left 4 bits and adding this value to the base address. This calculation is shown in equation form below (all numbers in these equations are hex).

$$phys\_base\_addr = SCB[58]\ AND\ FFFFFFF0;$$

$$phys\_block\_addr = \{\ CPUID\ LSHIFT\ 4\ \} + phys\_base\_addr;$$

The format of the pair of quadwords for each CPU is shown in Figure 18–2.

Figure 18–2: Per-CPU Performance Monitoring Data Structure

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                       PMCTR0, low longword                                   | :+00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                       PMCTR0, high longword                                  | :+04
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 63 62 61 60|59 58 57 56|55 54 53 52|51 50 49 48|47 46 45 44|43 42 41 40|39 38 37 36|35 34 33 32
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                       PMCTR1, low longword                                   | :+08
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                       PMCTR1, high longword                                  | :+12
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 63 62 61 60|59 58 57 56|55 54 53 52|51 50 49 48|47 46 45 44|43 42 41 40|39 38 37 36|35 34 33 32
```

18.2.2 Memory Data Structure Updates

When the performance monitoring facility is enabled, the memory data structure is updated from the hardware counters if the PMCTR0 counter is more than half full and the current processor IPL is below 1B (hex), if a LDPCTX instruction is executed and the PME bit in the new PCB is off, or if the performance monitoring facility is disabled via a write to the PME processor register. The PME bit is internally implemented as ECR<PMF_ENABLE>, with conversion handled by microcode.

When the PMCTR0 counter reaches half full, an interrupt at IPL 1B (hex) is requested. This interrupt request is serviced like any other interrupt if the IPL of the processor is below that of the interrupt request IPL. Like any other interrupt, it is serviced between instructions (or in the middle of the interruptable string instructions). Unlike other interrupts, the performance monitoring interrupt is serviced entirely by microcode, with no software interrupt handler required.

When a performance monitoring interrupt occurs, microcode temporarily disables the facility, reads and clears the hardware counters, then updates the memory data structure with the hardware counts. The facility is then re-enabled, the interrupt is dismissed, and the interrupted instruction stream is restarted.

NOTE

> Although the performance monitoring facility is disabled during the memory update process, it is re-enabled for the restart of the interrupted instruction stream. Therefore, depending on what events were selected, the facility may count events that are part of the restart process.

At the maximum rate (one increment every 14ns CPU cycle), an interrupt is requested every 459 microseconds.

If a LDPCTX is executed and the PME bit in the new PCB is off, or if the performance monitoring facility is disabled via a write to the PME processor register, the microcode disables the performance monitoring facility, reads and clears the hardware counters, and updates the memory data structure for the CPU with the hardware counts.

NOTE

> The hardware counters are not cleared, and the memory data structures are not updated when the performance monitoring facility is disabled via a direct write to ECR<PMF_ENABLE>.

18.2.3 Configuring the Performance Monitoring Facility

Before the performance monitoring facility is enabled, software must select the source of the event to be counted. This is accomplished first by selecting the box that reports the event, and then by selecting the event that is to be counted. The box section is made by writing to the PMF_PMUX field in the ECR processor register, as indicated by Table 18–1.

Table 18–1: Performance Monitoring Facility Box Selection

ECR<PMF_PMUX> (binary)	Source of Information
00	Ibox
01	Ebox
10	Mbox
11	Cbox

The event selection within the box is made by writing to a processor register within the box, as described in subsequent sections, and in the box chapters elsewhere in this specification.

The hardware used to implement the 16-bit counters is constructed such that the PMCTR1 counter increments only if both its selected event, and the PMCTR0 selected event are true simultaneously. As such, PMCTR1 is a strict subset of PMCTR0. As a result, some combinations of event selections will not cause PMCTR1 to be incremented. In some boxes, the event selection is specified in such a way that compatible events are automatically selected. In other boxes, the user must specify compatible events. Where they are required, compatible events are described in the sections below.

18.2.3.1 Ibox Event Selection

The Ibox reports only one event, so if the Ibox is selected, that event is also selected. The Ibox inputs to the PMCTR0 and PMCTR1 hardware counters are shown in Table 18–2

Table 18–2: Ibox Event Selection

PMCTR0 Input	PMCTR1 Input	Description; Use
VIC Access	VIC Hit	VIC hits compared to total VIC accesses; VIC hit ratio.

18.2.3.2 Ebox Event Selection

The Ebox reports several events, as selected by the PMF_EMUX field in the ECR processor register. The Ebox inputs to the PMCTR0 and PMCTR1 counters are shown in Table 18–3.

Table 18–3: Ebox Event Selection

ECR<PMF_EMUX> (binary)	PMCTR0 Input	PMCTR1 Input	Description; Use
000	Cycles	S3 Stall	S3 stalls (source queue, MD, Wn, Fbox scoreboard hit, Fbox input) compared to total cycles; S3 stalls per unit time.
001	Cycles	EM+PA queue Stall	EM latch and PA queue stalls compared to total cycles; EM+PA queue stalls per unit time.
010	Cycles	Instruction Retire	Ebox and Fbox instructions retired compared to total cycles; CPI.
011	Cycles	Total stall	Total Ebox stalls compared to total cycles; Stalls per unit time.
100	Total stall	S3 Stall	S3 stalls compared to total stalls; S3 stalls as a percentage of all stalls.
101	Total stall	EM+PA queue Stall	EM latch and PA queue stalls compared to total stalls; EM and PA queue stalls as a percentage of all stalls.

Table 18–3 (Cont.): Ebox Event Selection

ECR<PMF_EMUX> (binary)	PMCTR0 Input	PMCTR1 Input	Description; Use
111	S5 Microword event	S5 Microword event	Number of times a microinstruction whose MISC field contained INCR.PERF.COUNT reached S5. By using the patchable control store, one may count microcode events by setting the MISC field of selected microwords to this value. If this event is selected, writing to the PMFCNT processor register will increment the counters via the MISC field decode.

18.2.3.3 Mbox Event Selection

The Mbox reports several events, as selected by the PMM field in the PCCTL processor register. The Mbox inputs to the PMCTR0 and PMCTR1 counters are shown in Table 18–4.

Table 18–4: Mbox Event Selection

PCCTL<PMM> (binary)	PMCTR0 Input	PMCTR1 Input	Description; Use
000	S0 I-stream TB access	S0 I-stream TB hit[1]	TB hits for S0 I-stream references compared to total TB accesses for S0 I-stream references; S0 I-stream TB hit ratio.
001	S0 D-stream TB access	S0 D-stream TB hit[1]	TB hits for S0 D-stream references compared to total TB accesses for S0 I-stream references; S0 D-stream TB hit ratio.
010	P0/P1 I-stream TB access	P0/P1 I-stream TB hit[1]	TB hits for P0 and P1 I-stream references compared to total TB accesses for P0 and P1 I-stream references; P0/P1 I-stream TB hit ratio.
011	P0/P1 D-stream TB access	P0/P1 D-stream TB hit[1]	TB hits for P0 and P1 D-stream references compared to total TB accesses for P0 and P1 D-stream references; P0/P1 D-stream TB hit ratio.
100	I-stream Pcache access	I-stream Pcache hit	Pcache hits for I-stream references compared to total Pcache accesses I-stream references; I-stream Pcache hit ratio.
101	D-stream Pcache access	D-stream Pcache hit	Pcache hits for D-stream references compared to total Pcache accesses D-stream references; D-stream Pcache hit ratio.
110	—	—	Selection causes UNPREDICTABLE behavior of the performance monitoring hardware.
111	Total reads and writes	Unaligned reads and writes	Unaligned virtual reads and writes compared to total virtual reads and writes; Unaligned references as a percentage of all references.

[1] TB hit count is unconditionally incremented when MAPEN=0

18.2.3.4 Cbox Event Selection

The Cbox reports several events, as selected by the PM_ACCESS_TYPE and PM_HIT_TYPE fields in the CCTL processor register. The Cbox inputs to the PMCTR0 counter are shown in Table 18–5 and the Cbox inputs to the PMCTR1 counter are shown in Table 18–6. For the Cbox, all of the PMCTR1 selections shown in Table 18–6 are compatible with all of the PMCTR0 selections shown in Table 18–5.

Table 18–5: Cbox PMCTR0 Event Selection

CCTL<PM_ACCESS_TYPE> (binary)	PMCTR0 Input
000	Bcache coherency access. PMCTR0 increments when the Bcache processes any coherency request from the NDAL.
001	Bcache coherency READ access. PMCTR0 increments when the Bcache processes a IREAD or DREAD coherency request from the NDAL.
010	Bcache coherency OREAD access. PMCTR0 increments when the Bcache processes an OREAD OR WRITE coherency request from the NDAL.
011	Selection causes UNPREDICTABLE behavior of the performance monitoring hardware.
100	Bcache CPU access. PMCTR0 increments when the Bcache processes any reference from the CPU.
101	Bcache CPU IREAD access. PMCTR0 increments when the Bcache processes an instruction-stream read request from the CPU.
110	Bcache CPU DREAD access. PMCTR0 increments when the Bcache processes an data-stream read, or read-with-modify-intent request from the CPU.
111	Bcache CPU OREAD access. PMCTR0 increments when the Bcache processes a data-stream read lock, write, or write unlock request from the CPU.

Table 18–6: Cbox PMCTR1 Event Selection

CCTL<PM_HIT_TYPE> (binary)	PMCTR1 Input
00	Bcache hit. PMCTR1 increments when a Bcache access results in any hit.
01	Bcache hit owned. PMCTR1 increments when a Bcache access results in an owned hit.
10	Bcache hit valid. PMCTR1 increments when a Bcache access results in a valid hit.
11	Bcache miss owned. PMCTR1 increments when a Bcache access results in a miss in which both the valid and owned bits were set.

18.2.4 Enabling and Disabling the Performance Monitoring Facility

The performance monitoring facility is enabled or disabled by setting or clearing the Performance Monitor Enable (PME) bit in the CPU. This bit may be written in one of three ways: with a write to the PME processor register, by loading a new value with a LDPCTX instruction from the PME bit in the new PCB, or by a direct write of the ECR<PMF_ENABLE> bit.

The format of the PME processor register is shown in Figure 18–3.

Figure 18–3: IPR 3D (hex), PME

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                              SBZ                                           |  | :PME
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                                                                                              |
                                                                                   ENABLE --+
```

If PME<0> is written with a 1, the performance monitoring facility is enabled. If PME<0> is written with a 0, the performance monitoring facility is disabled. Direct writes to ECR<PMF_ENABLE> are similar to writes to PME<0>, with the exception that the hardware counters are not automatically cleared, and the memory counters are not updated on an explicit write to ECR<PMF_ENABLE>.

The CPU PME bit is also loaded by the LDPCTX instruction from PCB+92<31>.

CAUTION

> The longword at offset 58 (hex) from the SCB and the correct unique CPUID value for each CPU must be initialized before the performance monitoring facility is enabled. Failure to do so will result in UNDEFINED behavior of the system.

The CPU PME bit is cleared, and the performance monitoring facility is disabled, at powerup.

18.2.5 Reading and Clearing the Performance Monitoring Facility Counts

In normal operation, microcode automatically updates the memory counters by reading the current value of the hardware counters, adding these values to the memory counters, and clearing the hardware counters. This is the preferred mode of operation.

However, there may be some situations in which software wishes to directly read or clear the hardware counters. The current value of the hardware counters may be read from the PMFCNT processor register, whose format is shown in Figure 18–4.

Figure 18–4: IPR 7B (hex), PMFCNT in PMF Format

```
 31 30 29 28|27 26 25 24|23 22 21 20|19 18 17 16|15 14 13 12|11 10 09 08|07 06 05 04|03 02 01 00
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|         Current Hardware PMCTR1 Value         |         Current Hardware PMCTR0 Value         | :PMFCNT
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

The current value of the 16-bit hardware PMCTR1 counter is returned in PMFCNT<31:16> and the current value of the 16-bit hardware PMCTR0 counter is returned in PMFCNT<15:0>.

The two 16-bit hardware counters may be explicitly cleared by software by writing a 1 to ECR<PMF_CLEAR>. If the counters are explicitly cleared, any outstanding interrupt request is also cleared. It is strongly suggested that the hardware counters not be cleared while the performance monitoring facility is enabled.

If the performance monitoring facility is configured to select the Ebox microword event (ECR<PMF_PMUX>=Ebox, ECR<PMF_EMUX>=S5 microword event, ECR<PMF_ENABLE>=1), a write of any value to the PMFCNT processor register will increment both hardware counters.

TEST NOTE

The performance monitoring facility hardware incrementers may be tested by clearing them via ECR<PMF_CLEAR>, selecting the Ebox S5 microword event, and enabling the facility. Each write to the PMFCNT processor register will then increment both hardware counters, and the result may be observed by reading the PMFCNT register. The interrupt request may be tested by incrementing the PMCTR0 hardware counter into bit<15>, which will cause an interrupt to be requested.

NOTE

If the 16-bit hardware counters are explicitly cleared by writing a 1 to ECR<PMF_CLEAR>, any count in these registers is lost and will not be included in the memory counters.

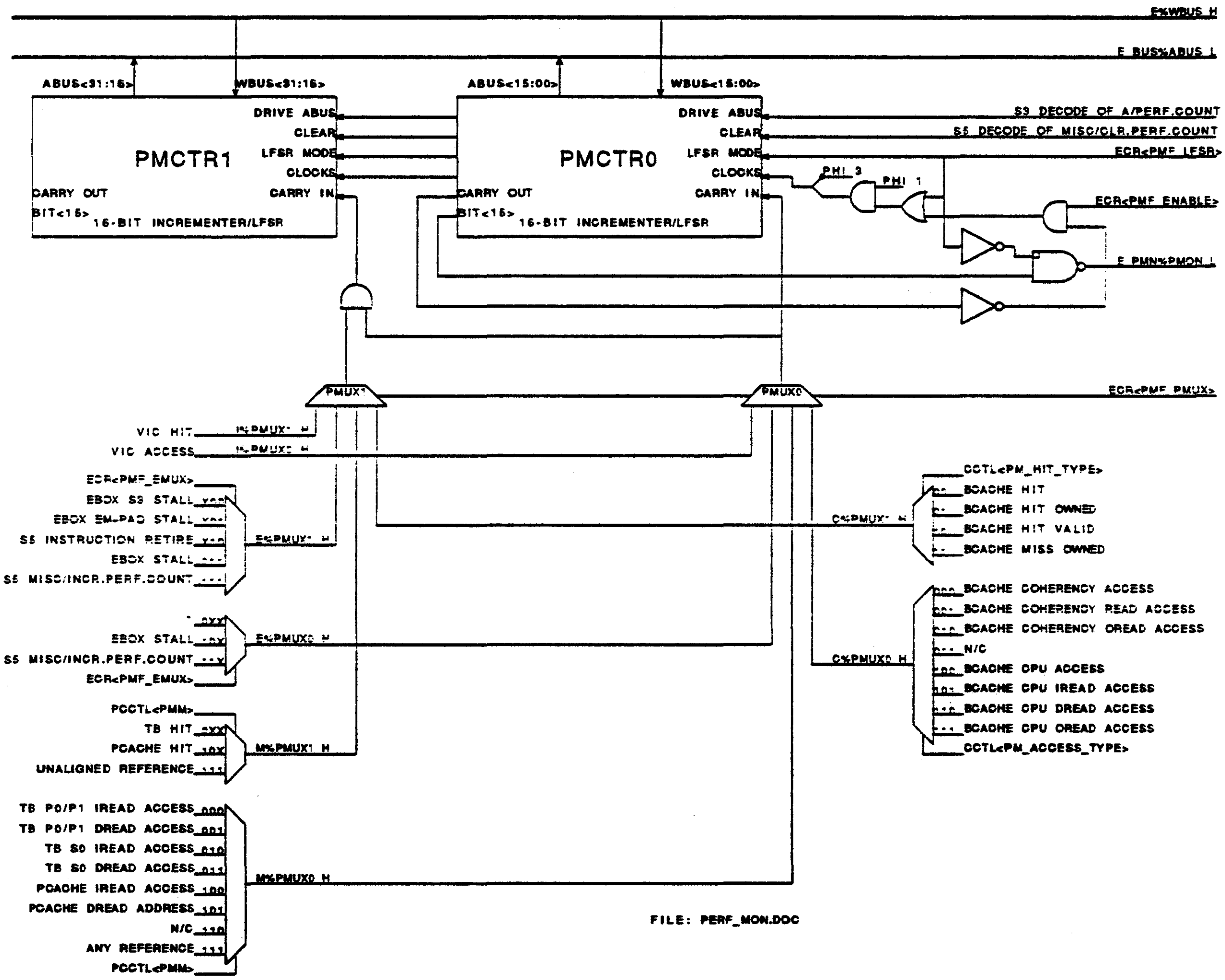
CAUTION

The performance monitoring hardware also provides the WBUS LFSR function under control of ECR<PMF_LFSR>. The operation of the hardware is UNDEFINED if both ECR<PMF_ENABLE> and ECR<PMF_LFSR> are on, or if software uses a single MTPR write to turn off one bit and turn on the other simultaneously. That is, if either bit is on, software must turn off both bits with one MTPR and turn on the other with a second MTPR.

18.3 Hardware and Microcode Implementation of the Performance Monitoring Facility

The performance monitoring facility is implemented via both CPU chip hardware and microcode. A block diagram of the performance monitoring hardware is shown in Figure 18–5.

Figure 18–5: Performance Monitoring Hardware Block Diagram



The lower 16 bits of the PMCTR0 and PMCTR1 performance counters are implemented as two 16-bit incrementers in the Ebox. Both incrementers have a common clear line which is driven from an S5 decode of MISC/CLR.PERF.COUNT, and each has a separate carry-in input to cause an increment in the appropriate counter. The 32-bit concatenated value from the incrementers can be read onto **E_BUS%ABUS_L** (the active-low variant of **E%ABUS_H**), and the upper bit of PMCTR0 is used to generate **E_PMN%PMON_L**, the performance monitoring facility interrupt request.

The PMCTR0 and PMCTR1 carry-in inputs are supplied by PMUX0 and PMUX1, with the PMCTR1 carry-in signal gated with the PMCTR0 carry-in signal. This makes PMCTR1 counter a strict subset of the PMCTR0 counter. Increments of both counters are suppressed if the performance monitoring facility is not enabled, or if the PMCTR0 counter has reached its maximum value.

The top-level selection of events is determined by ECR<PMF_PMUX>, which selects the source to PMUX0 and PMUX1. This selects the source (Ibox, Ebox, Mbox, Cbox) of the carry-in signals to each counter. Distributed in the appropriate boxes are second-level muxes which are selected to provide the actual source of the increment events for PMCTR0 and PMCTR1.

18.3.1 Hardware Implementation

The two 16-bit hardare counters are implemented as side-by-side incrementers in the Ebox datapath (this hardware also implements the Wbus LFSR reducer that is described in the testability section of Chapter 8). The carry-in signals for each of the counters are driven from two 4-to-1 muxes that are selected by ECR<PMF_PMUX>, and which select the appropriate source of inputs to the incrementers.

Logic in the Ibox, Mbox, and Cbox select the appropriate values to drive the two carry-in signals based on processor register fields in the box. The Ebox carry-in signals are selected locally and provide the fourth input to the muxes. The PMCTR1 carry-in signal is forced to be a subset of the PMCTR0 carry-in signal by ANDing the raw PMCTR1 carry-in signal with the PMCTR0 carry-in signal to produce the final PMCTR1 carry-in signal.

Because the PMCTR1 increment is a strict subset of the PMCTR0 increment, the ultimate source of the two carry-in signals align them such that they are valid in the same cycle. For example, if the selcted conditions are IREAD PCACHE ACCESS and PCACHE HIT, these two signals are valid in the same cycle, and they refer to the same reference. Therefore the assertion of IREAD PCACHE ACCESS is delayed until the cycle in which PCACHE HIT is valid. In addition to this, the source of the carry-in signals guarantees that any events that may be retried are only recorded once. For example, a particular Pcache access causes only one increment, even if it is retried multiple times.

When the 16-bit PMCTR0 counter increments into the high-order bit, an interrupt is requested by asserting the E_PMN%PMON_L signal to the interrupt section, unless the hardware is configured to enable LFSR mode. This signal is sampled by edge-sensitive logic, so the interrupt request is maintained until it is cleared by writing a 1 to the appropriate bit in the INT.SYS register, even if the performance monitoring facility hardware counters are subsequently cleared.

When the 16-bit PMCTR0 incrementer reaches its maximum value, subsequent increments of either counter are inhibited by blocking the clocks to the logic when a carry-out is detected from PMCTR0. In normal operation, this should not occur, but the counter may overflow if the interrupt request isn't serviced within several hundred microseconds, as would be the case if software spent an extended period of time a high IPL with the performance monitoring facility enabled.

The 32-bit concatenated value of the two 16-bit hardware incrementers can be read onto E_BUS%ABUS_L when selected by an S3 decode of A/PERF.COUNT. This is the mechanisim by which microcode retrieves the current values of the two incrementers. The 32-bit concatenated value is cleared by an S5 decode of MISC/CLR.PERF.COUNT. The clear is done independent of whether the logic is enabled for performance counting or LFSR mode.

18.3.2 Microcode Interaction with the Hardware

There are several points at which the microcode interacts with the performance monitoring facility hardware. At powerup, microcode clears both of the 16-bit hardware incrementers and any potential interrupt request.

MICROCODE RESTRICTION

> If the performance monitoring facility hardware incrementers are cleared in cycle 'n' via MISC/CLR.PERF.COUNT, INT.SYS<28> must be written with a 1 no earlier than cycle 'n+3' to guarantee that the interrupt request is cleared. This delay is due to latency introduced between the performance monitoring factility hardware and the interrupt section.

Microcode reads the current value of the hardware incrementers via A/PERF.COUNT as a byproduct of a read of the PMFCNT processor register, and as part of the process of updating the memory counters.

Microcode clears the hardware incrementers via MISC/CLR.PERF.COUNT when ECR<PMF_CLEAR> is written with a 1. Microcode also clears the incrementers after reading and updating the memory counters.

Microcode uses the CPUID processor register value to find the pair of quadwords that contain the performance counter values for this CPU. This value must be correctly initialized by either console firmware or software before the performance monitoring facility is enabled. The operation of the processor is UNDEFINED if CPUID is not correctly initialized.

The memory counters are updated under three circumstances: when a performance monitoring facility interrupt is serviced, when the facility is disabled via a write to the PME processor register, and when the facility is disabled by loading a new value of PME is LDPCTX. The memory updates are done in a common subroutine by disabling the facility by clearing ECR<PMF_ENABLE>, reading the current value of the hardware incrementers and then clearing them, and updating each quadword in memory with the appropriate 16-bit hardware value.

18.4 Revision History

Table 18–7: Revision History

Who	When	Description of change
Mike Uhler	12-Sep-1990	Reverse the definition of the TB selections for the Mbox performance monitoring mux
Mike Uhler	12-Jan-1990	Initial release
Mike Uhler	02-Jul-1990	Update to reflect implementation
Mike Uhler	13-Feb-1991	Update to relect pass 1 design
Mike Uhler	12-Aug-1991	Minor updates to clarify interrupt request

Chapter 19

Testability Micro-Architecture

19.1 Chapter Overview

This chapter describes the NVAX CPU chip's Testability Micro-Architecture—a framework of testability features implemented throughout the NVAX CPU chip.

The chapter does not detail the motivation for testability features or discuss the actual method of their uses in various life cycle testing phases. These is covered elsewhere. (For example, see in [1].)

19.2 The Testability Strategy

The NVAX CPU chip's testability strategy addresses the broad issue of providing cost-effective and thorough testing during many life cycle testing phases. The strategy specifically implements test features to support

- chip debug
- high fault coverage test at wafer probe and packaged chip test
- support "reduced probe contact" wafer probe test
- support for effective chip burn-in test
- support module interconnection test via boundary scan and in-circuit-test (ICT) via a single pin tristate feature.

The strategy uses a combination of a variety of testability techniques and approaches that are best suited to address the specific functional elements in the chip. The cost-effective implementation is realized by the appropriate consideration of global issues, by unifying the test objectives, by sharing test resources and by exploiting features inherent in the chip. The strategy also relies on leveraging off the design verification patterns in developing production test patterns to meet the fault coverage goals.

The test features are implemented such that they have no effect on the targeted performance of the chip.

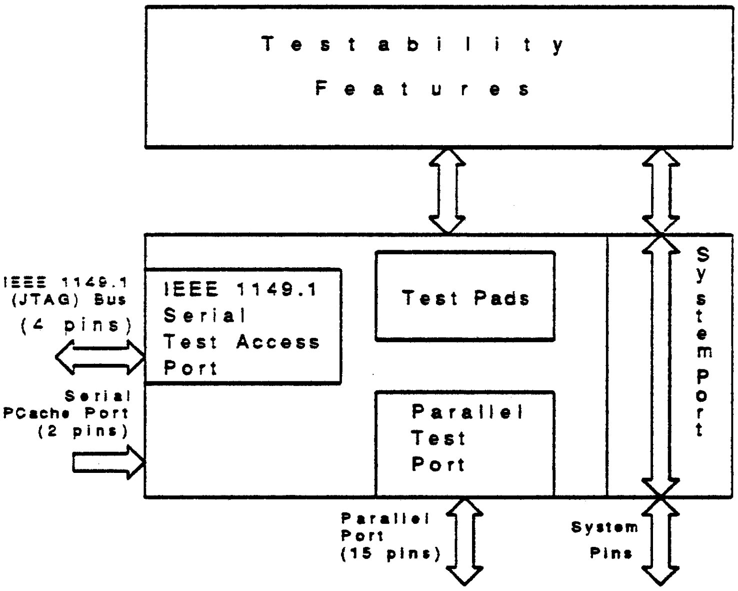
19.3 Test Micro-Architecture Overview

The NVAX CPU chip's Test Micro-Architecture consists of two principal elements: Test Interface Unit and the Testability Features.

Test Interface Unit

The Test Interface Unit (TIU) implements a comprehensive test access strategy for the NVAX CPU. It permits an efficient access to testability features implemented on the chip.

Figure 19–1: Test Interface Unit


Testability
Features
IEEE 1149.1 (JTAG) Bus (4 pins)
IEEE 1149.1 Serial Test Access Port
Test Pads
Serial PCache Port (2 pins)
Parallel Test Port
System Port
Parallel Port (15 pins)
System Pins

TIU shown in Figure 19–1 consists of three ports: an IEEE P1149.1 (JTAG) serial test port, a parallel test port and an "invisible" port consisting of test pads. The serial test port is a 4-pin dedicated test access port conforming to the IEEE P1149.1 (JTAG) standard. It is used for accessing the boundary scan register.

The parallel test port consists of 15 dedicated pins. This port is used for accessing internal scan registers and test features which benefit from parallel access (for example, microaddress bus).

The Test Pads primarily facilitates micro-probing during chip debug. These pads are located at strategic nodes throughout the chip.

The NVAX CPU also has a special 2-pin serial port consisting of **TEST_DATA_H** and **TEST_STROBE_H** that allow the PCache to be loaded serially under control from special microcode. This feature has been provided to support convenient self-test operation during the chip burn-in test. For more details see Section 19.7

In addition to these test ports, NVAX also uses the normal system port (pins) for test access. This access consists of using the VAX instructions to manipulate a testability feature or to perform the actual tests on the chip's logic.

Table 19–1 summarizes the dedicated test pins for NVAX.

Table 19–1: NVAX CPU's Test Pins

Pin Name	Pin Type	Pin Function
TDI_H	Input, Pull-up	IEEE 1149.1 Serial Test Data Input
TDO_H	Output, Tri-state, 2 receivers	IEEE P1149.1 Serial Test Data Output
TMS_H	Input, Pull-up	IEEE 1149.1 Test Mode Select
TCK_H	Input, pull-down	IEEE 1149.1 Test Clock
PP_CMD_H<2:0>	Input, pull-up	Parallel Port: Command Pins
PP_DATA_H<11:0>	Output	Parallel Port: Data Pins
DISABLE_OUT_L	Input, Pull-up	Disables (tristate) all output drivers
TEST_DATA_H	Input, Pull-up	Data for serially loading PCache
TEST_STROBE_H	Input, Pull-up	Strobe for serially loading PCache
OSC_TEST_H	Input	Test clock enable. See Section 3.2.2
OSC_TC1_H. OSC_TC2_H	Input	Test clocks. See Section 3.2.2
TEMP_H	Output	Temperature sensor. See Section 3.2.5

Testability Features

The testability features facilitate the testing of the chip, module, or system. The testability features are scattered throughout the NVAX CPU chip. The features implemented primarily use internal scan registers, LFSR Reducers and boundary scan register.

19.4 Parallel Test Port

This port allows the critical chip nodes to be either controlled or monitored in parallel. The port consists of 15 dedicated test pins as follows:

- **PP_DATA_H<11:0>**: A 12 bit output pins that provide control to or observability of various internal nodes.
- **PP_CMD_H<2:0>**: Selects up to eight different test configurations at the parallel port.

Table 19–2 lists the Parallel Port's configurations.

NOTE

1. When the parallel port is not in use, internal pull-ups on **PP_CMD_H<2:0>** pins force the port into an inactive (Ebox observe MAB) state.

2. PP_CMD_H< 0 > pin is also used as pseudo-TRST_L pin to reset JTAG circuits.

Table 19–2: Parallel Port Operating Modes

Command Pins		Data Pins	
PP_CMD_H<2:0>	**Port Mode**	**PP_DATA_H<11:0>**	**Signals controlled/Observed**
1 1 1	Observe MAB (Default)	PP_DATA_H<11>	Internal PHI_2.
		PP_DATA_H<10:0>	Ebox MAB. See Section 9.5.
1 1 0	Observe Mbox	PP_DATA_H<11:9>	S5 Reference Source. See Section
		PP_DATA_H<8:4>	S5 command. See Table 12–1.
		PP_DATA_H<3>	M%MME_FAULT_H.
		PP_DATA_H<2>	S5 Abort.
		PP_DATA_H<1>	S5 TB Miss.
		PP_DATA_H<0>	S5 PCache Hit.
1 0 1	Observe Cbox/Mbox	PP_DATA_H<11:9>	Cbox BC_TS_CMD<2:0>. See Table 13–
		PP_DATA_H<8>	Cbox DEALLOC.
		PP_DATA_H<7>	Cbox BC_HIT.
		PP_DATA_H<6:4>	Mbox MD Destination. See Sectio
		PP_DATA_H<3:0>	Mbox MME State. See Section 12.
1 0 0	Observe Ibox	PP_DATA_H<11>	Internal PHI_2.
		PP_DATA_H<10:7>	Undefined.
		PP_DATA_H<6:0>	I-MAB. See Section 7.11.3.
0 1 1	Enable LFSR Mode	PP_DATA_H<11:0>	Undefined.
0 1 0	Undefined	PP_DATA_H<11:0>	Undefined.
0 0 1	Shift ISRs	PP_DATA_H<11:3>	ISR1 (Control Store data).
		PP_DATA_H<2:0>	ISR2 (Other internal scan data).
0 0 0	Force MAB	PP_DATA_H<11:0>	Undefined. See Section 9.5.

19.4.1 Parallel Port Operation

Internal Scan Registers

When shifting, the ISR bits are serial to parallel converted. They change every third cycle on internal PHI_4. This gives usable time with respect to the NDAL clocks. The parallel port commands are captured synchronously with respect to the NDAL clocks, in NDAL phase 3. In order to give full flexibility in capturing a given internal cycle, a mechanism is provided to delay the capture-and-start-shifting event by 0, 1, or 2 cycles. This delay is determined by the state of the parallel port bits PP_CMD< 1:0 > immediately before entering the Shift ISR mode. ('00' corresponds to zero delay, '01' corresponds to 1 cycle delay and '10' correspond to two cycle delays.) See the timing diagrams in Figure 19–2

Chapter 20

Electrical Characteristics

20.1 Introduction

This chapter specifies the electrical characteristics to which one must adhere in order to incorporate the chip in a system. Related information may be obtained from the following documents:

1. NVAX Module Signal Integrity Handbook.
2. CMOS-4 Technology File, revision 2.3.
3. NVAX CPU Module Inter-chip Specification.
4. NVAX CPU Chip Functional Specification, Chapter 3, Chapter 13, and Chapter 17.

20.2 NVAX DC Operating Characteristics

20.2.1 Maximum Ratings

Table 20–1: Maximum Ratings

Parameter	sym	min	max	units	comments
internal supply voltage	VDDi	3.0	3.465	Vdc	3.3V +5%/-10% including power supply ripple
external supply voltage	VDDe	3.0	3.465	Vdc	3.3V +5%/-10% including power supply ripple
power dissipation @ 10ns cycle			16.3	watts	measured at VDDi=VDDe= 3.465V
power dissipation @ 12ns cycle			13.8	watts	measured at VDDi=VDDe= 3.465V
power dissipation @ 14ns cycle			12.0	watts	measured at VDDi=VDDe= 3.465V
power dissipation @ 18ns cycle			9.7	watts	measured at VDDi=VDDe= 3.465V
junction temperature	Tj	0	100	deg C	specific ambient temperature depends on board design and air flow

Table 20–2: Power Dissipation Across Voltage and Cycle Time

Cycle time	min@3.2V	max@3.2V	max@3.465V	max@3.6V	units
10ns cycle	8.3	13.9	16.3	17.6	watts
12ns cycle	7.1	11.8	13.8	14.9	watts
14ns cycle	6.2	10.3	12.0	13.0	watts
18ns cycle	5.0	8.3	9.7	10.4	watts

The power dissipation numbers given are worst-case average power dissipation measurements; they do not represent the peak instantaneous power dissipated on NVAX. The worst-case average power values were developed from the measured power dissipated when a worst-case pattern was run on an NVAX chip in a Neptune system.

20.2.2 Pin Driver Impedance

Table 20–3 contains the acceptable range for output driver impedance, assuming worst case environmental skews.

Table 20–3: NVAX Pin Driver Impedance

Name	Rterm low	Rterm high	Rds low	Rds high	Z low	Z high
P%ACK_L	10	15	19	37	29	52
P%CMD_H<3:0>	10	15	65	125	75	140
P%CPU_HOLD_L	12	18	20	41	32	59
P%CPU_REQ_L	12	18	20	41	32	59
P%CPU_SUPPRESS_L	12	18	20	41	32	59
P%DR_DATA_H<63:0>	11	17	20	41	31	58
P%DR_ECC_H<7:0>	11	17	20	41	31	58
P%DR_INDEX_H<20:3>	4	6	12	25	16	31
P%DR_OE_L	4	6	12	25	16	31
P%DR_WE_L	4	6	12	25	16	31
P%ID_H<2:0>	10	15	65	125	75	140
P%MACHINE_CHECK_H	10	15	65	125	75	140
P%NDAL_H<63:0>	10	15	65	125	75	140
P%PARITY_H<2:0>	10	15	65	125	75	140
P%PHI12_OUT_H	8	12	8	23	16	35
P%PHI23_OUT_H	8	12	8	23	16	35
P%PHI34_OUT_H	8	12	8	23	16	35
P%PHI41_OUT_H	8	12	8	23	16	35
P%PP_DATA_H<11:0>	10	15	65	125	75	140
P%SYS_RESET_L	12	18	20	41	32	59
P%TDO_H	10	15	65	125	75	140
P%TS_ECC_H<5:0>	11	17	20	41	31	58
P%TS_INDEX_H<20:5>	12	18	20	41	32	59
P%TS_OE_L	12	18	20	41	32	59
P%TS_OWNED_H	11	17	20	41	31	58
P%TS_TAG_H<31:17>	11	17	20	41	31	58
P%TS_WE_L	12	18	20	41	32	59

Key to pin characteristics:

Rterm — termination resistance
Rds — device resistance
Z — sum of resistance range

Conditions of test:

Vdd = 3.465v
Tj = 0 and 100 degrees Centigrade
Rds measured with the pin shorted to Vdd=3.465v for measuring N-MOS characteristics
Rds measured with the pin shorted to Vss=0.0v for measuring P-MOS characteristics
Pins cannot tolerate shorts for prolonged periods. The above information is provided for test purposes only.

20.2.3 Pin Capacitance

Table 20–4: Maximum Pin Capacitance

Pin Types	Rating	Unit
I/O and output only pins	12.0	pF
input only pins except for **P%PHIXX_IN_H**	7.5	pF
P%PHIXX_IN_H	8.5	pF

Conditions of test (in simulation):

measured as pin capacitance to VSSi with all other pins returned to VSSi
Tj = 27 degrees Centigrade
measured at DC, zero bias for the junction capacitors

20.2.4 Pin Operating Levels

Table 20–5 summarizes the electrical characteristics for various pin operating levels. Table 20–6 identifies the operating level associated with each unique pin group.

Table 20–5: NVAX Pin Levels

Level Type	Vil	Vih	Vol	Iol	Voh	Ioh	Max Vin[1]	Leakage
TTL IO[2]	0.8	2.0	0.4	+2mA	2.5	-2mA	6V	100uAmps
TTL IN	0.8	2.0					4.5V	100uAmps
TTL IN PU[3]	0.8	2.0					Vdd+0.5V	±200-900uAmps
P%PP_CMD_ H<2:0>	0.8	2.0					Vdd+0.5V	1000uAmps
CMOS[4]	0.8	2.0	0.4	+2mA	2.6	-2mA	4.5V	100uAmps
CMOS[5]	0.8	2.0	Vss+0.1V	+40uA	Vdd-0.1V	-40uA	4.5V	100uAmps
ECL IN	-0.3V	+0.3V					Vdd+0.5V	100uAmps
ACK IN[6]	0.8V	2.0	0.4	+17mA			Vdd+0.5V	100uAmps

[1] maximum voltage tolerable without incurring damage

[2] 5-volt tolerant

[3] pins with active pull-up or pull-down

[4] with TTL load

[5] with CMOS load

[6] active pull-up to 3.3 volts

.4

Table 20–6: NVAX Pin Characteristics

Name	Type	Level	Voltage	Pull-x
P%ACK_L	B,OD	ACK	3	
P%ASYNC_RESET_L	I	TTL	3	+
P%CMD_H<3:0>	B	TTL	5	
P%CPU_GRANT_L	I	TTL	3	
P%CPU_HOLD_L	O	TTL	5	
P%CPU_REQ_L	O	TTL	5	
P%CPU_SUPPRESS_L	O	TTL	5	
P%CPU_WB_ONLY_L	I	TTL	3	
P%DISABLE_OUT_L	I	TTL	3	+
P%DR_DATA_H<63:0>	B	TTL	5	-
P%DR_ECC_H<7:0>	B	TTL	5	-
P%DR_INDEX_H<20:3>	O	TTL	3	
P%DR_OE_L	O	TTL	3	
P%DR_WE_L	O	TTL	3	
P%HALT_L	I	TTL	3	
P%H_ERR_L	I	TTL	3	
P%ID_H<2:0>	B	TTL	5	
P%INT_TIM_L	I	TTL	3	
P%IRQ_L<3:0>	I	TTL	3	
P%MACHINE_CHECK_H	O	TTL	5	
P%NDAL_H<63:0>	B	TTL	5	
P%OSC_H	I	ECL	3	
P%OSC_L	I	ECL	3	
P%OSC_TC1_H	I	CMOS	3	
P%OSC_TC2_H	I	CMOS	3	
P%OSC_TEST_H	I	CMOS	3	-
P%PARITY_H<2:0>	B	TTL	5	
P%PHI12_IN_H	I	CMOS	3	
P%PHI12_OUT_H	O	CMOS	3	
P%PHI23_IN_H	I	CMOS	3	
P%PHI23_OUT_H	O	CMOS	3	
P%PHI34_IN_H	I	CMOS	3	
P%PHI34_OUT_H	O	CMOS	3	
P%PHI41_IN_H	I	CMOS	3	
P%PHI41_OUT_H	O	CMOS	3	
P%PP_CMD_H<2:0>	I	TTL	3	+
P%PP_DATA_H<11:0>	O	TTL	5	
P%PWRFL_L	I	TTL	3	
P%SYS_RESET_L	O	TTL	5	
P%S_ERR_L	I	TTL	3	

Key to pin characteristics:

LEVEL — threshold levels as per Table 20–5
VOLTAGE — (5) 5V tolerant driver, (3) 3V tolerant driver - must not be exposed to 5V signals
PULL-X — (+) active pull-up, (-) active pull-down
TYPE — (B) bidirectional, (I) input, (O) output, (OD) open drain

Table 20–6 (Cont.): NVAX Pin Characteristics

Name	Type	Level	Voltage	Pull-x
P%TCK_H	I	TTL	3	-
P%TDI_H	I	TTL	3	+
P%TDO_H	O	TTL	5	
P%TEMP_H	O	< 3V	3	
P%TEST_DATA_H	I	TTL	3	+
P%TEST_STROBE_H	I	TTL	3	+
P%TMS_H	I	TTL	3	+
P%TS_ECC_H<5:0>	B	TTL	5	-
P%TS_INDEX_H<20:5>	O	TTL	5	
P%TS_OE_L	O	TTL	5	
P%TS_OWNED_H	B	TTL	5	-
P%TS_TAG_H<31:17>	B	TTL	5	-
P%TS_VALID_H	B	TTL	5	-
P%TS_WE_L	O	TTL	5	

Key to pin characteristics:

LEVEL — threshold levels as per Table 20–5
VOLTAGE — (5) 5V tolerant driver, (3) 3V tolerant driver - must not be exposed to 5V signals
PULL-X — (+) active pull-up, (-) active pull-down
TYPE — (B) bidirectional, (I) input, (O) output, (OD) open drain

20.3 NVAX AC Operating Characteristics

This section specifies AC timing parameters, but is not intended to illustrate detailed transactional operation.

20.3.1 AC Conditions of Test

1. Tj = 0 to 70 degrees Centigrade
2. VDDi = 3.3 volts +3.9%/-3% (3.2 to 3.43V)
3. VDDe = 3.3 volts +3.9%/-3% (3.2 to 3.43V)
4. Voltage levels used for timing specifications as per Table 20–5.
5. Pin loading used for timing specifications as per Table 20–7.

Table 20–7: Pin Loading for AC Tests

		Loading required for chip test on Takeda 3381	
Pin	**Total Pin Loading**	**Series Resistor**	**Series Capacitor**
P%DR_INDEX_H<20:3>	140 pF	10 ohms	100 pF
P%DR_OE_L	140 pF	10 ohms	100 pF
P%DR_WE_L	140 pF	10 ohms	100 pF
P%TS_INDEX_H<20:5>	60 pF	15 ohms	20 pF
P%TS_OE_L	60 pF	15 ohms	20 pF
P%TS_WE_L	60 pF	15 ohms	20 pF
P%PHIXX_OUT_H	70 pF	22 ohms	30 pF
all others	40 pF	none	none

The AC conditions of test given were designed specifically with the Neptune and Omega systems in mind, in order to maximize chip yield. The AC conditions of test may be changed in the future depending upon chip yields and the needs of the system partners.

This page intentionally left blank.

20.3.2 NDAL Timing Specification

NDAL signal timing is specified as phase and constant offsets from the NDAL clock inputs. The chip operating frequency determines the phase time.

Figure 20–1: NDAL Pin Timing Relative to the NDAL CLOCKS

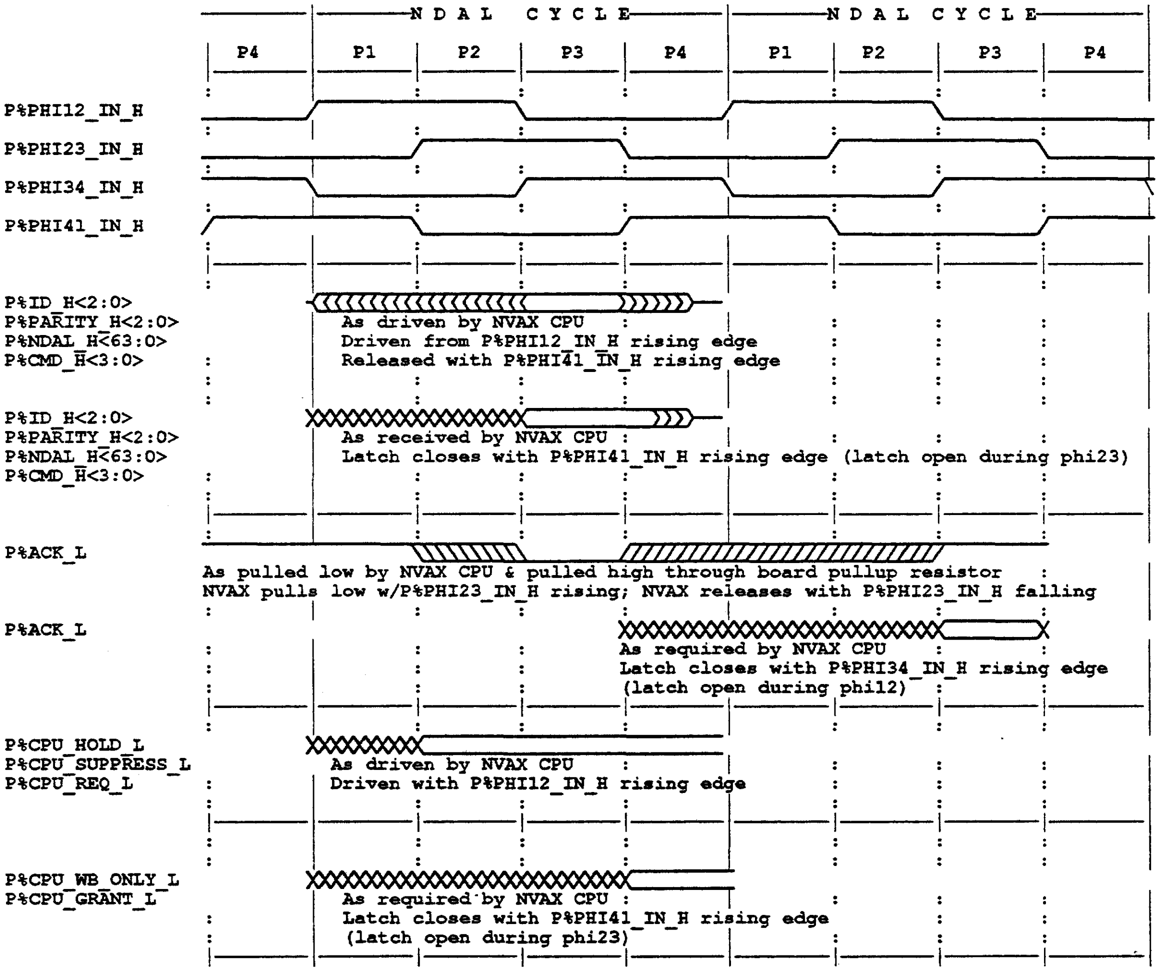

NDAL CYCLE
NDAL CYCLE
P4
P1
P2
P3
P4
P1
P2
P3
P4
P%PHI12_IN_H
P%PHI23_IN_H
P%PHI34_IN_H
P%PHI41_IN_H
P%ID_H<2:0>
P%PARITY_H<2:0>
P%NDAL_H<63:0>
P%CMD_H<3:0>
As driven by NVAX CPU
Driven from P%PHI12_IN_H rising edge
Released with P%PHI41_IN_H rising edge
P%ID_H<2:0>
P%PARITY_H<2:0>
P%NDAL_H<63:0>
P%CMD_H<3:0>
As received by NVAX CPU
Latch closes with P%PHI41_IN_H rising edge (latch open during phi23)
P%ACK_L
As pulled low by NVAX CPU & pulled high through board pullup resistor
NVAX pulls low w/P%PHI23_IN_H rising; NVAX releases with P%PHI23_IN_H falling
P%ACK_L
As required by NVAX CPU
Latch closes with P%PHI34_IN_H rising edge
(latch open during phi12)
P%CPU_HOLD_L
P%CPU_SUPPRESS_L
P%CPU_REQ_L
As driven by NVAX CPU
Driven with P%PHI12_IN_H rising edge
P%CPU_WB_ONLY_L
P%CPU_GRANT_L
As required by NVAX CPU
Latch closes with P%PHI41_IN_H rising edge
(latch open during phi23)

Table 20–8: NDAL AC timing specs

Input Pin	Setup Time[1]	Hold Time
P%NDAL_H<63:0>	1 phase to **P%PHI41_IN_H** R	**P%PHI41_IN_H** R + 2ns[2]
P%CMD_H<3:0>	"	"
P%ID_H<2:0>	"	"
P%PARITY_H<2:0>	"	"
P%ACK_L	0 ns to **P%PHI34_IN_H** R	**P%PHI34_IN_H** R + 1 phase
P%CPU_WB_ONLY_L	0 ns to **P%PHI41_IN_H** R	**P%PHI41_IN_H** R + 1 phase
P%CPU_GRANT_L	"	"

Output Pin	Output Valid	Output Tristate
P%NDAL_H<63:0>	**P%PHI12_IN_H** R + 2 phases	**P%PHI41_IN_H** R + 1 phase
P%CMD_H<3:0>	"	"
P%ID_H<2:0>	"	"
P%PARITY_H<2:0>	"	"
P%ACK_L	**P%PHI23_IN_H** R + 1 phase (low transition), **P%PHI23_IN_H** F + 3 phases(high transition)[3]	-
P%CPU_HOLD_L	**P%PHI12_IN_H** R + 1 phase	-
P%CPU_SUPPRESS_L	"	-
P%CPU_REQ_L	"	-

[1] R means the rising edge of the clock is used; F means the falling edge of the clock is used.

[2] Data may be held capacitively during the hold time.

[3] **P%ACK_L** is pulled up to 3.3v through a resistor in the system and in the test environment.

add HOLD times

20.3.3 BCACHE Timing Specification

Due to the chip's clocking structure, BCACHE timing is specified relatively between the various BCACHE signals.

Figure 20–2 and Figure 20–3 show the Bcache timing for a generic NVAX system. Table 20–9 and Table 20–10 specify the RAM timing constraints under *NVAX input requirements* subtitle, and the guaranteed chip output under *NVAX output responses*. This data should be used to establish chip input requirements and output responses in a generic system environment. Signal delays are dependent on chip packaging and board design.

For the OMEGA system, the chip must meet the input constraints and the output responses specified in Table 20–11 and Table 20–12. The OMEGA system operates at a 14ns clock cycle, using a 128 KB cache with 16Kx4 25ns data RAMs and 4Kx4 25ns tag RAMs. This configuration requires the Bcache processor register settings CCTL⟨DATA_SPEED⟩=01 and CCTL⟨TAG_SPEED⟩=1 to allow one slip cycle for both data and tag RAM access. See Chapter 13.

The specific timing for XNP systems is shown in Figure 20–4 and Figure 20–5. The chip must meet the input constraints and the output responses specified in Table 20–13 and Table 20–14. The XNP system operates at a 14, 12, or 10ns clock cycle, using a 2 MB cache with 256Kx4 20ns data RAMs and 64Kx4 15ns tag RAMs. This configuration requires the Bcache processor register settings CCTL⟨DATA_SPEED⟩=01 and CCTL⟨TAG_SPEED⟩=1 to allow one slip cycle for both data and tag RAM access.

The timing constraints for both the OMEGA and XNP systems are based upon the RAM specifications rather than upon NVAX predicted behavior. Actual signal delays are dependent on chip packaging and board design.

Figure 20–2: Generic Data RAM Timing Diagram

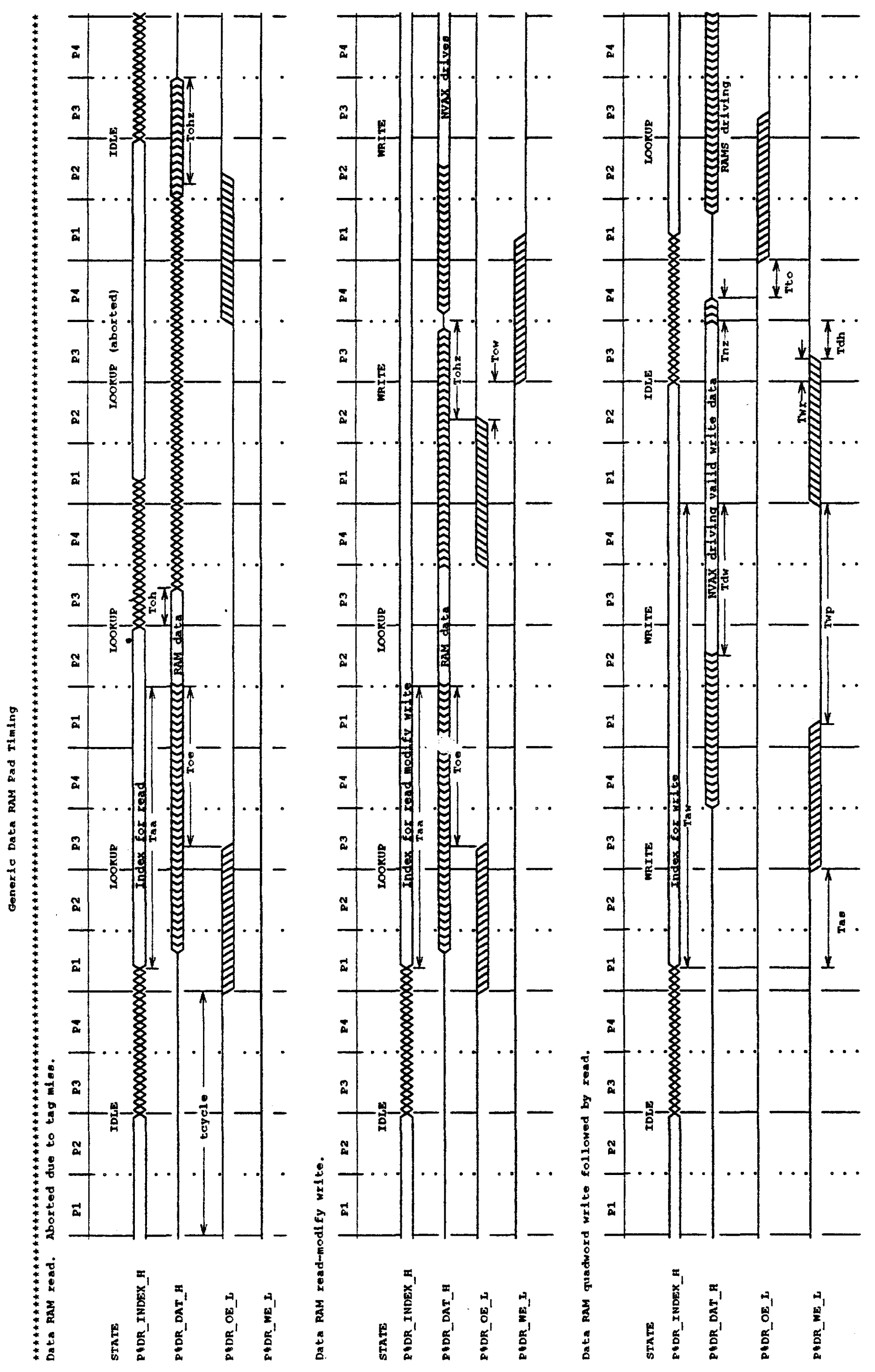


Table 20–9: Generic Data RAM Timing Specification

Param	Function	NVAX Input Requirement
Taa	address access to RAM data valid	≤ (7 phases - **P%DR_INDEX_H** drive-to-valid delay)
Toe	OE assertion to RAM data valid	≤ (5 phases - **P%DR_OE_L** deasserted-to-asserted delay)
Toh	RAM data output hold from INDEX change	≥ 2.0ns
Tohz	OE deassertion to RAM data high-z	≤ (4 phases - **P%DR_OE_L** asserted-to-deasserted delay)

Param	Function	NVAX Output Response
Tto	data high-z to OE assertion	≥ 0.0ns
Tdw	data valid to WE deassertion	≥ (5 phases - **P%DR_DAT_H** drive-to-valid delay)
Twp	WE pulse	≥ (6 phases - **P%DR_WE_L** deasserted-to-asserted delay)
Taw	address valid to end of write	≥ (10 phases - **P%DR_INDEX_H** drive-to-valid delay)
Tnz	NVAX tristate time	≤ 1 phase
Twr	write recovery (WE deassertion to INDEX change)	≥ 0.0ns
Tdh	data hold after WE deassertion	≥ 0.0ns
Tas	address setup	≥ 0.0ns
Tow	OE deassertion to WE assertion	≥ 0.0ns

Figure 20–3: Generic Tag RAM Timing Diagram

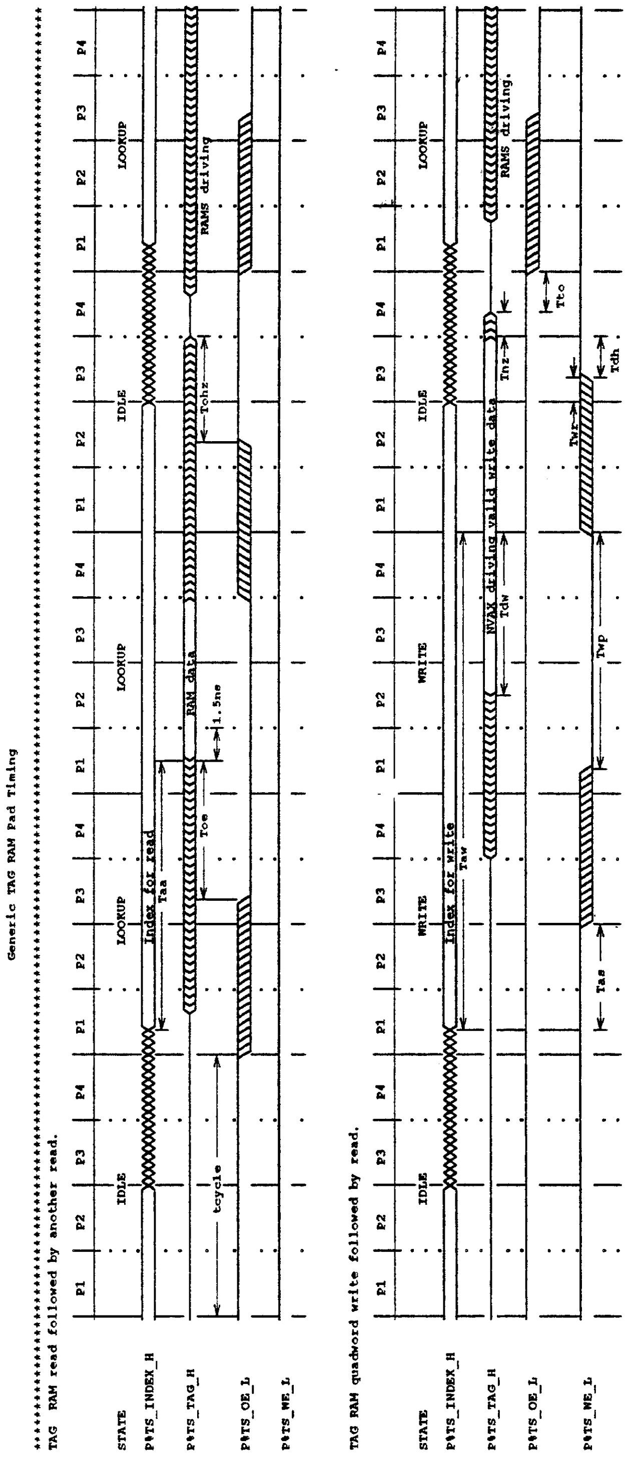


Table 20–10: Generic Tag RAM Timing Specification

Param	Function	NVAX Input Requirement
Taa	address access to RAM data valid	≤ (7 phases - **P%DR_INDEX_H** drive-to-valid delay - 1.5ns)
Toe	OE assertion to RAM data valid	≤ (5 phases - **P%DR_OE_L** drive-to-valid delay - 1.5ns)

Param	Function	NVAX Output Response
Tto	high-z to OE assertion	≥ 1.5ns
Tdw	data valid to WE deassertion	≥ (5 phases - **P%TS_TAG_H** drive-to-valid delay)
Twp	WE pulse	≥ (6 phases - **P%TS_WE_L** drive-to-valid delay)
Twr	write recovery	≥ -2.0ns
Tdh	data hold time	≥ 1.0ns
Tas	address setup	≥ (4 phases - **P%TS_INDEX_H** drive-to-valid delay)

Table 20–11: OMEGA-Specific Data RAM Timing Specification

128KB Bcache, 25ns Data RAMs, 25ns Tag RAMs, 14ns cycle

NVAX Test Input Requirements

Param	Function	Timing	Measuring Point	Notes
Taa	address access to data valid	≥ 25.0ns	INDEX 2.4H/.4L	must be met before tester drives data
Toe	OE assertion to data valid	≥ 12.0ns	OE .4L	must be met before tester drives data
Toh	output hold	≤ 3.0ns	INDEX .4H/2.4L	tester hold time
Tohz	OE deassertion to data high-z	≥ 10.0ns	OE 2.4H	tester hold time, chip overdrives
Tcycle	internal cycle time	14.0ns		

NVAX Test Output Responses

Param	Function	Timing	Measuring Point	Notes
Tto	high-z to OE assertion	≥ 0.0ns	OE 2.4L	
Tdw	data valid to WE deassertion	≥ 10.0ns	DAT 2.4H/.4L	
Twp	WE pulse	≥ 15.0ns	WE .4L, WE 2.4H	
Twr	write recovery	≥ 0.0ns	WE .4L, INDEX .4H/2.4L	
Tdh	data hold time	≥ 0.0ns	WE 2.4H	
Tas	address setup	≥ 0.0ns	INDEX 2.4H/.4L, WE 2.4L	
Tow	OE deassertion to WE assertion	≥ 0.0ns	OE 2.4H, WE .4L	

Table 20–12: OMEGA Specific Tag RAM Timing Specification

512KB Bcache, 12ns Data RAMs, 12ns Tag RAMs

NVAX Test Input Requirements

Param	Function	Timing	Measuring Point	Notes
Taa	address access to data valid	≥ 12.0ns	INDEX 2.4H/.4L	must be met before tester drives data
Toe	OE assertion to data valid	≥ 6.0ns	OE .4L	must be met before tester drives data
Tcycle	internal cycle time	14.0ns		

NVAX Test Output Responses

Param	Function	Timing	Measuring Point	Notes
Tto	high-z to OE assertion	≥ 0.0ns	OE 2.4L	
Tdw	data valid to WE deassertion	≥ 6.0ns	DAT 2.4H/.4L	
Twp	WE pulse	≥ 12.0ns	WE .4L, WE 2.4H	
Twr	write recovery	≥ 0.0ns	WE .4L, INDEX .4H/2.4L	
Tdh	data hold time	≥ 0.0ns	WE 2.4H	
Tas	address setup	≥ 0.0ns	INDEX 2.4H/.4L, WE 2.4L	

Figure 20–4: XNP Specific Data RAM Timing Diagram

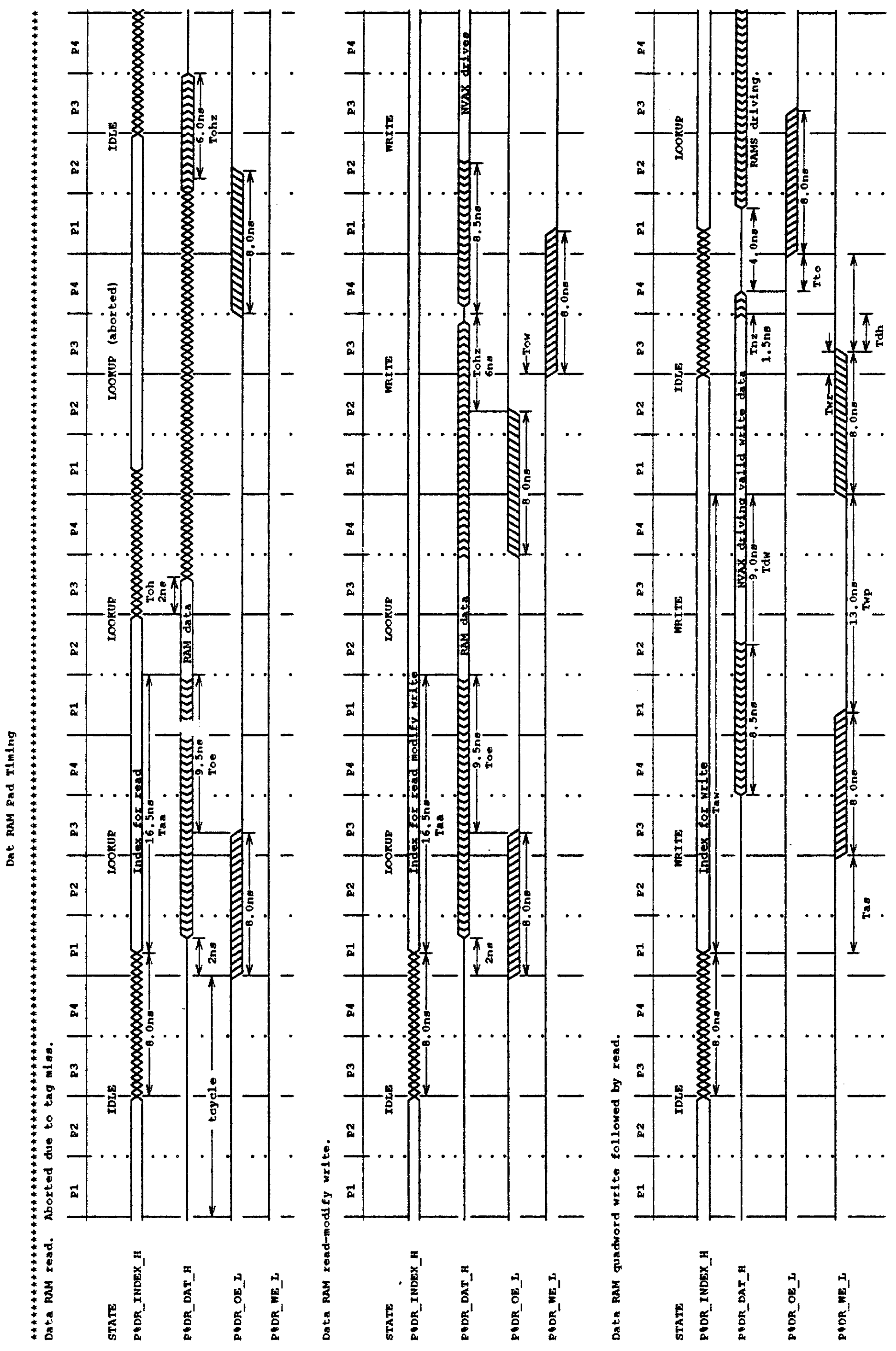


Table 20–13: XNP Specific Data RAM Timing Specification

2MB Bcache, 20ns Data RAMs, 15ns Tag RAMs

NVAX Test Input Requirements

Param	Function	Timing	Measuring Point	Notes
Taa	address access to data valid	≥ 20.0ns	INDEX 2.4H/.4L	must be met before tester drives data
Toe	OE assertion to data valid	≥ 10.0ns	OE .4L	must be met before tester drives data
Toh	output hold	≤ 5.0ns	INDEX .4H/2.4L	tester hold time
Tohz	OE deassertion to data high-z	≥ 10.0ns	OE 2.4H	tester hold time, chip overdrives
Tcycle	internal cycle time	14.0, 12.0, or 10.0ns		

NVAX Test Output Responses

Param	Function	Timing	Measuring Point	Notes
Tto	high-z to OE assertion	≥ 0.0ns	OE 2.4L	
Tdw	data valid to WE deassertion	≥ 12.0ns	DAT 2.4H/.4L	
Twp	WE pulse	≥ 14.0ns	WE .4L, WE 2.4H	
Twr	write recovery	≥ 0.0ns	WE .4L, INDEX .4H/2.4L	
Tdh	data hold time	≥ 0.0ns	WE 2.4H	
Tas	address setup	≥ 0.0ns	INDEX 2.4H/.4L, WE 2.4L	
Tow	OE deassertion to WE assertion	≥ 0.0ns	OE 2.4H, WE .4L	

Figure 20–5: XNP Specific Tag RAM Timing Diagram

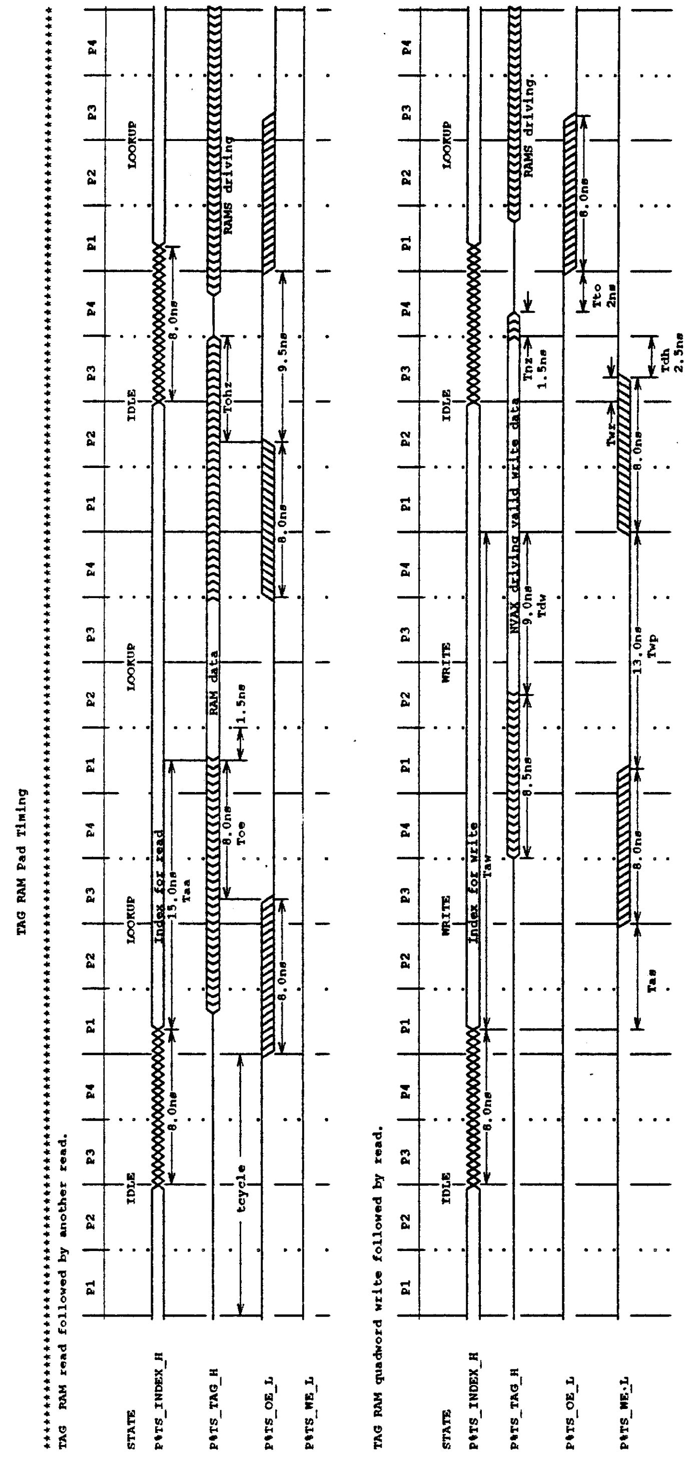


Table 20–14: XNP Specific Tag RAM Timing Specification

2MB Bcache, 20ns Data RAMs, 15ns Tag RAMs

NVAX Test Input Requirements

Param	Function	Timing	Measuring Point	Notes
Taa	address access to data valid	≥ 15.0ns	INDEX 2.4H/.4L	must be met before tester drives data
Toe	OE assertion to data valid	≥ 8.0ns	OE .4L	must be met before tester drives data
Tcycle	internal cycle time	14.0, 12.0, or 10.0ns		

NVAX Test Output Responses

Param	Function	Timing	Measuring Point	Notes
Tto	high-z to OE assertion	≥ 0.0ns	OE 2.4L	
Tdw	data valid to WE deassertion	≥ 7.0ns	TAG 2.4H/.4L	
Twp	WE pulse	≥ 15.0ns	WE .4L, WE 2.4H	
Twr	write recovery	≥ 0.0ns	WE .4L, INDEX .4H/2.4L	
Tdh	data hold time	≥ 0.0ns	WE 2.4H	
Tas	address setup	≥ 0.0ns	INDEX 2.4H/.4L, WE 2.4L	

20.3.4 Other Pin Timing Specifications

20.3.4.1 Clock Timing

When **P%OSC_TEST_H** is not asserted the chip receives the master clock through the **P%OSC_H** and **P%OSC_L** pins. Operation of the chip at the maximum internal clock speed of 100 MHz requires an input clock frequency of 400 MHz. These pins require capacitively-coupled, differential waveforms, 180 degrees out of phase at inverted ECL levels. The peak-to-peak differential voltage must be at least 600mV, with a differential symmetry of 60/40 or better. The voltage should not exceed the absolute value of Vdd plus 500 mV during operation.

When **P%OSC_TEST_H** is asserted the chip receives the master clock through the **P%OSC_TC1_H** and **P%OSC_TC2_H** pins. Operation of the chip at the maximum internal clock speed of 100 MHz requires an input clock frequency of 200MHz. These pins require waveforms that are 90 degrees out of phase at CMOS input levels (Table 20–5). Each edge must be place within an accuracy of ± 24 degrees.

The chip provides four double phase NDAL clocks on the **P%PHIXX_OUT_H** pins. The chip also receives these clocks through the **P%PHIXX_IN_H** pins. The relationship of the four clocks to the internal CPU clock cycle is shown in Figure 20–6.

Figure 20–6: Relationship of Internal and NDAL Clock Cycles

```
CPU CYCLE       | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 |
                    PHI1        PHI2        PHI3        PHI4
NDAL CYCLE      |-----------|-----------|-----------|-----------|

PHI12_OUT_H     /~~~~~~~~~~~~~~~~~~~~~~~\_______________________/

PHI23_OUT_H     ____________/~~~~~~~~~~~~~~~~~~~~~~~\____________

PHI34_OUT_H     \_______________________/~~~~~~~~~~~~~~~~~~~~~~~\

PHI41_OUT_H     ~~~~~~~~~~~~\_______________________/~~~~~~~~~~~~
```

The following skew specifications must be met for all NDAL clock receivers. Inter-clock skew is dependent on the electrical characteristics of the chip environment.

1. The rising edge of any clock will be present at all receivers within ± 0.5 ns, as measured from the CMOS Vih level (see Table 20–5).
2. The falling edge of any clock will be present at all receivers within ± 0.5 ns, as measured from the CMOS Vil level.
3. The skew between the rising edge of any phase and the falling edge of any other phase will be no more than ± 0.75 ns, as measured from Voh to Vol.
4. The NDAL clocks will have an edge rate of 2.0 ns or better, measured at the receiver, between the 10% and 90% points.

20.3.4.2 Reset Timing

P%ASYNC_RESET_L is an asynchronous input. It must be asserted for a minimum of 7 NDAL cycles. The **P%SYS_RESET_L** output is asserted asynchronously whenever **P%ASYNC_RESET_L** is asserted. **P%SYS_RESET_L** is deasserted synchronously with the rising edge of **P%PHI12_OUT_H**. Figure 20–7 shows the relationship between the reset signals and the clocks.

Figure 20–7: System Reset Timing

```
                                                              *************************************************
                                                              |  NDAL CYCLE   |  NDAL CYCLE   |  NDAL CYCLE   |
                                                              |               |               |               |
                                                              | P1| P2| P3| P4| P1| P2| P3| P4| P1| P2| P3| P4|
                                                              |---|---|---|---|---|---|---|---|---|---|---|---|
                                                              +   |   |   |   +   |   |   |   +   |   |   |   +
P%ASYNC_RESET_L ~~\\\_________________   ..  ..  ___________________/////~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                     <-- Asserted for a minimum of 7 NDAL Cycles ---> |   |   +   |   |   |   +   |   |   |   +
P%SYS_RESET_L   ~~\\\\\\\\\\__________   ..  ..  _____________________________________________//////////~~~~~~
                                                              +   |   |   |   +   |   |   |   +   |   |   |   +
                ^                               P%PHI12_OUT_H /~~~~~~~\_______/~~~~~~~\_______/~~~~~~~\_______/
                | P%ASYNC_RESET_L asynchronous                +   |   |   |   +   |   |   |   +   |   |   |   +
                | assertion causes asynchronous P%PHI23_OUT_H ____/~~~~~~~\_______/~~~~~~~\_______/~~~~~~~\_____
                | assertion of P%SYS_RESET_L.                 +   |   |   |   +   |   |   |   +   |   |   |   +
                                                P%PHI34_OUT_H \_______/~~~~~~~\_______/~~~~~~~\_______/~~~~~~~\
                                                              +   |   |   |   +   |   |   |   +   |   |   |   +
                                                P%PHI41_OUT_H ~~~~\_______/~~~~~~~\_______/~~~~~~~\_______/~~~~
                                                              +   |   |   |   +   |   |   |   +   |   |   |   +
                                                              *************************************************
```

The clock generator can be reset to a known state by using the **P%TEST_DATA_H** input as shown in Figure 20–8. With **P%AYSYNC_RESET_L** asserted, all clock inputs are stopped briefly (500 nS MAX). The states of test clocks **P%OSC_TC1_H** and **P%OSC_TC2_H** when stopped must be the same, either both high or both low. **P%TEST_DATA_H** should be driven low to effect the clock generator reset. This immediately places the clock generator into NVAX Φ_2 and NDAL Φ_1. **P%TEST_DATA_H** is then driven high and clocking of the chip is resumed. On the first oscillator cycle following resumption of clocking, the generator will transition into NVAX Φ_3 and begin normal sequencing. **P%AYSYNC_RESET_L** must remain asserted for at least 7 NDAL cycles following resumption of clocking.

Figure 20–8: Clock Generator Reset Timing

```
CPU Phase       XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  2   3   4   1   2   3   4   1
NDAL Phase      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  1  |    2           3
                                                                                        |
P%OSC_H         /~\_/~\_/~\_/~\____________________________________________________________________________________
                                                                                        |
P%OSC_L         \_/~\_/~\_/~\_/~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                                                                        |
P%OSC_TC1_H     __________________________/~~~\___/~~~\___/~~~\___/~~~\_________________/~~~\___/~~~\___/~~~\___/~
                                                                        |               |
P%OSC_TC2_H     ____________________________/~~~\___/~~~\___/~~~\___/~~~\_________________/~~~\___/~~~\___/~~~\___
                                                                        |               |
INTERNAl OSC    /~\_/~\_/~\_/~\___________/~\_/~\_/~\_/~\_/~\_/~\_/~\_/~\_______________/~\_/~\_/~\_/~\_/~\_/~\_/~
                               |          |                             |               |
P%ASYNC_RESET_L __________________________________________________________________________________________________
                               |          |                             |               |
P%SYS_RESET_L   __________________________________________________________________________________________________
                               |          |                             |               |
P%TEST_DATA_H   ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\_____/~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                               |          |                             |   |     |     |
NDAL Phase 1    ===============SSSSSSSSSSS==================================/////~~~~~~~~~~~\____________________/~
                               |          |                             |   |     |     |
                               |          |                            /    |     |     |
                               |          |                            Setup Assert Hold
                  Note 1          Note 2            Note 3             10 nS 10 nS  10 nS         Note 4
                                                                       min.  min.   min.

                  * CHIP POWER-UP *                                *  CLOCK RESET SEQUENCE *
```

- K_SEC%OSC1_H is the internal master clock produced from either the **P%OSC_H** and **P%OSC_L** inputs, or the **P%OSC_TC1_H** and **P%OSC_TC2_H** inputs. **P%OSC_TEST_H** is used to select the clock source as described in this clock specification.
- S indicates a static (non-changing) NDAL Φ_1.

Timing Notes:

1. ECL pin inputs **P%OSC_H** and **P%OSC_L** must be used to supply clocks to chip prior to and during power-up. Inputs **P%OSC_TC1_H** and **P%OSC_TC2_H** must be held low in order to prevent latch-up.
2. Switch to test clocks **P%OSC_TC2_H** and **P%OSC_TC2_H**. Start measure out 1pat on chip tester.
3. Clocks restarted to restore internal chip signals prior to clock-reset sequence.
4. **P%ASYNC_RESET_L** must remain asserted for a minimum of 7 NDAL cycles following restart of clocks.

20.3.4.3 Interrupt, Error, and Test Pin Timing

P%DISABLE_OUT_L and **P%TCK_H** are an asynchronous inputs.

P%TEMP_H is an asynchronous output.

When **P%PP_CMD_H<2:0>** selects Φ_2 on **P%PP_DATA_H<11>** (see Chapter 19) then the output is asynchronous.

The timing for the interrupt, parallel port, serial port, and boundary scan, and error pins is shown in Table 20–15.

Table 20–15: Interrupt, Test, and Boundary Scan Pin AC timing specs

Input Pin	Setup Time[1]	Hold Time
P%PWRFL_L	3.0 ns to P%PHI41_IN_H R	P%PHI41_IN_H R + 0.0 ns
P%HALT_L	3.0 ns to P%PHI41_IN_H R	P%PHI41_IN_H R + 0.0 ns
P%H_ERR_L	3.0 ns to P%PHI41_IN_H R	P%PHI41_IN_H R + 0.0 ns
P%INT_TIM_L	3.0 ns to P%PHI41_IN_H R	P%PHI41_IN_H R + 0.0 ns
P%S_ERR_L	3.0 ns to P%PHI41_IN_H R	P%PHI41_IN_H R + 0.0 ns
P%IRQ_L<3:0>	1 phase to P%PHI41_IN_H R	P%PHI41_IN_H R + 1 phase
P%TEST_DATA_H	1 phase to P%PHI41_IN_H R	P%PHI41_IN_H R + 1 phase
P%TEST_STROBE_H	1 phase to P%PHI41_IN_H R	P%PHI41_IN_H R + 1 phase
P%PP_CMD_H<2:0>	1 phase to P%PHI41_IN_H R	P%PHI41_IN_H R + 1 phase
P%TDI_H	3.0 ns to P%TCK	P%TCK F + 3.0 ns
P%TMS_H	3.0 ns to P%TCK	P%TCK F + 3.0 ns
Output Pin	**Output Valid**	
P%MACHINE_CHECK_H	P%PHI12_IN_H R + 1 phase	
P%PP_DATA_H<10:0>	P%PP_DATA_H<11> R + 3 phases	
P%TDO_H	P%TCK R + 10.0 ns	

[1] R means the rising edge of the clock is used; F means the falling edge of the clock is used.

20.4 Revision History

Table 20–16: Revision History

Who	When	Description of change
Rebecca Stamm	3-Dec-1991	Revision 1.2, correct power supply numbers, leakage numbers, AC conditions of test
Rebecca Stamm	9-Oct-1991	Revision 1.1, update power numbers and AC test conditions
John F. Brown	30-Aug-1991	Revision 1.0, first edition released
John F. Brown	20-Jun-1991	Revision 0.1, first edition for review
Mike Uhler	13-Feb-1991	Revision 0.0, add template

Appendix A

Processor Register Definitions

This appendix contains the SDL (Structure Definition Language) definitions for the NVAX processor registers. These definitions are used by chip verification code, and it is strongly recommended that software groups use the same definitions to minimize errors in the generating new definitions.

NOTE

> The file shown below is maintained by the NVAX CPU chip design group and is constantly being updated as changes are made to the design. It is included here simply as a means to document processor register definitions used in examples throughout this specification. The latest machine-readable version of this file should always be obtained from the NVAX CPU chip design group.

```
module $PR19DEF;

{*+
{*  Nvax - Specific Processor Register Definitions
{*
{*
{*  To convert this file to a macro library, do the following:
{*
{*   SDL/LANGUAGE=MACRO/COPYRIGHT/VMS_DEVELOPMENT/LIST PR19DEF
{*   LIBRARY/CREATE/MACRO/SQUEEZE PR19DEF PR19DEF
{*-

    constant REVISION equals 30 prefix PR19$;  /* Revision number of this file

/* In the definitions below, registers are annotated with one of the following
/* symbols:
/*
/*      RW - The register may be read and written
/*   RO - The register may only be read
/*   WO - The register may only be written
/*
/* For RO and WO registers, all bits and fields within the register are also
/* read-only or write-only.  For RW registers, each bit or field within
/* the register is annotated with one of the following:
/*
/*   RW - The bit/field may be read and written
/*   RO - The bit/field may be read; writes are ignored
/*   WO - The bit/field may be written; reads return an UNPREDICTABLE result.
/*   WZ - The bit/field may be written; reads return a 0
/*   WC - The bit/field may be read; writes cause state to clear
/*   RC - The bit/field may be read, which also causes state to clear; writes are ignored
```

```
aggregate PR19DEF union prefix PR19;

/* Architecturally-defined registers which have different characteristics
/* on this CPU.

constant TODR equals %x1B tag $;   /* Time Of Year Register (RW)

constant MCESR equals %x26 tag $;   /* Machine check error register (WO)

constant SAVPC equals %x2A tag $;   /* Console saved PC (RO)

constant SAVPSL equals %x2B tag $;   /* Console saved PSL (RO)
    PR19SAVPSL_BITS structure fill prefix SAVPSL$;
 PSL_LO bitfield length 8 mask;   /* Saved PSL bits <7:0>
 HALTCODE bitfield length 6 mask;  /* Halt code containing one of the following values
     constant HALT_HLTPIN equals %x2;  /* HALT_L pin asserted
     constant HALT_PWRUP equals %x3;  /* Initial powerup
     constant HALT_INTSTK equals %x4;  /* Interrupt stack not valid
     constant HALT_DOUBLE equals %x5;  /* Machine check during exception processing
     constant HALT_HLTINS equals %x6;  /* Halt instruction in kernel mode
     constant HALT_ILLVEC equals %x7;  /* Illegal SCB vector (bits<1:0>=11)
     constant HALT_WCSVEC equals %x8;  /* WCS SCB vector (bits<1:0>=10)
     constant HALT_CHMFI equals %xA;  /* CHMx on interrupt stack
     constant HALT_IE0 equals %x10;  /* ACV/TNV during machine check processing
     constant HALT_IE1 equals %x11;  /* ACV/TNV during KSNV processing
     constant HALT_IE2 equals %x12;  /* Machine check during machine check processing
     constant HALT_IE3 equals %x13;  /* Machine check during KSNV processing
     constant HALT_IE_PSL_101 equals %x19; /* PSL<26:24>=101 during interrupt or exception
     constant HALT_IE_PSL_110 equals %x1A; /* PSL<26:24>=110 during interrupt or exception
     constant HALT_IE_PSL_111 equals %x1B; /* PSL<26:24>=111 during interrupt or exception
     constant HALT_REI_PSL_101 equals %x1D; /* PSL<26:24>=101 during REI
     constant HALT_REI_PSL_110 equals %x1E; /* PSL<26:24>=110 during REI
     constant HALT_REI_PSL_111 equals %x1F; /* PSL<26:24>=111 during REI
 INVALID bitfield length 1 mask;   /* Invalid SAVPSL if = 1
 MAPEN bitfield length 1 mask;   /* MAPEN<0>
 PSL_HI bitfield length 16 mask;   /* Saved PSL bits <31:16>
    end PR19SAVPSL_BITS;

constant IORESET equals %x37 tag $;   /* I/O system reset register (WO)

constant PME equals %x3D tag $;    /* Performance monitoring enable (RW)

constant SID equals %x3E tag $;    /* System identification register (RO)
    PR19SID_BITS structure fill prefix SID$;
     UCODE_REV bitfield length 8 mask;  /* Microcode (chip) revision number
 NONSTANDARD_PATCH bitfield length 1 mask; /* PCS loaded with a non-standard patch
 PATCH_REV bitfield length 5 mask;  /* Patch revision number
 FILL_1 bitfield length 10 fill tag $$;
 TYPE bitfield length 8 mask;   /* CPU type code (19 decimal for NVAX)
    end PR19SID_BITS;
```

```
/* System-level required registers.
/* These registers are for testability and diagnostics use only.
/* They should not be referenced in normal operation.

constant IAK14 equals %x40 tag $;   /* Level 14 interrupt acknowledge (RO)

constant IAK15 equals %x41 tag $;   /* Level 15 interrupt acknowledge (RO)

constant IAK16 equals %x42 tag $;   /* Level 16 interrupt acknowledge (RO)

constant IAK17 equals %x43 tag $;   /* Level 17 interrupt acknowledge (RO)

    PR19IAK_VECTOR structure fill prefix IAK$;  /* Vector returned in response to IAK1x read
 IPL17 bitfield length 1 mask;   /* Force IPL 17, independent of actual level
 PR bitfield length 1 mask;   /* Passive release
 SCB_OFFSET bitfield length 14 mask;  /* LW offset in SCB of interrupt vector
 FILL_1 bitfield length 16 fill tag $$;
    end PR19IAK_VECTOR;

constant CWB equals %x44 tag $;    /* Clear write buffers (RW)
```

```
/* Ebox registers.

/* Ebox register definition

constant INTSYS equals %x7A tag $;   /* Interrupt system status register (RW)
    PR19INTSYS_BITS structure fill prefix INTSYS$;
     ICCS6 bitfield length 1 mask;   /* ICCS<6> (RO)
 SISR bitfield length 15 mask;   /* SISR<15:1> (RO)
 INT_ID bitfield length 5 mask;   /* ID of highest pending interrupt (RO)
     constant INT_ID_HALT equals %x1F;  /* Halt pin
     constant INT_ID_PWRFL equals %x1E;  /* Power fail
     constant INT_ID_H_ERR equals %x1D;  /* Hard error
     constant INT_ID_INT_TIM equals %x1C; /* Interval timer
     constant INT_ID_PMON equals %x1B;  /* Performance monitor
     constant INT_ID_S_ERR equals %x1A;  /* Soft error
     constant INT_ID_IRQ3 equals %x17;  /* IPL 17 device interrupt
     constant INT_ID_IRQ2 equals %x16;  /* IPL 16 device interrupt
     constant INT_ID_IRQ1 equals %x15;  /* IPL 15 device interrupt
     constant INT_ID_IRQ0 equals %x14;  /* IPL 14 device interrupt
     constant INT_ID_SISR15 equals %x0F;  /* SISR<15>
     constant INT_ID_SISR14 equals %x0E;  /* SISR<14>
     constant INT_ID_SISR13 equals %x0D;  /* SISR<13>
     constant INT_ID_SISR12 equals %x0C;  /* SISR<12>
     constant INT_ID_SISR11 equals %x0B;  /* SISR<11>
     constant INT_ID_SISR10 equals %x0A;  /* SISR<10>
     constant INT_ID_SISR9 equals %x09;  /* SISR<9>
     constant INT_ID_SISR8 equals %x08;  /* SISR<8>
     constant INT_ID_SISR7 equals %x07;  /* SISR<7>
     constant INT_ID_SISR6 equals %x06;  /* SISR<6>
     constant INT_ID_SISR5 equals %x05;  /* SISR<5>
     constant INT_ID_SISR4 equals %x04;  /* SISR<4>
     constant INT_ID_SISR3 equals %x03;  /* SISR<3>
     constant INT_ID_SISR2 equals %x02;  /* SISR<2>
     constant INT_ID_SISR1 equals %x01;  /* SISR<1>
     constant INT_ID_NO_INT equals %x00;  /* No interrupt
 FILL_1 bitfield length 3 fill tag $$;
 INT_TIM_RESET bitfield length 1 mask;  /* Interval timer interrupt reset (WC)
 FILL_2 bitfield length 2 fill tag $$;
 S_ERR_RESET bitfield length 1 mask;  /* Soft error interrupt reset (WC)
 PMON_RESET bitfield length 1 mask;  /* Performance monitoring interrupt reset (WC)
 H_ERR_RESET bitfield length 1 mask;  /* Hard error interrupt reset (WC)
 PWRFL_RESET bitfield length 1 mask;  /* Power fail interrupt reset (WC)
 HALT_RESET bitfield length 1 mask;  /* Halt pin interrupt reset (WC)
    end PR19INTSYS_BITS;
```

```
/* Ebox registers, continued.

constant PMFCNT equals %x7B tag $;   /* Performance monitoring facility count register (RW)
    PR19PMFCNT_BITS structure fill prefix PMFCNT$;
     PMCTR0 bitfield length 16 mask;   /* PMCTR0 word
     PMCTR1 bitfield length 16 mask;   /* PMCTR1 word
    end PR19PMFCNT_BITS;

constant PCSCR equals %x7C tag $;   /* Patchable control store control register (RW)
    PR19PCSCR_BITS structure fill prefix PCSCR$;
     FILL_1 bitfield length 8 fill tag $$;
 PAR_PORT_DIS bitfield length 1 mask;  /* Disable parallel port control of scan chain (RW)
 PCS_ENB bitfield length 1 mask;   /* Enable use of patchable control store (RW)
 PCS_WRITE bitfield length 1 mask;  /* Write scan chain to patchable control store (WO)
 RWL_SHIFT bitfield length 1 mask;  /* Shift read-write latch scan chain by one bit (WO)
 DATA bitfield length 1 mask;   /* Data to be shifted into the PCS scan chain (RW)
 FILL_2 bitfield length 10 fill tag $$;
 NONSTANDARD_PATCH bitfield length 1 mask; /* PCS loaded with a non-standard patch (RW)
 PATCH_REV bitfield length 5 mask;  /* Patch revision number (RW)
 FILL_3 bitfield length 3 fill tag $$;
    end PR19PCSCR_BITS;

constant ECR equals %x7D tag $;    /* Ebox control register (RW)
    PR19ECR_BITS structure fill prefix ECR$;
 VECTOR_PRESENT bitfield length 1 mask;  /* Vector unit present (RW)
 FBOX_ENABLE bitfield length 1 mask;  /* Fbox enabled (RW)
 TIMEOUT_EXT bitfield length 1 mask;  /* Select external timebase for S3 stall timeout timer (RW)
 FBOX_ST4_BYPASS_ENABLE bitfield length 1 mask; /* Fbox stage 4 conditional bypass enable (RW)
 TIMEOUT_OCCURRED bitfield length 1 mask; /* S3 stall timeout occurred (WC)
 TIMEOUT_TEST bitfield length 1 mask;  /* Select test mode for S3 stall timeout (RW)
 TIMEOUT_CLOCK bitfield length 1 mask;  /* Clock S3 timeout (RW)
 ICCS_EXT bitfield length 1 mask;  /* Full ICCS implemented in external logic (RW)
 FILL_1 bitfield length 5 fill tag $$;
 FBOX_TEST_ENABLE bitfield length 1 mask; /* Enable test of Fbox (RW)
 FILL_2 bitfield length 2 fill tag $$;
 PMF_ENABLE bitfield length 1 mask;  /* Performance monitoring facility enable (RW)
 PMF_PMUX bitfield length 2 mask;  /* Performance monitoring facility master select (RW)
     constant PMUX_IBOX equals %b00;  /* Select Ibox
     constant PMUX_EBOX equals %b01;  /* Select Ebox
     constant PMUX_MBOX equals %b10;  /* Select Mbox
     constant PMUX_CBOX equals %b11;  /* Select Cbox
 PMF_EMUX bitfield length 3 mask;  /* Performance monitoring facility Ebox mux select (RW)
     constant EMUX_S3_STALL equals %b000; /* Measure S3 stall against total cycles
     constant EMUX_EM_PA_STALL equals %b001; /* Measure EM+PA queue stall against total cycles
     constant EMUX_CPI equals %b010;  /* Measure instructions retired against total cycles
     constant EMUX_STALL equals %b011;  /* Measure total stalls against total cycles
     constant EMUX_S3_STALL_PCT equals %b100; /* Measure S3 stall against total stalls
     constant EMUX_EM_PA_STALL_PCT equals %b101; /* Measure EM+PA queue stall against total stalls
     constant EMUX_UWORD equals %b111;  /* Count microword increments
 PMF_LFSR bitfield length 1 mask;  /* Performance monitoring facility Wbus LFSR enable (RW)
 FILL_3 bitfield length 8 fill tag $$;
 PMF_CLEAR bitfield length 1 mask;  /* Clear performance monitoring hardware counters (WO)
    end PR19ECR_BITS;
```

```
/* Mbox TB registers.
/* These registers are for testability and diagnostics use only.
/* They should not be referenced in normal operation.

constant MTBTAG equals %x7E tag $;   /* Mbox TB tag fill (WO)
    PR19MTBTAG_BITS structure fill prefix MTBTAG$;
     TP bitfield length 1 mask;   /* Tag parity bit
 FILL_1 bitfield length 8 fill tag $$;
 VPN bitfield length 23 mask;   /* Virtual page number of address (VA<31:9>)
    end PR19MTBTAG_BITS;

constant MTBPTE equals %x7F tag $;   /* Mbox TB PTE fill (WO)
    PR19MTBPTE_BITS structure fill prefix MTBPTE$; /* Format is normal PTE format, except for PTE parity bit
     PFN bitfield length 23 mask;   /* Page frame number (PA<31:9>)
 FILL_1 bitfield length 1 fill tag $$;
 P bitfield length 1 mask;   /* PTE parity
 FILL_2 bitfield length 1 fill tag $$;
 M bitfield length 1 mask;   /* Modify bit
 PROT bitfield length 2 mask;   /* Protection field
 V bitfield length 1 mask;   /* PTE valid bit
    end PR19MTBPTE_BITS;
```

```
/* Vector architecture registers

constant VPSR equals %x90 tag $;   /* Vector processor status register (RW)
    PR19VPSR_BITS structure fill prefix VPSR$;
     VEN bitfield length 1 mask;   /* Vector processor enabled (RW)
 RST bitfield length 1 mask;   /* Vector processor state reset (WO)
 FILL_1 bitfield length 5 fill tag $$;
 AEX bitfield length 1 mask;   /* Vector arithmetic exception (WC)
 FILL_2 bitfield length 16 fill tag $$;
 IMP bitfield length 1 mask;   /* Implementation-specific hardware error (WC)
 FILL_3 bitfield length 6 fill tag $$;
 BSY bitfield length 1 mask;   /* Vector processor busy (RO)
    end PR19VPSR_BITS;

constant VAER equals %x91 tag $;   /* Vector arithmetic exception register (RO)
    PR19VAER_BITS structure fill prefix VAER$;
     F_UNDF bitfield length 1 mask;   /* Floating underflow
 F_DIVZ bitfield length 1 mask;   /* Floating divide-by-zero
 F_ROPR bitfield length 1 mask;   /* Floating reserved operand
 F_OVFL bitfield length 1 mask;   /* Floating overflow
 FILL_1 bitfield length 1 fill tag $$;
 I_OVFL bitfield length 1 mask;   /* Integer overflow
 FILL_2 bitfield length 10 fill tag $$;
 REGISTER_MASK bitfield length 16 mask;  /* Vector destination register mask
    end PR19VAER_BITS;

constant VMAC equals %x92 tag $;   /* Vector memory activity register (RO)

constant VTBIA equals %x93 tag $;   /* Vector translation buffer invalidate all (WO)
```

```
/* Cbox registers.

constant CCTL equals %xA0 tag $;   /* Cbox control register (RW)
    PR19CCTL_BITS structure fill prefix CCTL$;
     ENABLE bitfield length 1 mask;   /* Enable Bcache (RW)
 TAG_SPEED bitfield length 1 mask;  /* Tag RAM speed (RW)
     constant TAG_3_CYCLES equals 0;  /* Select tag RAM speed: 3-cycle read rep/3-cycle write rep
     constant TAG_4_CYCLES equals 1;  /* Select tag RAM speed: 4-cycle read rep/4-cycle write rep
 DATA_SPEED bitfield length 2 mask;  /* Data RAM speed (RW)
     constant DATA_2_CYCLES equals 0;  /* Select data RAM speed: 2-cycle read rep/3-cycle write rep
     constant DATA_3_CYCLES equals 1;  /* Select data RAM speed: 3-cycle read rep/4-cycle write rep
     constant DATA_4_CYCLES equals 2;  /* Select data RAM speed: 4-cycle read rep/5-cycle write rep
 SIZE bitfield length 2 mask;   /* Bcache size (RW)
     constant SIZE_128KB equals 0;  /* Select 128KB Bcache
     constant SIZE_256KB equals 1;  /* Select 256KB Bcache
     constant SIZE_512KB equals 2;  /* Select 512KB Bcache
     constant SIZE_2MB equals 3;   /* Select 2MB Bcache
 FORCE_HIT bitfield length 1 mask;  /* Force Bcache hit (RW)
 DISABLE_ERRORS bitfield length 1 mask;  /* Disable Bcache ECC errors (RW)
 SW_ECC bitfield length 1 mask;   /* Enable use of software ECC (RW)
 TIMEOUT_TEST bitfield length 1 mask;  /* Enable test of Cbox read timeout counters (RW)
 DISABLE_PACK bitfield length 1 mask;  /* Disable write packing (RW)
 PM_ACCESS_TYPE bitfield length 3 mask;  /* Performance monitoring access type (RW)
     constant PMAT_COH equals 0;   /* Coherency access of either type
     constant PMAT_COH_READ equals 1;  /* Coherency access for READ
     constant PMAT_COH_OREAD equals 2;  /* Coherency access for OREAD
     constant PMAT_CPU equals 4;   /* CPU access of any type
     constant PMAT_CPU_IREAD equals 5;  /* CPU access for IREAD
     constant PMAT_CPU_DREAD equals 6;  /* CPU access for DREAD
     constant PMAT_CPU_OREAD equals 7;  /* CPU access for OREAD
 PM_HIT_TYPE bitfield length 2 mask;  /* Performance monitoring hit type (RW)
     constant PMHT_HIT equals 0;   /* Hit
     constant PMHT_HIT_OWNED equals 1;  /* Hit on owned block
     constant PMHT_HIT_VALID equals 2;  /* Hit on valid block
     constant PMHT_MISS_OWNED equals 3;  /* Miss on owned block (causes writeback)
 FORCE_NDAL_PERR bitfield length 1 mask;  /* Forces 1 parity error on the NDAL, on next outgoing transaction
 FILL_1 bitfield length 13 fill tag $$;
 SW_ETM bitfield length 1 mask;   /* Enter software error transition mode (RW)
 HW_ETM bitfield length 1 mask;   /* Error transition mode entered due to error (WC)
    end PR19CCTL_BITS;

constant BCDECC equals %xA2 tag $;   /* Bcache data ram ECC (WO)
    PR19BCDECC_BITS structure fill prefix BCDECC$;
     FILL_1 bitfield length 6 fill tag $$;
 ECCLO bitfield length 4 mask;   /* ECC check bits <3:0>
     FILL_2 bitfield length 12 fill tag $$;
 ECCHI bitfield length 4 mask;   /* ECC check bits <7:4>
 FILL_3 bitfield length 6 fill tag $$;
    end PR19BCDECC_BITS;
```

```
/* Cbox registers, continued

constant BCETSTS equals %xA3 tag $;   /* Bcache error tag status (RW)
    PR19BCETSTS_BITS structure fill prefix BCETSTS$;
 LOCK bitfield length 1 mask;   /* Tag store registers are locked due to an error (WC)
 CORR bitfield length 1 mask;   /* Correctable error occurred (WC)
 UNCORR bitfield length 1 mask;       /* Uncorrectable error occurred (WC)
 BAD_ADDR bitfield length 1 mask;      /* Addressing error occurred (WC)
 LOST_ERR bitfield length 1 mask;      /* Error occured while register was locked (WC)
 TS_CMD bitfield length 5 mask;       /* Tag store command which caused error (RO)
     constant CMD_DREAD equals %b00111;  /* Command was D-stream tag lookup
     constant CMD_IREAD equals %b00011;  /* Command was I-stream tag lookup
     constant CMD_OREAD equals %b00010;         /* Command was OREAD tag lookup for write or read lock
     constant CMD_WUNLOCK equals %b01000; /* Command was write unlock tag lookup (done only under ETM)
     constant CMD_R_INVAL equals %b01101; /* Command was inval tag lookup for NDAL DREAD or IREAD
     constant CMD_O_INVAL equals %b01001; /* Command was inval tag lookup for NDAL OREAD or WRITE
     constant CMD_IPR_DEALLOC equals %b01010; /* Command was tag lookup for IPR deallocate
 FILL_1 bitfield length 22 fill tag $$;
    end PR19BCETSTS_BITS;

constant BCETIDX equals %xA4 tag $;   /* Bcache error tag index (RO)

constant BCETAG equals %xA5 tag $;   /* Bcache error tag (RO)
    PR19BCETAG_BITS structure fill prefix BCETAG$;
     FILL_1 bitfield length 9 fill tag $$;
 VALID bitfield length 1 mask;   /* Valid bit
 OWNED bitfield length 1 mask;   /* Ownership bit
 ECC bitfield length 6 mask;   /* ECC bits
 TAG bitfield length 15 mask;   /* tag data
    end PR19BCETAG_BITS;
```

```
/* Cbox registers, continued

constant BCEDSTS equals %xA6 tag $;   /* Bcache error data status (RW)
    PR19BCEDSTS_BITS structure fill prefix BCEDSTS$;
 LOCK bitfield length 1 mask;   /* Data RAM registers are locked due to an error (WC)
 CORR bitfield length 1 mask;   /* Correctable ECC error occurred (WC)
 UNCORR bitfield length 1 mask;   /* Uncorrectable ECC error occurred (WC)
 BAD_ADDR bitfield length 1 mask;  /* Addressing error occurred (WC)
 LOST_ERR bitfield length 1 mask;  /* Error occurred while register was locked (WC)
 FILL_1 bitfield length 3 fill tag $$;
 DR_CMD bitfield length 4 mask;   /* Data RAM command which caused error (RO)
     constant CMD_DREAD equals %b0111;  /* Command was D-stream data lookup
     constant CMD_IREAD equals %b0011;  /* Command was I-stream data lookup
     constant CMD_WBACK equals %b0100;  /* Command was writeback data lookup
     constant CMD_RMW equals %b0010;   /* Command was read-modify-write data lookup
 FILL_2 bitfield length 20 fill tag $$;
    end PR19BCEDSTS_BITS;

constant BCEDIDX equals %xA7 tag $;   /* Bcache error data index (RO)

constant BCEDECC equals %xA8 tag $;   /* Bcache error ECC (RO)
    PR19BCEDECC_BITS structure fill prefix BCEDECC$;
     FILL_1 bitfield length 6 fill tag $$;
 ECCLO bitfield length 4 mask;   /* Bcache data ECC syndrome bits <3:0>
     FILL_2 bitfield length 12 fill tag $$;
 ECCHI bitfield length 4 mask;   /* Bcache data ECC syndrome bits <7:4>
 FILL_3 bitfield length 6 fill tag $$;
    end PR19BCEDECC_BITS;
```

```
/* Cbox registers, continued

constant CEFADR equals %xAB tag $;   /* Fill error address (RO)

constant CEFSTS equals %xAC tag $;   /* Fill error status (RW)
    PR19CEFSTS_BITS structure fill prefix CEFSTS$;
     RDLK bitfield length 1 mask;   /* Error occurred during a read lock (WC)
 LOCK bitfield length 1 mask;   /* CEFSTS & CEFADR registers are locked due to an error (WC)
 TIMEOUT bitfield length 1 mask;   /* Fill failed due to transaction timeout (WC)
 RDE bitfield length 1 mask;   /* Fill failed due to Read Data Error (WC)
 LOST_ERR bitfield length 1 mask;  /* Error occurred while register was locked (WC)
 ID0 bitfield length 1 mask;   /* NDAL id<0> for failed read (RO)
 IREAD bitfield length 1 mask;   /* Error occured during an IREAD (RO)
 OREAD bitfield length 1 mask;   /* Error occurred during an OREAD (RO)
 WRITE bitfield length 1 mask;   /* Error occurred during a write (RO)
 TO_MBOX bitfield length 1 mask;   /* Data was destined for the Mbox (RO)
 RIP bitfield length 1 mask;   /* READ invalidate was pending (RO)
 OIP bitfield length 1 mask;   /* OREAD invalidate was pending (RO)
 DNF bitfield length 1 mask;   /* Data was not to be validated when fill completed (RO)
 RDLK_FL_DONE bitfield length 1 mask;  /* Last fill for read lock received (RO)
 REQ_FILL_DONE bitfield length 1 mask;  /* Requested fill quadword was received for this read.
 COUNT bitfield length 2 mask;   /* Number of requested QW of fill received (RO)
 FILL_1 bitfield length 4 fill tag $$;
 UNEXPECTED_FILL bitfield length 1 mask;  /* RDE or RDR was received from the NDAL when fill_cam not valid (WC)
 FILL_2 bitfield length 10 fill tag $$;
    end PR19CEFSTS_BITS;
```

```
/* Cbox registers, continued

constant NESTS equals %xAE tag $;    /* NDAL error status (RW)
    PR19NESTS_BITS structure fill prefix NESTS$;
 NOACK bitfield length 1 mask;   /* Outgoing command was NACKed (WC)
 BADWDATA bitfield length 1 mask;  /* BADWDATA cycle transmitted (WC)
 LOST_OERR bitfield length 1 mask;  /* Outgoing error was lost while register was locked (WC)
 PERR bitfield length 1 mask;   /* NDAL parity error detected (WC)
 INCON_PERR bitfield length 1 mask;  /* Inconsistent parity error (parity error detected on
 LOST_PERR bitfield length 1 mask;  /* NDAL parity error detected while register was locked (WC)
       /* ACKed transaction) (WC)
 FILL_1 bitfield length 26 fill tag $$;
    end PR19NESTS_BITS;

constant NEOADR equals %xB0 tag $;    /* NDAL error output address (RO)

constant NEOCMD equals %xB2 tag $;    /* NDAL error output command (RO)
    PR19NEOCMD_BITS structure fill prefix NEOCMD$;
 CMD bitfield length 4 mask;   /* NDAL command on outgoing error transaction (see below)
 ID bitfield length 3 mask;   /* NDAL ID on outgoing error transaction
 FILL_1 bitfield length 1 fill tag $$;
 BYTE_EN bitfield length 8 mask;   /* Byte enables on outgoing error transaction
 FILL_2 bitfield length 14 fill tag $$;
 LEN bitfield length 2 mask;   /* Length on outgoing error transaction (see below)
    end PR19NEOCMD_BITS;

constant NEDATHI equals %xB4 tag $;    /* NDAL error data high (RO)

constant NEDATLO equals %xB6 tag $;    /* NDAL error data low (RO)

constant NEICMD equals %xB8 tag $;   /* NDAL error input command (RO)
    PR19NEICMD_BITS structure fill prefix NEICMD$;
 CMD bitfield length 4 mask;   /* NDAL command received on error transaction (see below)
 ID bitfield length 3 mask;   /* NDAL ID received error on transaction
 PARITY bitfield length 3 mask;   /* NDAL parity bits received error on transaction
 FILL_1 bitfield length 22 fill tag $$;
    end PR19NEICMD_BITS;
```

```
/* Cbox registers, continued

/* Encoded NDAL length values

constant LEN_HW equals %b00 prefix NDAL$;  /* Length = hexaword
constant LEN_QW equals %b10 prefix NDAL$;  /* Length = quadword
constant LEN_OW equals %b11 prefix NDAL$;  /* Length = octaword

/* encoded NDAL command values

constant CMD_NOP equals %b0000 prefix NDAL$;  /* Command = NOP
constant CMD_WRITE equals %b0010 prefix NDAL$;  /* Command = Write
constant CMD_WDISOWN equals %b0011 prefix NDAL$; /* Command = Write disown
constant CMD_IREAD equals %b0100 prefix NDAL$;  /* Command = I-read
constant CMD_DREAD equals %b0101 prefix NDAL$;  /* Command = D-read
constant CMD_OREAD equals %b0110 prefix NDAL$;  /* Command = O-read
constant CMD_RDE equals %b1001 prefix NDAL$;  /* Command = Read data error
constant CMD_WDATA equals %b1010 prefix NDAL$;  /* Command = Write data
constant CMD_BADWDATA equals %b1011 prefix NDAL$; /* Command = Bad write data
constant CMD_RDR0 equals %b1100 prefix NDAL$;  /* Command = Read data return 0
constant CMD_RDR1 equals %b1101 prefix NDAL$;  /* Command = Read data return 1
constant CMD_RDR2 equals %b1110 prefix NDAL$;  /* Command = Read data return 2
constant CMD_RDR3 equals %b1111 prefix NDAL$;  /* Command = Read data return 3
```

```
/* Cbox registers, continued

constant BCTAG equals %x01000000 tag $;   /* First of 64K Bcache tag IPRs (RW)
constant BCTAG_128KB_MAX equals %x0101FFE0 tag $; /* Last tag IPR for 128KB Bcache
constant BCTAG_256KB_MAX equals %x0103FFE0 tag $; /* Last tag IPR for 256KB Bcache
constant BCTAG_512KB_MAX equals %x0107FFE0 tag $; /* Last tag IPR for 512KB Bcache
constant BCTAG_2MB_MAX equals %x011FFFE0 tag $;  /* Last tag IPR for 2MB Bcache
    constant IPR_INCR equals %x20 prefix BCTAG$; /* Increment between Bcache tag IPR numbers
    PR19BCTAG_BITS structure fill prefix BCTAG$;
     FILL_1 bitfield length 9 fill tag $$;
 VALID bitfield length 1 mask;   /* Valid bit (RW)
 OWNED bitfield length 1 mask;   /* Ownership bit (RW)
 ECC bitfield length 6 mask;   /* ECC bits (RW)
 TAG bitfield length 15 mask;   /* tag data (RW)
    end PR19BCTAG_BITS;

constant BCFLUSH equals %x01400000 tag $;  /* First of 64K Bcache tag deallocate IPRs (WO)
constant BCFLUSH_128KB_MAX equals %x0141FFE0 tag $; /* Last deallocate IPR for 128KB Bcache
constant BCFLUSH_256KB_MAX equals %x0143FFE0 tag $; /* Last deallocate IPR for 256KB Bcache
constant BCFLUSH_512KB_MAX equals %x0147FFE0 tag $; /* Last deallocate IPR for 512KB Bcache
constant BCFLUSH_2MB_MAX equals %x015FFFE0 tag $; /* Last deallocate IPR for 2MB Bcache
    constant IPR_INCR equals %x20 prefix BCFLUSH$; /* Increment between Bcache deallocate IPR numbers
```

```
/* Ibox registers.

constant VMAR equals %xD0 tag $;   /* VIC memory address register
    PR19VMAR_BITS structure fill prefix VMAR$;
 FILL_1 bitfield length 2 fill tag $$;
 LW bitfield length 1 mask;   /* longword within quadword
 SUB_BLOCK bitfield length 2 mask;  /* sub-block indicator
 ROW_INDEX bitfield length 6 mask;  /* cache row index
 ADDR bitfield length 21 mask;   /* error address
    end PR19VMAR_BITS;

constant VTAG equals %xD1 tag $;   /* VIC tag register
    PR19VTAG_BITS structure fill prefix VTAG$;
 V bitfield length 4 mask;   /* data valid bits
 DP bitfield length 4 mask;   /* data parity bits
 TP bitfield length 1 mask;   /* tag parity bit
 FILL_1 bitfield length 2 fill tag $$;  /* unused bits (zero)
 TAG bitfield length 21 mask;   /* tag
    end PR19VTAG_BITS;

constant VDATA equals %xD2 tag $;   /* VIC data register

constant ICSR equals %xD3 tag $;   /* Ibox control and status register (RW)
    PR19ICSR_BITS structure fill prefix ICSR$;
 ENABLE bitfield length 1 mask;   /* VIC enable bit (RW)
 FILL_1 bitfield length 1 fill tag $$;
 LOCK bitfield length 1 mask;   /* Register is locked due to an error (WC)
 DPERR bitfield length 1 mask;   /* Data parity error (RO)
 TPERR bitfield length 1 mask;   /* Tag parity error (RO)
 FILL_2 bitfield length 27 fill tag $$;
    end PR19ICSR_BITS;

constant BPCR equals %xD4 tag $;   /* Ibox branch prediction control register
    PR19BPCR_BITS structure fill prefix BPCR$;
 HISTORY bitfield length 4 mask;   /* branch history bits
 FILL_1 bitfield length 1 fill tag $$;
 MISPREDICT bitfield length 1 mask;  /* history of last branch
 FLUSH_BHT bitfield length 1 mask;  /* flush branch history table
 FLUSH_CTR bitfield length 1 mask;  /* flush branch hist addr counter
 LOAD_HISTORY bitfield length 1 mask;  /* write new history to array
 FILL_1 bitfield length 7 fill tag $$;  /* unused bits (must be zero)
 BPU_ALGORITHM bitfield length 16 mask;  /* branch prediction algorithm
     constant BPU_ALGORITHM equals %xFECA; /* default value for BPU_ALGORITHM field
    end PR19BPCR_BITS;

/* The following two registers are for testability and diagnostics use only.
/* They should not be referenced in normal operation.

constant BPC equals %xD6 tag $;   /* Ibox Backup PC (RO)

constant BPCUNW equals %xD7 tag $;   /* Ibox Backup PC with RLOG unwind (RO)
```

```
/* Mbox internal memory management registers.
/* These registers are for testability and diagnostics use only.
/* In normal operation, the equivalent architecturally-defined registers
/* should be used instead.

constant MP0BR equals %xE0 tag $;   /* Mbox P0 base register (RW)

constant MP0LR equals %xE1 tag $;   /* Mbox P0 length register (RW)

constant MP1BR equals %xE2 tag $;   /* Mbox P1 base register (RW)

constant MP1LR equals %xE3 tag $;   /* Mbox P1 length register (RW)

constant MSBR equals %xE4 tag $;   /* Mbox system base register (RW)

constant MSLR equals %xE5 tag $;   /* Mbox system length register (RW)

constant MMAPEN equals %xE6 tag $;   /* Mbox memory management enable (RW)
```

```
/* Mbox registers.

constant PAMODE equals %xE7 tag $;   /* Mbox physical address mode (RW)
    PR19PAMODE_BITS structure fill prefix PAMODE$;
 MODE bitfield length 1 mask;   /* Addressing mode(1 = 32bit addressing) (RW)
     constant PA_30 equals 0;   /* 30-bit PA mode
     constant PA_32 equals 1;   /* 32-bit PA mode
 FILL_1 bitfield length 31 fill tag $$;
    end PR19PAMODE_BITS;

constant MMEADR equals %xE8 tag $;   /* Mbox memory management fault address (RO)

constant MMEPTE equals %xE9 tag $;   /* Mbox memory management fault PTE address (RO)

constant MMESTS equals %xEA tag $;   /* Mbox memory management fault status (RO)
    PR19MMESTS_BITS structure fill prefix MMESTS$;
 LV bitfield length 1 mask;   /* ACV fault due to length violation
 PTE_REF bitfield length 1 mask;   /* ACV/TNV fault occurred on PPTE reference
 M bitfield length 1 mask;   /* Reference had write or modify intent
 FILL_1 bitfield length 11 fill tag $$;
 FAULT bitfield length 2 mask;   /* Fault type, one of the following:
     constant FAULT_ACV equals 1;  /* ACV fault
     constant FAULT_TNV equals 2;  /* TNV fault
     constant FAULT_M0 equals 3;   /* M=0 fault
 FILL_2 bitfield length 10 fill tag $$;
 SRC bitfield length 3 mask;   /* Shadow copy of LOCK bits (see MSRC$ constants below)
 LOCK bitfield length 3 mask;   /* Lock status (see MSRC$ constant below)
    end PR19MMESTS_BITS;

constant TBADR equals %xEC tag $;   /* Mbox TB parity error address (RO)

constant TBSTS equals %xED tag $;   /* Mbox TB parity error status (RW)
    PR19TBSTS_BITS structure fill prefix TBSTS$;
 LOCK bitfield length 1 mask;   /* Register is locked due to an error (WC)
 DPERR bitfield length 1 mask;   /* Data parity error (RO)
 TPERR bitfield length 1 mask;   /* Tag parity error (RO)
 EM_VAL bitfield length 1 mask;   /* EM latch was valid when error occurred (RO)
 CMD bitfield length 5 mask;   /* S5 command when TB parity error occured (RO)
 FILL_1 bitfield length 20 fill tag $$;
 SRC bitfield length 3 mask;   /* Source of original refernce (see MSRC$ constants below) (RO)
    end PR19TBSTS_BITS;

    constant IREF_LATCH equals 6 prefix MSRC$;  /* Source of fault was IREF latch
    constant SPEC_QUEUE equals 4 prefix MSRC$;  /* Source of fault was spec queue
    constant EM_LATCH equals 0 prefix MSRC$;  /* Source of fault was EM latch
```

```
/* Mbox Pcache registers

constant PCADR equals %xF2 tag $;    /* Mbox Pcache parity error address (RO)

constant PCSTS equals %xF4 tag $;    /* Mbox Pcache parity error status (RW)
    PR19PCSTS_BITS structure fill prefix PCSTS$;
 LOCK bitfield length 1 mask;    /* Register is locked due to an error (WC)
 DPERR bitfield length 1 mask;    /* Data parity error occurred (RO)
 RIGHT_BANK bitfield length 1 mask;  /* Right bank tag parity error occurred (RO)
 LEFT_BANK bitfield length 1 mask;  /* Left bank tag parity error occurred (RO)
 CMD bitfield length 5 mask;    /* S6 command when Pcache parity error occured (RO)
 PTE_ER_WR bitfield length 1 mask;  /* Hard error on PTE DREAD occurred (orig ref was WRITE) (WC)
 PTE_ER bitfield length 1 mask;    /* Hard error on PTE DREAD occurred (WC)
 FILL_1 bitfield length 21 fill tag $$;
    end PR19PCSTS_BITS;

constant PCCTL equals %xF8 tag $;     /* Mbox Pcache control (RW)
    PR19PCCTL_BITS structure fill prefix PCCTL$;
 D_ENABLE bitfield length 1 mask;  /* Enable for invalidate, D-stream read/write/fill (RW)
 I_ENABLE bitfield length 1 mask;  /* Enable for invalidate, I-stream read/fill (RW)
 FORCE_HIT bitfield length 1 mask;  /* Enable force hit on Pcache references (RW)
 BANK_SEL bitfield length 1 mask;  /* Select left bank if 0, right bank if 1 (RW)
 P_ENABLE bitfield length 1 mask;  /* Enable parity checking (RW)
 PMM bitfield length 3 mask;    /* Mbox performance monitor mode (RW)
 ELEC_DISABLE bitfield length 1 mask;  /* Pcache electrical disable bit (RW) */
 RED_ENABLE bitfield length 1 mask;  /* Redundancy enable bit (RO) */
 FILL_1 bitfield length 22 fill tag $$;
    end PR19PCCTL_BITS;

constant PCTAG equals %x01800000 tag $;    /* First of 256 Pcache tag IPRs (RW)
constant PCTAG_MAX equals %x01801FE0 tag $;  /* Last of 256 Pcache tag IPRs
    constant IPR_INCR equals %x20 prefix PCTAG$; /* Increment between Pcache tag IPR numbers
    PR19PCTAG_BITS structure fill prefix PCTAG$;
     A bitfield length 1 mask;    /* Allocation bit corresponding to index of this tag (RW)
 V bitfield length 4 mask;    /* Valid bits corresponding to the 4 data subblocks (RW)
 P bitfield length 1 mask;    /* Tag parity (RW)
 FILL_1 bitfield length 6 fill tag $$;
 TAG bitfield length 13 mask;    /* Tag bits (RW)
    end PR19PCTAG_BITS;

constant PCDAP equals %x01C00000 tag $;   /* First of 1024 Pcache data parity IPRs (RW)
constant PCDAP_MAX equals %x01C01FF8 tag $;  /* Last of 1024 Pcache data parity IPRs
    constant IPR_INCR equals %x8 prefix PCDAP$;  /* Increment between Pcache data parity IPR numbers
    PR19PCDAP_BITS structure fill prefix PCDAP$;
     DATA_PARITY bitfield length 8 mask;  /* Even byte parity for the addressed quadword (RW)
 FILL_1 bitfield length 24 fill tag $$;
    end PR19PCDAP_BITS;

end PR19DEF;
end_module $PR19DEF;
```

A.1 Revision History

Table A–1: Revision History

Who	When	Description of change
Mike Uhler	14-Aug-90	Update to latest version
Mike Uhler	01-Dec-89	Initial version

Index