MC88110

SECOND GENERATION RISC MICROPROCESSOR USER'S MANUAL

M MOTOROLA

MOTOROLA

MC88110

Second Generation
RISC Microprocessor
User's Manual

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

Section 2
Programming Model

Section 3
Addressing Modes and Instruction Set Summary

TABLE OF CONTENTS (Continued)

Section 4
Floating-Point Implementation

TABLE OF CONTENTS (Continued)

Section 5
Graphics Unit Implementation

TABLE OF CONTENTS (Continued)

Section 6
Instruction and Data Caches

TABLE OF CONTENTS (Continued)

Section 7
Exceptions

TABLE OF CONTENTS (Continued)

Section 8
Memory Management Units

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

Section 9
Instruction Timing and Code Scheduling Considerations

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

Section 10
Instruction Set

Section 11
System Hardware Design

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Concluded)

Appendix A
Bit Scan Bit Definition

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Continued)

LIST OF ILLUSTRATIONS (Continued)

LIST OF ILLUSTRATIONS (Continued)

LIST OF ILLUSTRATIONS (Continued)

LIST OF ILLUSTRATIONS (Continued)

LIST OF ILLUSTRATIONS (Concluded)

LIST OF TABLES

LIST OF TABLES (Continued)

LIST OF TABLES (Concluded)

SECTION 1
OVERVIEW

The MC88110 is the second implementation of the 88000 family of reduced instruction set computer (RISC) microprocessors. The MC88110 is a Symmetric Superscalar™ machine capable of issuing and retiring two instructions per clock without any special alignment, ordering, or type restrictions on the instruction stream. Instructions are issued to multiple execution units, execute in parallel, and can complete out of order, with the machine automatically keeping results in the correct program sequence. This symmetric superscalar design allows sustained performance to approach the peak performance capability.

The MC88110 uses dual instruction issue and simple instructions with extremely rapid execution times to yield maximum efficiency and throughput for 88000 systems. Instructions either execute in one clock cycle, or effective one clock cycle execution is achieved through internal pipelining. Ten independent execution units communicate with a general register file and an extended register file through multiple 80-bit internal buses. Each of the register files has sufficient bandwidth to supply four operands and receive two results per clock cycle. Each of the pipelined execution units, including those that execute floating-point and data movement instructions, can accept a new instruction and retire a previous instruction on every clock cycle.

In a single chip implementation, the MC88110 integrates the central processing unit (CPU), floating-point unit (FPU), graphics processing unit (GPU), virtual memory address translation, instruction cache, and data cache.

The CPU contains two arithmetic logic units (ALUs) that allow two integer instructions to issue and execute in each clock cycle. The multiply and floating-point add execution units are fully pipelined and provide the same high performance for single-, double-, and double-extended-precision floating-point operations.

The graphics processing unit provides dedicated hardware to allow direct manipulation of pixel-oriented data types. This ability, combined with exceptional floating-point performance and high data throughput, allows the MC88110 to provide high performance three-dimensional (3D) graphics capability, including shading, Z-buffering, and compositing.

Symmetric Superscalar is a trademark of Motorola, Inc.

1

The MC88110 also includes two on-chip 8K-byte caches and two on-chip memory management units (MMUs): one cache and MMU for instructions and one cache and MMU for data. Additionally, on-chip logic maintains data cache coherency in multiprocessor applications.

The MC88110 maintains compatibility with MC88100 user application software. Also, a full line of highly optimizing compilers, operating systems, application programs, and development tools has been developed for the 88000 family.

This section provides an overview of the MC88110, including a feature list and an overview of the 88000 family. In addition, there is a block diagram of the MC88110, a description of each execution unit, the MC88110 execution model, and a brief summary of the instruction set. Instruction mnemonics used in this section are defined in detail in **Section 10 Instruction Set**.

1.1 FEATURE LIST

The major features of the MC88110 are as follows:

- Symmetric Superscalar Design Which Issues Two Instructions Per Clock
- Ten Independent Execution Units and Two Eight Ported Register Files:
    - —Superscalar Instruction Unit
    - —80-Bit Integer, Floating-Point, and Graphics Multiply Execution Unit
    - —80-Bit Integer and Floating-Point Divide Execution Unit
    - —80-Bit Extended-Precision Floating-Point Add Execution Unit
    - —Two 64-Bit 3D Graphics Execution Units
    - —Two 32-Bit Integer Arithmetic Logic Execution Units
    - —32-Bit Bit-Field Execution Unit
    - —Data Unit with Load Buffer and Store Reservation Station
    - —Thirty-Two 32-Bit General Registers for Operand Storage
    - —Thirty-Two 80-Bit Extended Registers for Additional Floating-Point Operand Storage
- High Performance Instruction Execution
    - —Single-Clock Integer, Logical, Bit-Field, and Graphics Operations
    - —Single-, Double-, and Double-Extended-Precision IEEE 754 Floating-Point Compatibility (Up to Two Operations Executed per Clock Cycle)
    - —Pipelined Load and Store Operations

- High Performance Instruction and Data Throughput
  - —Internal Harvard Architecture
  - —Separate 8K-byte Instruction and Data Caches: 2-Way Set-Associative, Physically Addressed
  - —80-Bit Internal Data Paths
  - —32-Entry Branch Target Instruction Cache for Branch Acceleration
  - —Run-Time Reordering of Loads and Stores
  - —Speculative Instruction Execution
  - —Register Scoreboard Managing Data Dependencies in Hardware
  - —Decoupled Data Cache Accesses
  - —Data Cache Write-Through and Write-Back Operation
- Extensible Architecture Facility Through Special Function Units
- Facilities for Enhanced System Performance
  - —64-Bit Split-Transaction External Data Bus with Burst Transfers
  - —Hardware Enforced Data Cache Coherency (Bus Snooping) for Multiprocessor Applications
  - —Critical-Word-First Burst Cache Line Fills with Instruction and Data Streaming
  - —Instruction and Data Address Translation Caches with Page and Block Entries
  - —High-Speed Interrupt Processing with Minimal Interrupt Latency
- System Software Flexibility
  - —Hardware or Software Address Translation Cache Refill
  - —Data Address Breakpoints for Software Debugging
  - —Selectable Big-Endian or Little-Endian Byte Ordering
- JTAG Boundary Scan for In-System Testability

1.2 88000 FAMILY OVERVIEW

The following paragraphs give an overview of the features which are common to all members of the 88000 family, including the register-to-register architecture, the simplified addressing modes, the instruction formats, the special function units (SFUs), and the optimizing software.

1.2.1 Register-to-Register Architecture

The 88000 family defines register-to-register operations for all computational instructions. Source operands for these instructions are accessed from the on-chip registers or are provided as immediate values embedded in the instruction opcode. The computational instruction results are stored in separate on-chip registers, allowing source operand registers to be reused in subsequent instructions. Data is transferred between memory and registers with load and store instructions only.

1.2.2 Simplified Addressing Modes

The 88000 family has simplified addressing modes for memory and register accesses. Address calculations are simple, efficient, and execute quickly. All computational instructions are implemented as register-to-register or register-plus-immediate-value instructions which eliminates memory access delays in these instructions.

1.2.3 Instruction Formats

All 88000 instructions are implemented as single-word (32-bit) opcodes. Formats are consistent across all instruction types, allowing for efficient decoding in parallel with operand accesses. This fixed instruction length and consistent format greatly simplifies instruction pipelining.

1.2.4 Levels of Privilege

The 88000 family has two instruction execution modes: supervisor mode and user mode. The supervisor mode is the higher privilege level. In supervisor mode, memory and control register access is unrestricted. The supervisor mode is typically used by operating systems and other system-level resources. The user mode is the lower privilege level. In user mode, resource access is limited to the user memory space, general registers, extended registers, and some floating-point control registers. Application software is typically executed in user mode.

1.2.5 Special Function Units

The 88000 family provides the flexibility for extensions, as technology and applications evolve, through the definition of special function units (SFUs) within the overall instruction mapping. An SFU is defined as a set of instructions, with a common opcode field, which provides additional functionality to the base architecture. The architecture defines a base instruction set and seven reserved sets of opcodes to support up to seven SFUs. These SFUs may or may not be included in an implementation of the architecture. Any SFU can be added to, or removed from, a given implementation of the 88000 family with no impact on the base architecture.

In addition to the base instruction set, the MC88110 implements two SFUs: the floating-point unit, which is SFU1, and the graphics processing unit, which is SFU2. Figure 1-1 illustrates how the concept of SFUs is integrated with the MC88110 hardware. In this diagram, each of the boxes represent hardware on the MC88110. The top box, representing the instruction fetch and decode circuitry, is part of the instruction unit. The three ovals are a conceptual representation of the base instruction set and the two SFU instruction sets. When an SFU is enabled, the instructions in that SFU's instruction set can be issued.
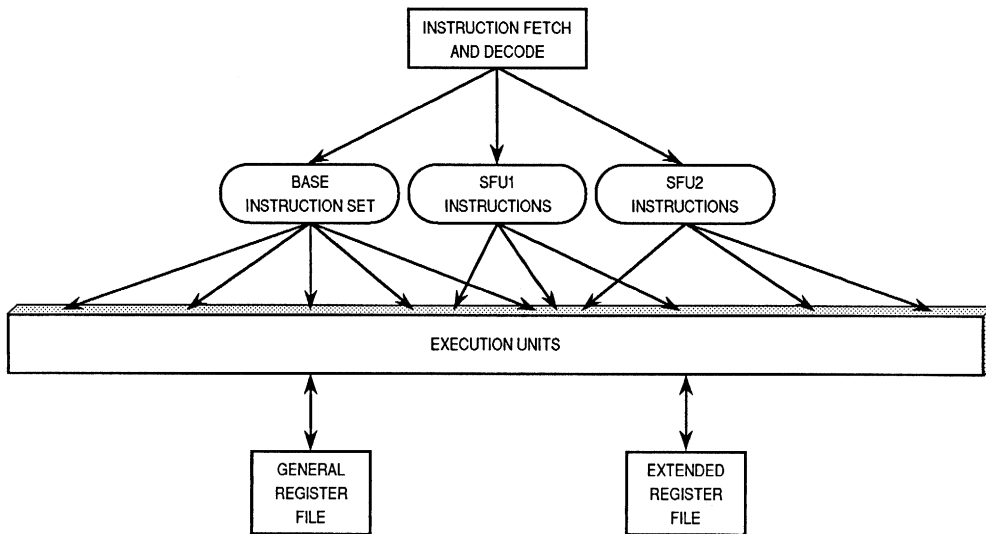
Figure 1-1. SFU Conceptual Diagram

The instruction unit fetches, decodes, and issues each instruction to the appropriate execution unit. The instructions fall into one of three categories: those in the base instruction set, SFU1, or SFU2. The execution units receive source data from the register files or from the instruction opcode and perform the specified operation. When the results are ready, they are written to the appropriate register file.

Figure 1-2 shows which execution units are used by each of the instruction sets. The base instruction set uses the instruction, data, integer, bit-field, divide, and multiply execution units. The SFU1 (floating-point) instruction set uses the divide, multiply, and floating-point add execution units. The SFU2 (graphics) instruction set uses the multiply, pixel add, and pixel pack execution units. Note that the general register file can provide data for and receive data from all ten of the execution units and all three of the instruction sets. The extended register file can only be accessed by the SFU1 instruction set and those instructions in the base instruction set that transfer data to and from memory.
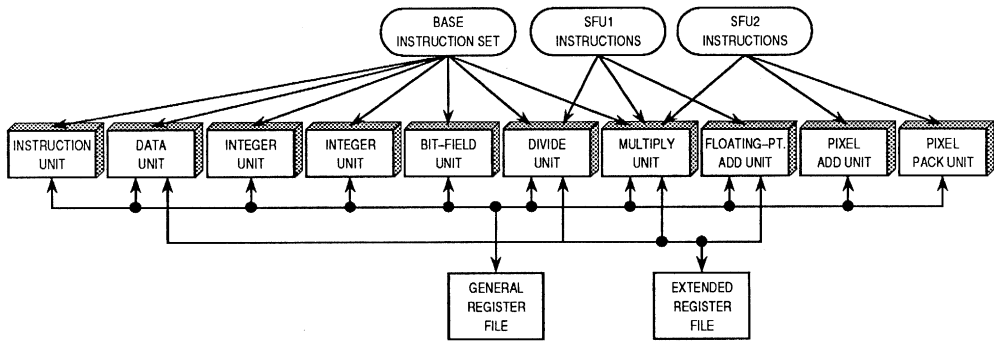
Figure 1-2. SFU Hardware Use

1.2.6 Optimizing Software

Optimizing compilers, linkers, and operating systems, which have been designed in conjunction with the design of the MC88110, are essential contributors to its performance. This software performs optimizations based on the multiple, independent execution units of the MC88110; instructions are scheduled to maximize parallelism and instruction throughput. This software also makes efficient use of the MC88110 instruction set and register model.

A register usage convention has been established that supports the cross-linking of procedures from various compilers and languages. With this convention, compilers and linkers allocate the general registers in a manner that minimizes data movement to and from memory, even during procedure calls.

1.3 MC88110 PROCESSOR OVERVIEW

The MC88110 contains ten execution units (see Figure 1-3) which operate independently and concurrently. The integer, floating-point, graphics, multiply, and divide execution units perform computational operations. The data unit performs the data memory accesses, while the instruction unit performs instruction fetches and many of the control functions for the MC88110.

The integer execution units include two identical ALUs, which perform 32-bit arithmetic and logic operations, and one bit-field execution unit, which performs all bit-field operations. The multiply execution unit handles all integer, floating-point, and graphics multiply instructions. The divide execution unit handles integer and floating-point divide instructions. The floating-point add execution unit handles the remaining floating-point arithmetic instructions. The graphics execution units include a pixel adder, which performs the remaining graphics arithmetic instructions, and the pixel packer, which performs pixel pack and unpack functions.

In addition to the execution units, the MC88110 contains a general register file and an extended register file. The MC88110 also has six 80-bit internal buses that are used for passing operands between the register files and the different execution units: four of the

buses provide the execution units with source data from the register files or the instruction encoding, and two buses return the results from the execution units to the register files.

To speed up memory accesses and instruction fetching, the MC88110 has one cache and MMU for data accesses, and one cache and MMU for instruction fetches. The data cache contains duplicate address tags to facilitate snooping in multiprocessor environments. There is also a target instruction cache (TIC), which contains the target instructions for recently taken branches.

The bus interface unit arbitrates between external instruction and data accesses and controls the external bus.
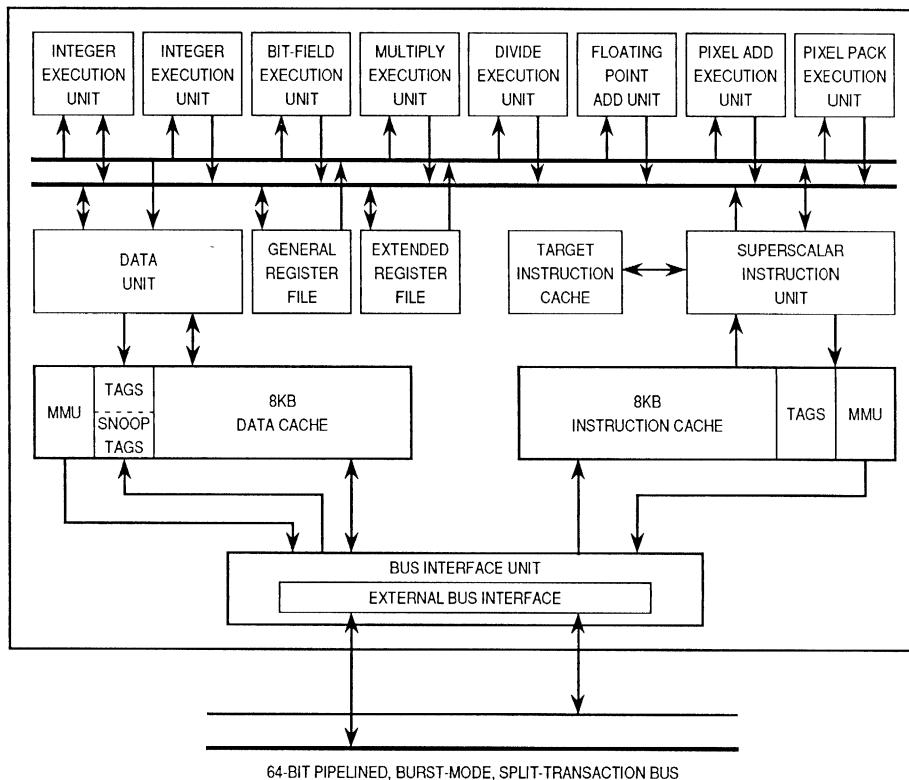
Figure 1-3. MC88110 Block Diagram

1.3.1 Internal Buses

The MC88110 has two source 1 buses, two source 2 buses and two destination buses. These 80-bit buses perform all internal data transfers between the register files and the execution units. The source 1 and source 2 buses transfer source operands to the execution units. All source data originates from the register files or from 16-bit immediate values embedded in instructions. The destination buses transfer data from the execution units to the register files.

Arbitration for the internal buses is performed by the sequencer, which is a part of the instruction unit. The contents of the source registers for an instruction are gated onto the source buses under control of the sequencer. When an execution unit completes an instruction, it requests a slot on a destination bus. Since there are only two destination buses and more than two instructions can complete at any time, the sequencer prioritizes the data transfers on the destination buses.

1.3.2 General Register File

The general register file (GRF) consists of thirty-two 32-bit registers which are designated as **r0** through **r31**. The **r0** register always contains the constant zero and can be used by instructions requiring the constant zero as an operand. The GRF can provide operands for all computational instructions, can serve as the data source or destination for load and store instructions, and can provide addresses for branch and memory-access instructions.

The GRF has six output ports and two input ports. Four of the six output ports allow source operands to be simultaneously placed on the two source 1 and two source 2 buses so that two instructions can be executed per clock. The last two output ports are used to write the contents of the destination registers for the current instructions into the history buffer. (For more information on the history buffer, see **Section 7 Exceptions**.) The input ports are used to move the results from completed instructions from the two destination buses into destination registers.

1.3.3 Extended Register File

The extended register file (XRF) consists of thirty-two 80-bit extended registers which are designated as **x0** through **x31**. The **x0** register always contains the constant zero and can be used by instructions requiring the constant zero as an operand. The remaining registers in the XRF can contain data objects of any of the three defined floating-point data formats: single-, double-, or double-extended-precision. The extended registers can provide operands for all floating-point instructions and can serve as the data source or destination for load and store instructions.

The XRF has six output ports and two input ports. Four of the six output ports allow source operands to be simultaneously placed on the two source 1 and two source 2 buses so that two instructions can be executed per clock. The last two output ports are used to write the contents of the destination registers for the current instructions into the history buffer. (For more information on the history buffer, see **Section 7 Exceptions**.)

The input ports are used to move the results from completed instructions from the two destination buses into destination registers.

1.3.4 Integer Execution Units

There are three integer execution units: two identical ALUs and one bit-field unit. Each of the integer execution units completes instruction execution in one clock cycle. Since there are two identical ALUs, two ALU instructions can be issued simultaneously; therefore, arithmetic and logical instructions are never stalled due to the execution units being unavailable. Integer multiply and divide are multi-cycle instructions and are executed by the multiply and divide execution units, not by the ALUs.

1.3.5 Multiply and Divide Execution Units

The multiply unit executes all integer, floating-point, and graphics multiplies; the divide unit executes all integer and floating-point divides. The multiply unit is implemented as a three-stage pipeline; therefore, since all multiplies are three-cycle instructions, one multiply can be issued in each clock cycle. The divide unit is an iterative multi-cycle execution unit, so only one divide instruction can be executing at any time.

1.3.6 Floating-Point Function Unit

The FPU, implemented as SFU1, provides high performance mixed-mode operations for single-, double-, and double-extended-precision floating-point data. The FPU operations are executed in either the multiply, divide, or floating-point add execution units. Floating-point operands can be stored in either the general register file or the extended register file. The MC88110 also implements three control registers to support the FPU.

The floating-point add execution unit performs the integer/floating-point conversion instructions and all floating-point arithmetic instructions except the multiply and divide. The floating-point add unit is implemented as a three-stage pipeline; therefore, since floating-point adds are three-cycle instructions, one floating-point add can be issued in each clock cycle. The floating-point multiply and divide instructions are executed by the multiply and divide execution units.

The three control registers associated with the FPU are the floating-point exception cause register (FPECR), the floating-point status register (FPSR), and the floating-point control register (FPCR). Information about the cause of floating-point exceptions is recorded in the FPECR. This register is privileged and can only be accessed by supervisor code. The FPSR and FPCR contain information on IEEE exception conditions (divide by zero, overflow, etc.) and control the floating-point rounding mode. The FPSR and FPCR are not privileged and can be accessed by either user or supervisor code. The FPU control registers are described in detail in **Section 4 Floating-Point Implementation**.

1.3.7 Graphics Processing Function Unit

The process of rendering realistic animated 3D images in real time is computationally intensive. The process has five major steps: 1) viewpoint transformation, 2) lighting, 3) raster conversion, 4) image processing, and 5) display. Because of its exceptional floating-point performance, the MC88110 is capable of rapidly performing viewpoint transformation and lighting calculations on complex images. The flexible computational instructions and high data throughput of the MC88110 allow efficient coding of the bit block transfer algorithm (bitblt) and other algorithms necessary to achieve good display system performance. To achieve good interactive performance, raster conversion, and image processing phases requires hardware support beyond that found in most conventional microprocessors. The graphics processing function unit (GPU), implemented as SFU2, is targeted at improving the performance of these phases of the rendering process.

The MC88110 includes two independent execution units to support the GPU: the pixel add execution unit and the pixel pack execution unit. Graphics operands are made up of multiple pixels of varying length, which are packed into 64-bit fields and stored in register pairs in the general register file. The graphics instructions process the individual fields within the 64-bit fields in parallel, avoiding the need to separate them and operate on them individually. The graphics multiply instruction is executed by the multiply execution unit.

1.3.8 Instruction Unit/Sequencer

The MC88110 contains an instruction unit/sequencer which provides centralized control of data flow among the execution units and the register files. The instruction unit/sequencer enforces data interlocks, directs data from the register files onto and off of the buses, maintains a state history of the processor's actions, and performs the flow control instructions. The following paragraphs describe the instruction unit and the sequencer.

1.3.8.1 INSTRUCTION UNIT. The instruction unit fetches instruction pairs from the instruction cache, performs the first steps of instruction decode, and provides instructions to the appropriate execution units via encoded internal control signals. The instruction unit also executes flow control instructions and performs other related tasks such as exception processing. In addition, the register scoreboard and the general control registers are contained in the instruction unit.

1.3.8.1.1 Program Flow. The instruction unit fetches instructions from the cache as dictated by program flow. Program flow includes sequential accesses, jump and branch instructions, and exception vectoring.

The instruction unit executes all flow control instructions. It calculates the return pointer for jump to subroutine (**jsr**) and branch to subroutine (**bsr**) instructions and saves the return pointer in register one (**r1**) of the general register file. The return pointer is either the address of the first instruction after the **jsr** or **bsr** instruction, or the address of the second instruction after the **jsr.n** or **bsr.n** instruction (**.n** indicates delayed branching).

1.3.8.1.2 Exception Processing. The instruction unit includes two features which are used for exception processing: the vector base register (VBR) and the history buffer. The VBR points to a memory page containing all of the exception vectors. When an exception occurs, the exception target address is computed using the value in the VBR.

The history buffer is a first-in-first-out (FIFO) queue which records relevant machine state information at the time each instruction is issued. Each instruction remains in the history buffer until it completes execution and all instructions which were issued before it complete execution. When an exception occurs, the effects of any instructions which completed out of order before the faulting instruction are eliminated using the information from the history buffer. Any instructions issued before the faulting instruction are allowed to complete execution before exception processing begins.

1.3.8.1.3 Register Scoreboard. Instructions in a code sequence begin execution sequentially but can complete out of order. To avoid register conflicts between instructions which are executed out of order, the instruction unit contains a register scoreboard for the general register file and the extended register file. The register scoreboard keeps track of which registers are unavailable and which are ready for use.

Every register except registers **r0** and **x0** has a dedicated bit in the register scoreboard. When an instruction is issued that takes longer than one clock cycle to execute, the scoreboard bit corresponding to the destination register is set. When the instruction finishes execution, the register becomes available, and the scoreboard bit is cleared.

When an instruction requires the contents of a register and/or needs to use a register as a destination, the appropriate scoreboard bit or bits are checked to determine if the register(s) are available. If the required registers for an instruction are flagged as in use in the register scoreboard (i.e., one of the required registers is the destination register for a previous instruction which is still executing), execution of the instruction is delayed until the required registers become available. In this case, the appropriate scoreboard bits are checked by the instruction unit on each clock cycle until all the registers are available. If the second instruction of an issue pair requires a register which is specified as the destination for the first instruction of that issue pair, then execution of the second instruction is delayed until the first instruction completes execution.

1.3.8.1.4 General Control Registers. The instruction unit also contains the general control registers which include supervisor-only storage registers, a processor identification register (PID), and a processor status register (PSR). The function of the storage registers is programmer defined. The general control registers also include several exception-time registers and registers for the control of the data and instruction caches and MMUs.

1.3.8.2 SEQUENCER. The sequencer performs register write-back arbitration and exception arbitration, and generates control signals for the instruction unit and the internal buses.

When an execution unit has a result to write to a register, the execution unit requests the write-back arbiter to grant a slot on the destination bus. If an interrupt is pending, the

write-back arbiter prohibits register write-back grants except for memory-access results. If no interrupt is pending, the write-back arbiter generates a control signal that gates the data onto a destination bus and into the selected register. If three or more execution units request a slot, the write-back arbiter grants the two available write-back slots according to a defined priority scheme. In this scheme, one-cycle instructions have priority over instructions from multi-stage pipeline execution units.

If data on the destination bus is needed immediately by another instruction, the sequencer sends a control signal which causes the data to be forwarded directly from the destination bus onto the selected source bus in addition to being written into the appropriate register. This feature is called feed-forwarding.

The exception arbiter controls exception recognition and resolves recognition of multiple exceptions according to the priority of the exceptions. Interrupts have priority over internally generated exceptions (except for data access exceptions); however, there is no priority associated with internally generated exceptions, so they are handled in order. Exceptions are described fully in **Section 7 Exceptions**.

1.3.9 Instruction Cache

The MC88110 has an 8K-byte, 2-way set associative, physically addressed instruction cache. The instruction cache is 2-way set associative to maximize the hit rate, and uses physical address tags so the cache does not need to be flushed on a context switch. Instruction cache coherency is maintained by software and supported by a fast hardware invalidation capability.

The instruction cache is configured as 128 sets which contain two lines each. Each line contains eight 32-bit words, an address tag, and a valid bit. A block diagram of the instruction cache organization is shown in Figure 1-4.
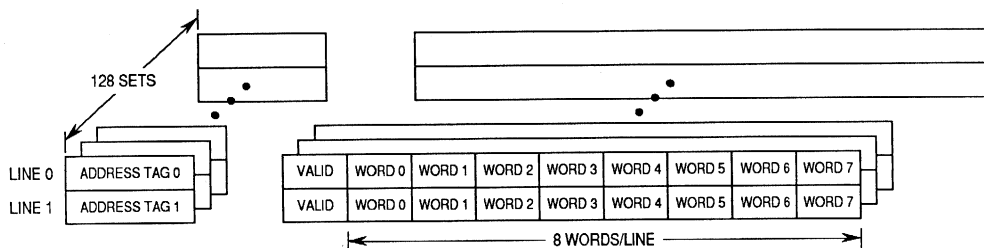
Figure 1-4. Instruction Cache Organization

Each instruction cache line contains eight contiguous words from memory which are loaded from an 8-word boundary (i.e., bits A4–A0 of the logical addresses are zero); thus, a cache line will never cross a page boundary. All bus operations that load instructions into the cache from memory are performed on a line basis (i.e., an entire line

is filled). New lines are allocated into empty cache lines if any are available. A pseudorandom replacement algorithm is used to select a cache line when no empty lines are available.

Bus transactions that load instructions into the cache always begin with the address of the missed word, regardless of the word's location within a cache line. The missed word is transferred to the instruction unit as soon as it is received from the bus so that instruction issue can be resumed as quickly as possible.

On each clock cycle, the instruction unit provides the cache with the address of the first instruction of the next instruction pair to be executed. In the case of a cache hit, the instruction cache returns both the referenced instruction and the one following it; thus, the instruction unit is provided with two instructions in each clock cycle as long as a cache miss does not occur.

1.3.10 Target Instruction Cache

The MC88110 has a TIC, which is a fully associative 32-entry logically addressed cache. Each entry in the TIC contains the first two instructions of a branch target instruction stream, a 31-bit logical address tag, and a valid bit. The 31-bit logical address tag holds a supervisor/user bit and the upper 30 bits of the address of the branch instruction.

When a branch instruction occurs, the TIC is accessed (using the address of the branch) in parallel with the decode of the branch instruction. If there is a TIC hit, the two instructions corresponding to the branch instruction are sent from the TIC to the instruction unit. The instruction unit can then issue those instructions if the branch is taken, eliminating much of the delay associated with changes in instruction flow. The details of the operation of the TIC are discussed in **Section 9 Instruction Timing and Code Scheduling Considerations**.

1.3.11 Instruction MMU

The instruction MMU provides two 4G-byte logical address spaces: one for supervisor code and one for user code. The MMU enforces access privileges for these spaces on block and page levels. Used and modified status is maintained by software for each page to assist implementation of a demand-paged virtual memory system.

Memory management performance is maximized by two instruction address translation caches (ATCs) that provide address translation in parallel with no time penalty. The ATCs consist of the page address translation cache (PATC) and the block address translation cache (BATC). The PATC is a 32-entry, fully-associative cache which contains translations for 4K-byte memory pages. The PATC is automatically maintained by MC88110 hardware or can be maintained by system software. The BATC is an 8-entry, fully-associative cache that contains translations for block sizes ranging from 512K-byte to 64M-byte. The BATC entries are managed by system software.

1.3.12 Data Unit

The data unit interfaces with the data cache and MMU and executes instructions that access data memory. The data unit contains a dedicated calculation unit for address computation. Addresses are formed by adding the source 1 register operand specified by the instruction to either a source 2 register operand or a 16-bit immediate value embedded in the instruction. This address is sent to the data cache, which performs the memory access.

Memory accesses are pipelined in the data unit. The data unit contains a series of load address buffers and store address/data buffers, which operate as two independent FIFO queues. These queues are the load buffer and the store reservation station. After being issued, all load (**ld**) and store (**st**) instructions pass through the appropriate buffer or reservation station.

The data unit executes the buffered load and store instructions as cache, memory, and data become available. The data unit always executes **ld** instructions in program order with respect to other **ld** instructions. Likewise, **st** instructions are executed in program order with respect to other **st** instructions. However, **ld** instructions are allowed to execute out of order with respect to **st** instructions. In the event that a **st** instruction is stalled in the store reservation station waiting for data from a previous computation, subsequent **ld** instructions can bypass the pending **st** instruction and can have access to the memory system. To ensure memory consistency, the MC88110 compares load addresses to store addresses and does not allow **ld** instructions to run ahead of **st** instructions for which there is an address match. If necessary, all loads and stores can be forced to run in strict program sequence by setting a bit in the processor status register (see **Section 2 Programming Model**).

1.3.13 Data Cache

The MC88110 includes an 8K-byte, 2-way set-associative, physically addressed data cache. The data cache is 2-way set-associative to maximize the hit rate and uses physical address tags so the cache does not need to be flushed on a process switch. The data cache supports both write-through and write-back memory update policies which are selectable on a page-by-page or block-by-block basis.

The data cache is configured as 128 sets which contain two lines each. Each line contains eight 32-bit words, an address tag, and status bits. A block diagram of the data cache organization is shown in the Figure 1-5.

128 SETS

LINE 0 | ADDRESS TAG 0

LINE 1 | ADDRESS TAG 1

| STATE | WORD 0 | WORD 1 | WORD 2 | WORD 3 | WORD 4 | WORD 5 | WORD 6 | WORD 7 |

| STATE | WORD 0 | WORD 1 | WORD 2 | WORD 3 | WORD 4 | WORD 5 | WORD 6 | WORD 7 |

|← 8 WORDS/LINE →|
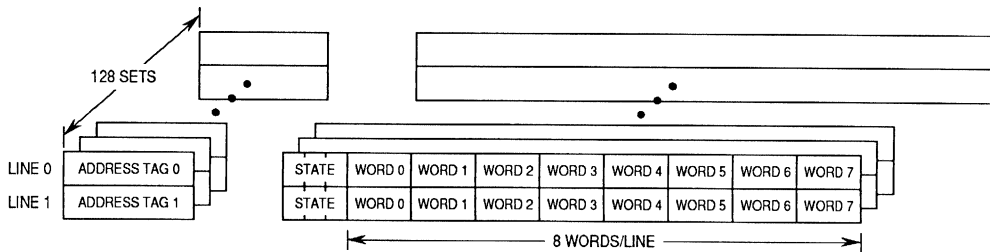
Figure 1-5. Data Cache Organization

Each data cache line contains eight contiguous words from memory which are loaded from an 8-word boundary (i.e., bits A4–A0 of the logical addresses are zero); thus, a cache line will never cross a page boundary. All bus operations that load data into the cache from memory are performed on a line basis (i.e., an entire line is filled). New lines are allocated into empty cache lines if any are available. A pseudorandom replacement algorithm is used to select a cache line when no empty lines are available.

Bus transactions that load data into the cache always begin with the address of the missed word, regardless of the word's location within a cache line. The missed word is transferred to the data unit as soon as it is received from the bus so that instruction execution can be resumed as quickly as possible.

The data cache provides a decoupling feature to improve cache performance. When the decoupling feature is enabled, the data unit can continue making cache accesses while the data cache is waiting to receive data from the bus. These cache accesses are called decoupled cache accesses. If a decoupled cache access hits in the cache and does not require an external bus transaction, the access is allowed to complete. If a decoupled cache access requires an external bus transaction, no further decoupled accesses are allowed, and the cache access is restarted when the cache is available.

Data cache coherency is automatically maintained by hardware bus snooping. There are duplicate address tags and dual-ported state bits associated with each line in the cache to prevent snooping traffic on the bus from interfering with processor operation and degrading performance.

1.3.14 Data MMU

The data MMU provides two 4G-byte logical address spaces: one for supervisor data and one for user data. The MMU enforces access privileges for these spaces on block and page levels. Used and modified status is maintained by software for each page to assist implementation of a demand-paged virtual memory system.

Memory management performance is increased by two data ATCs that provide address translation with no time penalty. The ATCs consist of the PATC and the BATC. The PATC is a 32-entry, fully-associative cache which contains translations for 4K-byte memory pages. The PATC is automatically maintained by MC88110 hardware or can be

maintained by system software. The BATC is an eight-entry, fully-associative cache that contains translations for block sizes ranging from 512K-byte to 64M-byte. The BATC entries are managed by system software.

1.3.15 External Bus Overview

The MC88110 external bus interface includes a 32-bit address bus, a 64-bit data bus, 48 control and information signals, and 8 test pins (see Figure 1-6). The address of the instruction or data needed by the processor is driven on the address bus. Similarly, the requested instruction or data is transferred to the processor on the data bus. The bus interface control and information signals include the byte parity, transfer attribute, arbitration, transfer control, snoop control, processor status, and interrupt signals. There are also eight test pins used to test selected internal circuitry.
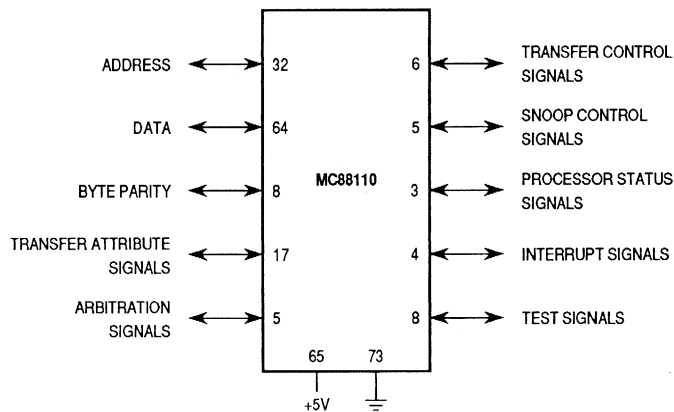
Figure 1-6. MC88110 External Bus Interface

The data bus can support transfer sizes of 8-, 16-, 32-, or 64- bits in one bus cycle. Data transfers occur in either single-beat transactions or four-beat burst transactions. A single-beat transaction is a data transfer of 64 bits or less. Single-beat transactions are caused by noncached accesses which access memory directly (i.e. reads and writes when caching is disabled, cache inhibited accesses, invalidation cycles, **xmem** transactions, writes in write-through/store-through mode, and allocate loads). Burst transactions, made up of four consecutive two-word transfers, are initiated when an entire line in the cache is read from or written to memory.

The MC88110 bus supports multiple processors with a built-in cache coherency mechanism called bus snooping. Bus snooping is a technique whereby all devices on the bus monitor all transactions to ensure that all local copies of data (in caches) remain consistent.

The MC88110 supports split bus transactions in which different processors can have ownership of the address bus and data bus at the same time. This potentially increases

system performance by allowing multiple bus transactions to be in progress simultaneously. The bus also supports pipelining, which allows the address phase of a transaction to overlap the data phase of other transactions. The complexity of the pipeline levels is dependent on external circuitry.

1.3.16 System Debugging Features

The MC88110 contains a debug signal, breakpoint registers, and dedicated user-accessible test logic to facilitate the debugging of MC88110 systems. The debug signal, when asserted, disables all caches, MMUs, and breakpoints. This forces all instruction and data accesses to appear on the bus, making it easier to track program flow.

The data MMU contains two data breakpoint registers which can be used by a debugger program to force an exception to occur when accesses are made to specified logical addresses. If the data breakpoints are enabled, the MMU compares the logical address of each access to the 32-bit logical address in each of the breakpoint registers. If there is a match, then a data access exception is taken. For more information on the breakpoint registers, see **Section 8 Memory Management Units**.

The dedicated user-accessible test logic is fully compatible with IEEE Standard 1149.1-1990 *Standard Test Access Port and Boundary Scan Architecture*. The test logic is implemented using static logic design and is independent of the system logic of the device. The test logic includes a 16-state controller, two test data registers (the bypass register and the boundary scan register), and a test access port that consists of five dedicated signal pins. The boundary scan register links all device signal pins into a single shift register.

The MC88110 test logic provides the capability to perform the following procedures:
1. Boundary scan operations to test circuit board electrical continuity.
2. Bypass the MC88110 for a given circuit board test by effectively replacing the test data register by single cell (the bypass register).
3. Sample the MC88110 system pins during operation and transparently shift out the result through the boundary scan register.
4. Disable the output drive of all input/output pins and output pins during circuit board testing. The single-bit bypass register is selected when in the output drive disabled mode.

1.4 EXECUTION MODEL

The following paragraphs briefly describe the register set and some general timing considerations. This section also includes a listing of the MC88110 instruction set.

1.4.1 Register Set

The MC88110 has two programming models: one that corresponds to the supervisor mode of operation and one that corresponds to the user mode of operation. The programming models incorporate three types of registers that provide data and

execution information to the execution units. The following list briefly describes the three types of registers:

1. General Registers (**r31–r0**)—These registers can contain program data (source operands and instruction results). All of these registers have read/write access. Register **r0** contains the constant zero, and writing to **r0** has no effect on the register.

2. Extended Registers (**x31–x0**)—These registers can contain floating-point data (source operands and instruction results). All of these registers have read/write access. Register **x0** contains the constant zero, and writing to **x0** has no effect.

3. Control Registers—These registers contain status, execution control, and exception processing information. Some of these registers have read/write access, while others are read-only. Most control registers can be accessed only in supervisor mode.

1.4.2 General Timing Considerations

A superscalar machine is one which can issue multiple instructions concurrently from a conventional linear instruction stream. The MC88110 is a superscalar implementation of the 88000 architecture in which two instructions are decoded and issued to multiple execution units during each clock cycle. Any complications due to the superscalar implementation are transparent to the software.

There are several factors which affect instruction issue timing. These factors include the following:

- Whether instructions can be prefetched from the instruction cache (a cache hit), or must be fetched from main memory (a cache miss).
- Whether data dependencies exist which will force an instruction stall while source data is being generated.
- Whether execution units are available to accept additional instructions.
- Whether the history buffer is full.

Instructions are issued to the execution units in strict program sequence. If the first instruction in an issue pair cannot be issued, then neither instruction in the pair is issued. If the first instruction in the pair is issued but the second cannot, then the second instruction is moved into the vacated first-issue position, and a new instruction is placed in the second-issue position. If both instructions in the pair are issued, then two new instructions are fetched from the instruction cache to be issued in the next clock cycle.

When two instructions are considered for issue in the same clock cycle, there are no restrictions placed on instruction type or address alignment for either instruction in the issue pair. In other words, instructions in either slot can be from any word-aligned memory location and can be issued to any execution unit (provided it is available and there are no data dependencies). This is known as symmetric superscalar instruction issue.

Figure 1-7 illustrates symmetric superscalar instruction issue. In this illustration, instruction N is not bound to be issued to any particular execution unit. Similarly, instruction N+1 is free to be issued to any available execution unit. This feature frees the compiler/programmer from the limitations of specific instruction ordering or alignment.
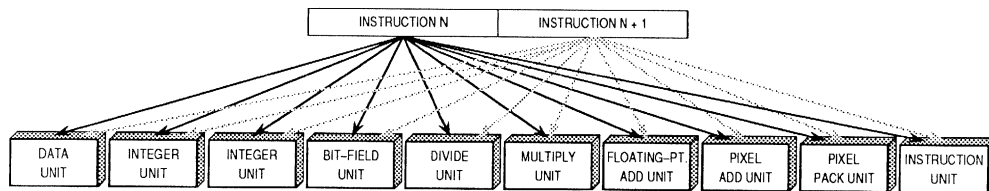
Figure 1-7. Symmetric Superscalar Instruction Issue

The execution unit pipelines are fully hardware interlocked via a scoreboard mechanism; therefore, data dependencies automatically delay instruction issue. The register scoreboard eliminates the need to schedule wasteful no operation (NOP) instructions into empty pipeline delay slots.

1.4.2.1 SOURCE AND DESTINATION DATA CONSIDERATIONS.

If an instruction attempts to use a source operand which is still being computed by a previous instruction, a data dependency exists. When a data dependency exists, instruction issue is stalled until all of the necessary source data is available. The MC88110 employs the register scoreboard as an efficient method for keeping track of when source data is available for an instruction.

The MC88110 implements several design features to reduce data-dependency overhead. The first feature, feed forwarding, allows the results from a previous instruction to be forwarded directly to a waiting instruction while simultaneously being written to the destination register. The second feature, branch prediction, reduces the delay caused by a data dependency for a branch instruction. In this case, the branch instruction is issued to a branch reservation station and instruction execution continues along the predicted path. Finally, a store instruction can be issued to the store reservation station even if source data is not yet available. For more information on these features, refer to **Section 9 Instruction Timing and Code Scheduling Considerations**.

Since the MC88110 allows instructions to complete out of order, there is the potential for an instruction's result to be overwritten by an instruction which issued earlier but completed later. To preclude this possibility, the scoreboard bit corresponding to the destination register is automatically checked as a condition for instruction issue. This ensures that updates to any given register are always completed in the order specified by the program and thus no data is ever incorrectly overwritten in the register files.

1.4.2.2 EXECUTION UNIT CONSIDERATIONS. For an instruction to be issued, the required execution unit must be available to begin execution of the instruction. The sequencer monitors the availability of all execution units and suspends instruction issue if the required execution unit is not available. An execution unit may not be available under the following circumstances:

1. A multi-cycle, nonpipelined unit can have only one instruction in execution at a time. Such a unit becomes busy when an instruction is issued to it, and it can not accept another instruction until the previous one completes. The divide unit is the only such unit on the MC88110.

2. An execution unit may become unavailable for additional instructions if its pipeline becomes full. This situation may occur if execution takes more clock cycles than the number of pipeline stages in the unit. This situation can only occur in the data unit. In addition, if the execution unit can not get access to a write-back slot while additional instructions continue to fill its pipeline, the pipeline may become full.

3. Execution units can accept only one instruction per clock. Issuing two instructions to the same unit on the same clock is prohibited.

Figure 1-8 illustrates which instruction pairs can and cannot be issued simultaneously due to the one instruction per execution unit per clock restriction. For example, if the first instruction in an issue pair is an integer arithmetic instruction, then the top row of the grid in Figure 1-8 shows that any type of instruction can be issued concurrently provided there are no data dependencies. On the other hand, if the first instruction in an issue pair were an integer multiply, then the fourth row of the grid in Figure 1-8 shows that another multiply (integer, graphics, or floating-point shown as the three white boxes on row four) cannot be issued concurrently. Note that the diagram is symmetric along the diagonal axis from the upper left to the lower right corner, indicating that this is a symmetric superscalar design. Note that Figure 1-8 is a condensed diagram which groups like instructions together. For a complete diagram listing each instruction, refer to **Section 9 Instruction Timing and Code Scheduling Considerations**.

arithmetic
logical
bit-field
multiply
divide
branch
load
store
fp multiply
fp add/sub/cmp
fp divide
graph. mult.
graph. add/sub
graph. pack

INTEGER — arithmetic
logical
bit-field
multiply
divide

FLOW CONTROL — branch

MEMORY — load
store

FLOATING POINT — fp multiply
fp add/sub/cmp
fp divide

GRAPHICS — graph. mult.
graph. add/sub
graph. pack

LEGEND:

CAN BE ISSUED SIMULTANEOUSLY (PROVIDED NO DATA DEPENDENCIES EXIST)

CANNOT BE ISSUED SIMULTANEOUSLY

Figure 1-8. Simultaneous Instruction Issue Restrictions

1.4.2.3 HISTORY BUFFER. Although the MC88110 issues instructions in strict sequential order, it is possible for instructions to complete execution out of order. The MC88110 keeps an internal FIFO queue of all instructions that are executing. This feature, the history buffer, keeps all details of out-of-order execution internal to the processor.

At the time of issue, an instruction is placed at the tail of the queue. The instructions move through the history buffer until they reach the head of the queue. An instruction reaches the head when all of the instructions in front of it have finished execution. However, since instructions can be executed out of order, it is possible for an instruction to have finished execution, but still be in the middle of the queue. An instruction is retired from the history buffer when it reaches the head and has finished execution.

The history buffer has 12 cells. If a multi-cycle instruction reaches the head of the buffer and takes a very long time to complete execution, it is possible to fill the history buffer to capacity. In this case, the MC88110 stalls instruction issue until the instruction at the head of the buffer completes execution and is retired from the queue.

1.5 INSTRUCTION SET SUMMARY

The MC88110 instruction set is divided into seven categories: integer arithmetic, floating-point arithmetic, graphics, logical, bit field, load/store/exchange, and flow control. The MC88110 instruction set is summarized in Figure 1-9.

Integer Arithmetic Instructions

Mnemonic	Description
add	Signed Add
addu	Unsigned Add
cmp	Integer Compare
divs	Signed Divide
divu	Unsigned Divide
muls	Signed Multiply
mulu	Unsigned Multiply
sub	Signed Subtract
subu	Unsigned Subtract

Bit-Field Instructions

Mnemonic	Description
clr	Clear Bit Field
ext	Extract Bit Field
extu	Unsigned Extract Bit Field
ff0	Find First Bit Clear
ff1	Find First Bit Set
mak	Make Bit Field
rot	Rotate Register
set	Set Bit Field

Logical Instructions

Mnemonic	Description
and	And
mask	Logical Mask Immediate
or	Or
xor	Exclusive Or

Graphics Instructions

Mnemonic	Description
padd	Pixel Add
padds	Pixel Add and Saturate
pcmp	Pixel Compare
pmul	Pixel Multiply
ppack	Pixel Truncate, Insert, and Pack
prot	Pixel Rotate Left
psub	Pixel Subtract
psubs	Pixel Subtract and Saturate
punpk	Pixel Unpack

Flow Control Instructions

Mnemonic	Description
bb0	Branch on Bit Clear
bb1	Branch on Bit Set
bcnd	Conditional Branch
br	Unconditional Branch
bsr	Branch to Subroutine
illop	Illegal Operation
jmp	Unconditional Jump
jsr	Jump to Subroutine
rte	Return from Exception
tb0	Trap on Bit Clear
tb1	Trap on Bit Set
tbnd	Trap on Bounds Check
tcnd	Conditional Trap

Load/Store/Exchange Instructions

Mnemonic	Description
ld	Load Register From Memory
lda	Load Address
ldcr	Load from Control Register
st	Store Register to Memory
stcr	Store to Control Register
xcr	Exchange Control Register
xmem	Exchange Register with Memory

Floating-Point Instructions

Mnemonic	Description
fadd	Floating-Point Add
fcmp	Floating-Point Compare
fcmpu	Unordered Floating-Point Compare
fcvt	Convert Floating-Point Precision
fdiv	Floating-Point Divide
fldcr	Load from Floating-Point Control Register
flt	Convert Integer to Floating-Point
fmul	Floating-Point Multiply
fsqrt	Floating-Point Square Root
fstcr	Store to Floating-Point Control Register
fsub	Floating-Point Subtract
fxcr	Exchange Floating-Point Control Register
int	Round Floating-Point to Integer
mov	Register-to-Register Move
nint	Round Floating-Point to Nearest Integer
trnc	Truncate Floating-Point to Integer

Figure 1-9. MC88110 Instruction Set

1

SECTION 2
PROGRAMMING MODEL

This section briefly describes the MC88110 processor states, registers and operand conventions. Exceptions are also briefly described in this section, but the details of individual exceptions (including exception recovery) are given in **Section 7 Exceptions**. Instruction mnemonics used in this section can be identified by referring to **Section 3 Addressing Modes and Instruction Set Summary**.

2.1 PROCESSOR STATES

The MC88110 is always in one of three states: normal instruction execution, exception, or reset. The reset state is entered when the $\overline{\text{RST}}$ signal is asserted. The exception state is entered when any of the following conditions occurs: external interrupts, memory access errors, internally recognized errors, or trap instructions. The following paragraphs describe the three states of the MC88110.

2.1.1 Reset State

When $\overline{\text{RST}}$ is recognized as asserted, all current processor operations are aborted, the control registers are initialized appropriately, and external signals are placed in the high-impedance state. When $\overline{\text{RST}}$ is negated the processor begins instruction execution at address zero.

2.1.2 Exception State

Exceptions are conditions that cause the processor to suspend execution of the current instruction stream and perform exception processing. Exception processing provides an efficient context switch so that system software can handle the exception condition while maintaining the integrity of the hardware and other software. Exception conditions include the following:

- External interrupts, signaled by the $\overline{\text{INT}}$ or $\overline{\text{NMI}}$ signals
- Memory access errors such as page faults and bus errors
- Internally recognized errors, such as divide-by-zero and arithmetic overflow
- Trap instructions
- Illegal instructions
- Privilege violations

When an exception is recognized by the processor, the execution context is saved into exception-time registers, the special function units are disabled, and the machine is placed in supervisor mode. Control is then passed to a designated exception handler routine. The exception handler routine processes the condition that caused the exception. The handler routine performs specific functions (e.g., fixing internal errors, aborting operations, or servicing interrupts) based on the type of exception that has occurred. The exception handler routine then restores the processor to normal operation.

The MC88110 implements a precise exception model. This means that the precise address of the faulting instruction is provided to the exception handler and that neither the faulting instruction nor any instructions logically following it in the code stream will appear to have been issued. Because of the precise exception model, it is not necessary for the internal pipeline states of the processor to be made visible to the software handlers.

Refer to **Section 7 Exceptions** for detailed information on exceptions.

2.1.3 Normal Instruction Execution State

During normal instruction execution, the MC88110 operates in one of two levels of privilege: supervisor mode or user mode. These levels define which address space is accessed and which registers are available to the programmer. The level of privilege is determined by the MODE bit in the processor status register (PSR). The following paragraphs describe the levels of privilege.

2.1.3.1 SUPERVISOR LEVEL OF PRIVILEGE. The supervisor mode is the higher level of privilege. The processor operates in this mode when the MODE bit is set. When operating in the supervisor mode, memory accesses reference the supervisor address space in data or instruction memory; however, the programmer can specify the **.usr** option for memory-access instructions to force access to user data address space. The supervisor mode allows execution of all instructions and allows access to all control registers and general registers.

Operating system software typically executes in supervisor mode. Among the operating system services provided are resource allocation (memory and peripherals), exception handling, and software execution control (task initiation, scheduling, etc.). Execution control normally includes controlling user programs and protecting the system from accidental or malicious corruption by a user program.

The MODE bit is set automatically when an exception is recognized so that the exception handler executes in supervisor mode. All bus transactions performed during exception processing reference supervisor address space. Reset also causes the MODE bit to be set, thus placing the processor in supervisor mode.

2.1.3.2 USER LEVEL OF PRIVILEGE. The processor operates in user mode when the MODE bit in the PSR is clear. Memory accesses in user mode can only reference user data and user instruction memory. Control register access is restricted in user mode. The only control registers accessible in this mode are the floating-point control and status registers. Attempting to access other control registers while in user mode causes an exception.

2.1.3.3 CHANGING LEVELS OF PRIVILEGE. The processor switches from user mode to supervisor mode under the following four conditions:

1. An exception occurs. Exceptions place the processor into the exception processing state, which includes switching to supervisor mode.

2. A reset is signaled.

3. A user program executes a trap instruction.

4. An interrupt or memory access fault occurs.

The processor switches from supervisor mode to user mode under the following two conditions:

1. The processor executes an **rte** instruction. The **rte** instruction restores the PSR, which returns the processor to user mode if the MODE bit of the restored PSR is clear.

2. A **stcr** or **xcr** instruction explicitly clears the MODE bit in the PSR. This method of clearing the MODE bit may cause the MC88110 to fetch the next few instructions from either supervisor or user space, and thus usually causes undesired program execution results.

2.2 REGISTER DESCRIPTION

The MC88110 contains three types of registers which provide data and execution information to the execution units and to software. Register access depends on the register type and current level of privilege. The following paragraphs describe programming model and the programmer's view of the general, extended, and control registers. Refer to **Section 4 Floating-Point Implementation** for more information on floating-point control registers and **Section 7 Exceptions** for more information on exception control registers.

2.2.1 Supervisor/User Programming Model

The supervisor programming model includes all general, extended, and control registers. The user programming model includes all general and extended registers, but only two of the control registers: the floating-point control register (FPCR) and floating-point status register (FPSR). Figure 2-1 illustrates the programming model.

The contents of the general control registers can be copied to and from the general registers using the **ldcr**, **stcr**, and **xcr** instructions. However, these instructions are privileged and therefore restrict access of the general control registers to supervisor mode software.

Figure 2-1. Programming Model

The diagram includes:

USER PROGRAMMING MODEL

GENERAL REGISTERS
- r0 — ZERO
- r1 — SUBROUTINE RETURN POINTER
- r2 ... r31 — TEMPORARY STORAGE REGISTERS

EXTENDED REGISTERS
- x0 — ZERO
- x1 ... x31 — TEMPORARY STORAGE REGISTERS

- fcr62 FPSR — FLOATING POINT STATUS REGISTER
- fcr63 FPCR — FLOATING POINT CONTROL REGISTER

SUPERVISOR PROGRAMMING MODEL

Register	Name	Description
cr0	PID	PROCESSOR IDENTIFICATION
cr1	PSR	PROCESSOR STATUS REGISTER
cr2	EPSR	EXCEPTION PROCESSOR STATUS REGISTER
cr4	EXIP	EXCEPTION EXECUTING INSTRUCTION POINTER
cr5	ENIP	EXCEPTION NEXT INSTRUCTION POINTER
cr7	VBR	VECTOR BASE REGISTER
cr16	SR0	STORAGE REGISTER 0
cr17	SR1	STORAGE REGISTER 1
cr18	SR2	STORAGE REGISTER 2
cr19	SR3	STORAGE REGISTER 3
cr20	SR4	STORAGE REGISTER 4
cr25	ICMD	INSTRUCTION MMU/CACHE/TIC COMMAND
cr26	ICTL	INSTRUCTION MMU/CACHE CONTROL
cr27	ISAR	INSTRUCTION SYSTEM ADDRESS
cr28	ISAP	INSTRUCTION MMU SUPERVISOR AREA POINTER
cr29	IUAP	INSTRUCTION MMU USER AREA POINTER
cr30	IIR	INSTRUCTION MMU ATC INDEX REGISTER
cr31	IBP	INSTRUCTION MMU BATC R/W PORT
cr32	IPPU	INSTRUCTION MMU PATC R/W PORT (UPPER)
cr33	IPPL	INSTRUCTION MMU PATC R/W PORT (LOWER)
cr34	ISR	INSTRUCTION ACCESS STATUS REGISTER
cr35	ILAR	INSTRUCTION ACCESS LOGICAL ADDRESS
cr36	IPAR	INSTRUCTION ACCESS PHYSICAL ADDRESS
cr40	DCMD	DATA MMU/CACHE COMMAND
cr41	DCTL	DATA MMU/CACHE CONTROL
cr42	DSAR	DATA SYSTEM ADDRESS REGISTER
cr43	DSAP	DATA MMU SUPERVISOR AREA POINTER
cr44	DUAP	DATA MMU USER AREA POINTER
cr45	DIR	DATA MMU ATC INDEX REGISTER
cr46	DBP	DATA MMU BATC R/W PORT
cr47	DPPU	DATA MMU PATC R/W PORT (UPPER)
cr48	DPPL	DATA MMU PATC R/W PORT (LOWER)
cr49	DSR	DATA ACCESS STATUS REGISTER
cr50	DLAR	DATA ACCESS LOGICAL ADDRESS
cr51	DPAR	DATA ACCESS PHYSICAL ADDRESS
fcr0	FPECR	FLOATING-POINT EXCEPTION CAUSE REGISTER

The contents of the floating-point control registers can be copied to and from the general registers using the **fldcr**, **fstcr**, and **fxcr** instructions. These instructions allow access of **fcr63** and **fcr62** to user mode software but restrict access of **fcr0** to supervisor mode software. Refer to **Section 4 Floating-Point Implementation** for a detailed description of the floating-point control registers.

2.2.2 General Register File

The general register file (GRF) consists of 32 general registers, each of which is 32 bits wide (see Figure 2-2). These registers can contain instruction operands and results and can provide address and bit-field information. All general registers have read/write access.

```
        31                                                     0
  r0   ┌────────────────────────────────────────────────────┐
       │                  CONTAINS ZERO                      │
  r1   ├────────────────────────────────────────────────────┤
       │             SUBROUTINE RETURN POINTER               │
  r2   ├────────────────────────────────────────────────────┤
       │                                                      │
       │                                                      │
       │            TEMPORARY STORAGE REGISTERS               │
       │                                                      │
       │                                                      │
  r31  └────────────────────────────────────────────────────┘
```
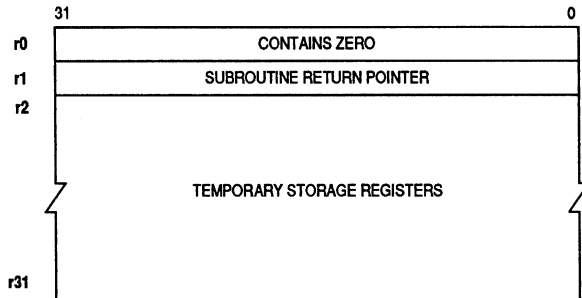
Figure 2-2. General Register File

There are two hardware restrictions on the use of certain general registers, which are as follows:

1. Register **r0**—This register always contains the constant zero and is always read as positive zero. Register **r0** can be used by instructions requiring the constant zero as an operand (e.g., compare to zero). Writing to **r0** is permissible but causes no modification to the contents of the register and, depending on the implementation, may or may not cause normal instruction side effects.

2. Register **r1**—The return pointer generated by the **bsr** and **jsr** instructions is stored in this register each time either of these instructions execute. Register **r1** is not protected; therefore, the return pointer (or any other data) contained in **r1** can be read or overwritten by software.

2.2.3 Extended Register File

The extended register file (XRF) consists of 32 extended registers, each of which is 80 bits wide. These registers can contain 32 data objects of any of the defined floating-point data formats: single, double, or double-extended precision. The extended registers can provide operands for all floating-point instructions and can serve as the data source or destination for **st** and **ld** instructions. See Figure 2-3 for an illustration of the XRF.

The extended register file has only one hardware restriction, which is on register **x0**. This register always contains the constant zero and is always read as positive zero. Register **x0** can be used by instructions requiring the constant zero as an operand. Writing to **x0** is permissible but causes no modification to the contents of the register, and, depending on the implementation, may or may not cause normal instruction side effects.
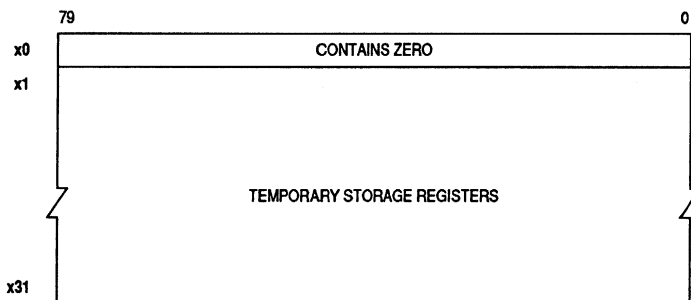
Figure 2-3. Extended Register File

2.2.4 Control Registers

The following paragraphs describe the general control registers and the floating-point control registers.

2.2.4.1 GENERAL CONTROL REGISTERS. The MC88110 contains 35 general control registers (see Table 2-1). These registers are accessible only in supervisor mode. Twelve of the general control registers are for the instruction cache and memory management unit (MMU), and twelve are for the data cache and MMU. The remaining registers provide status information, the base address of the exception vector table, and general-purpose storage.

The general control registers can be read using the **ldcr** instruction, and can be written using the **stcr** instruction. The **xcr** instruction exchanges the contents of a control register with the contents of the specified general register. When a control register is read, reserved bits are returned as zeros. Writes to reserved bits are ignored. When a read or write to an unimplemented control register is attempted, an unimplemented opcode exception is taken. When a register specified as "Motorola internal use only" is read, undefined data is returned. Writes to these registers will not cause an exception; however, subsequent reads are not guaranteed to return the previously written data.

The following paragraphs describe the processor identification (PID) register, the PSR, and the supervisor storage registers. Refer to **Section 4 Floating-Point Implementation**, **Section 6 Instruction and Data Caches**, **Section 7 Exceptions**, and **Section 8 Memory Management Units** for more detailed information on the other control registers.

Table 2-1. General Control Registers

Register Number	Acronym	Register Name
cr0	PID	Processor Identification Register
cr1	PSR	Processor Status Register
cr2	EPSR	Exception Processor Status Register
cr3	—	Unimplemented
cr4	EXIP	Exception Executing Instruction Pointer

Table 2-1. General Control Registers (Continued)

Register Number	Acronym	Register Name
cr5	ENIP	Exception Next Instruction Pointer
cr6	—	Unimplemented
cr7	VBR	Vector Base Register
c8–cr13	—	Unimplemented
cr14–cr15	—	Motorola Internal Use Only
cr16	SR0	Storage Register 0
cr17	SR1	Storage Register 1
cr18	SR2	Storage Register 2
cr19	SR3	Storage Register 3
cr20	SR4	Storage Register 4
cr21–cr24	—	Unimplemented
cr25	ICMD	Instruction MMU/Cache/TIC Command
cr26	ICTL	Instruction MMU/Cache Control
cr27	ISAR	Instruction System Address
cr28	ISAP	Instruction MMU Supervisor Area Pointer
cr29	IUAP	Instruction MMU User Area Pointer
cr30	IIR	Instruction MMU ATC Index Register
cr31	IBP	Instruction MMU BATC R/$\overline{\text{W}}$ Port
cr32	IPPU	Instruction MMU PATC R/$\overline{\text{W}}$ Port (Upper)
cr33	IPPL	Instruction MMU PATC R/$\overline{\text{W}}$ Port (Lower)
cr34	ISR	Instruction Access Status Register
cr35	ILAR	Instruction Access Logical Address
cr36	IPAR	Instruction Access Physical Address
cr37–cr39	—	Unimplemented
cr40	DCMD	Data MMU/Cache Command
cr41	DCTL	Data MMU/Cache Control
cr42	DSAR	Data System Address
cr43	DSAP	Data MMU Supervisor Area Pointer
cr44	DUAP	Data MMU User Area Pointer
cr45	DIR	Data MMU ATC Index Register
cr46	DBP	Data MMU BATC R/$\overline{\text{W}}$ Port
cr47	DPPU	Data MMU PATC R/$\overline{\text{W}}$ Port (Upper)
cr48	DPPL	Data MMU PATC R/$\overline{\text{W}}$ Port (Lower)
cr49	DSR	Data Access Status Register
cr50	DLAR	Data Access Logical Address
cr51	DPAR	Data Access Physical Address
cr52–cr63	—	Unimplemented

2.2.4.1.1 Processor Identification Register. The PID (**cr0**) contains the processor version number. This register is read only. The PID is shown in Figure 2-4.

31															16	15	8	7	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	ARCHITECTURAL REVISION		VERSION NUMBER		1
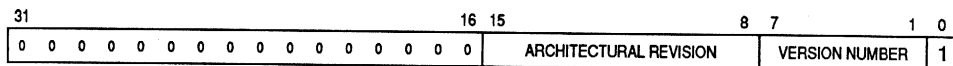
Figure 2-4. Processor Identification Register

Bits 31–16—Reserved

   Read as zero; not guaranteed to be zero in future implementations.

ARCH REVISION—Architectural Revision Number

   Identifies the particular processor (MC88100, MC88110, future generations, special purpose processors). The revision number changes when a major architectural change is made that warrants a new part number. The revision number for the MC88110 is 1.

VERSION #—Version Number

   Identifies the particular mask version of the MC88110 processor. The version number is changed by Motorola when mask changes are made that affect the functionality of the device.

Bit 0—Reserved

   Read as one; not guaranteed to be one in future implementations.

2.2.4.1.2 Processor Status Register. The bits in the PSR (**cr1**) are set by hardware or software to report the status of processor operations or to control processor operation. The operation of the various bits in the PSR depends on the value of the shadow freeze bit (EFRZ, bit 0) in the PSR. For a detailed explanation of the implications and effects of the EFRZ, refer to **Section 7 Exceptions**. In the paragraphs that follow, an asterisk (*) denotes the default state after reset. Figure 2-5 shows the PSR.

31	30	29	28	27	26	25	24		10	9		3	2	1	0
MODE	BO	SER	C		SGN	SRM					SFUD		MXM	IND	EFRZ

UNDEFINED-RESERVED FOR FUTURE USE
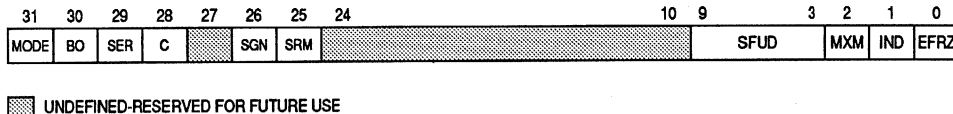
Figure 2-5. Processor Status Register

Mode—Supervisor/User Mode

This bit is set by hardware when the processor changes to supervisor mode due to an exception condition or trap instruction. The mode bit may be cleared by software to return the MC88110 to user mode.

0 = User Mode
1 = Supervisor Mode*

BO—Byte Ordering

This bit is set by software to indicate the current byte ordering. See **2.3.4.2 Byte Ordering** for a full description of byte ordering.

0 = Big Endian Byte Ordering*
1 = Little Endian Byte Ordering

SER—Serial Mode

The serial mode is generally used for debugging purposes since it significantly reduces machine throughput. This bit is set by software.

0 = Concurrent Instruction Execution
1 = Serial Instruction Execution*

C—Carry

This bit is modified by hardware according to the results of and add or subtract instruction. It is only modified when the instruction explicitly requests the use of the carry bit.

0 = Carry Not Generated by an Add or Subtract Instruction*
1 = Carry Generated by an Add or Subtract Instruction

Bit 27—Reserved

Read as zero; not guaranteed in future implementations. Writes are ignored.

SGN—Signed Immediate Mode

This bit is set by software to determine whether immediate offsets and constants are signed or unsigned.

0 = Immediate Offsets and Constants are Unsigned*
1 = Immediate Offsets and Constants are Signed Two's Complement

SRM—Serialize Memory

This bit is set by software to force serialization of the processor prior to load or store instruction execution.

0 = Concurrent Memory Instruction Execution
1 = Serialize Memory Instructions*

Bit 24-10—Reserved

Read as zero; not guaranteed in future implementations. Writes are ignored.

Bits 9-5—Special Function Unit Disable

These bits will be used to enable additional SFUs in future 88000 implementations. These bits are hardwired to "one" in the MC88110.

1 = Unimplemented SFUs Always Disabled

SFD2—Special Function Unit Two (SFU2) Disable

This bit disables SFU2, the graphics unit. This bit is automatically set by hardware when an exception or reset occurs, and can also be set or cleared explicitly by software.

0 = SFU2 Enabled
1 = SFU2 Disabled*

SFD1—Special Function Unit One (SFU1) Disable

This bit disables SFU1, the floating point unit. This bit is automatically set by hardware when an exception or reset occurs, and can also be set or cleared explicitly by software.

0 = SFU1 Enabled
1 = SFU1 Disabled*

MXM—Misaligned Access Exception Mask

This bit is set by software to disable the misaligned access exception. When this bit is set and a misaligned access is attempted, the processor truncates the address to a consistent boundary (see **2.3.4.1 Misaligned Access**).

0 = Misaligned Access Exception Mode Enabled
1 = Misaligned Access Exception Mode Disabled*

IND—Interrupt Disable

This bit is automatically set by hardware to disable interrupts when an exception occurs. This bit can also be set or cleared explicitly by software to specifically disable/enable interrupts. Interrupts must be disabled when shadowing is frozen to avoid an error exception.

0 = External Interrupt Enabled
1 = External Interrupt Disabled*

EFRZ—Exceptions Freeze

This bit is set by hardware when an exception occurs to preserve the processor context for the exception. This bit can also be set or cleared explicitly by the **stcr** or **xcr** instructions or implicitly by an **rte** instruction. If this bit is set and any exception occurs, the MC88110 takes the error exception. Setting the EFRZ bit in the PSR with an **stcr** or **xcr** instruction does not cause the EFRZ bit to be set in the EPSR.

0 = Exceptions Enabled
1 = Exceptions Disabled*

2.2.4.1.3 Supervisor Storage Registers. The integer unit contains five 32-bit supervisor storage registers which have read/write access. These registers provide high-speed storage space where supervisor software can store data and pointers without referencing memory. The use and content of these registers are determined by software.

2.2.4.2 FLOATING-POINT CONTROL REGISTERS. The floating-point control registers provide exception recovery and status and control information for the floating-point unit (FPU). Table 2-2 lists the floating-point control registers. Refer to **Section 4 Floating-Point Implementation** for detailed descriptions of these registers.

Table 2-2. Floating-Point Control Registers

Number	Acronym	Register Name
fcr0	FPECR	Floating-Point Exception Cause Register
fcr1–fcr61	—	Unimplemented
fcr62	FPSR	Floating-Point Status Register
fcr63	FPCR	Floating-Point Control Register

2.3 OPERAND CONVENTIONS

The following paragraphs describe the operand conventions for the MC88110, including a definition of the operand types and a description of how operands are organized in registers and in memory.

2.3.1 Operand Types

The MC88110 supports the following operand types:

Integer Operands:

Byte—8 Bits

Half Word—16 Bits

Word—32 Bits

Double Word—64 Bits

Bit-Field Operands:

Bit Field—1 to 32 Bits in a 32-Bit Register

Floating-Point Operands:

Single-Precision Floating-Point—32 Bits

Double-Precision Floating-Point—64 Bits

Double-Extended-Precision Floating-Point—80 Bits

Graphics Operands:

32-Bit Packed Nibbles

32-Bit Packed Bytes

64-Bit Packed Bytes

64-Bit Packed Half-Words

64-Bit Packed Half-Words

64-Bit Packed Words

The operand size for each instruction is either explicitly encoded in the instruction or implicitly defined by the instruction operation. Bit fields are defined by width and offset values given in the instruction or in a source register specified by the instruction. For more information on floating-point and graphics operands, refer to **Section 4 Floating-Point Implementation** and **Section 5 Graphics Unit Implementation,** respectively.

2.3.2 Data Organization in General Registers

The GRF can contain all types of operands except double-extended-precision floating-point operands. Graphics operand sizes range from 8 to 32 bits. These operands are packed into double words (64 bits), which are contained in a register pair.

Since the memory interface supports operand types other than 32-bit words, the MC88110 incorporates the following rules for placing memory data into registers or extracting data from registers for storing to memory (see Figure 2-6):

1. Byte operands are always contained in the lowest eight bits of a register. When a byte is loaded into a register, it is either sign- or zero-extended to 32 bits.

2. Half-word operands are always contained in the lowest 16 bits of a register. When a half word is loaded into a register, it is either sign- or zero-extended to 32 bits.

3. Word operands are contained in the entire 32 bits of a register.

4. Double-word operands are loaded to or extracted from two adjacent registers (rn and rn+1), with rn always even and always containing the higher order word.

5. Bit-field operands are defined by an offset and a width. The most significant bit (MSB) of a bit field is the bit closest to bit 31; the least significant bit (LSB) is the bit closest to bit 0. The value of the offset equals the bit number of the LSB of the bit field, and [offset + width −1] equals the bit number of the MSB of the bit field.

6. Single-precision floating-point operands are contained in the entire 32 bits of a register. Bit 31 contains the sign bit, bits 30-23 contain the exponent, and the remaining bits comprise the mantissa.

7. Double-precision floating-point operands are contained in two adjacent registers (rn and rn+1), with rn always even and always containing the higher order bits. In the upper order register (rn), bit 31 contains the sign bit, bits 30−20 contain the exponent, and bits 19−0 contain the upper bits of the mantissa. Bits 31−0 of the lower order register (rn+1) contain the lower bits of the mantissa.

Any double-word and double-precision floating-point operands aligned on odd-numbered register pairs (i.e., **r5:r6** instead of **r4:r5**) will cause the following exceptions to occur:

1. Floating-point instructions referencing an odd-numbered register pair will cause an SFU1 floating-point unimplemented exception.

2. Graphics instructions referencing an odd-numbered register pair will cause an SFU2 exception.

3. All other instructions referencing an odd-numbered register pair will cause an unimplemented opcode exception.

The exception handler will implement double-word alignment on odd-numbered register pairs in software. Since the software implementation will result in slower execution time, it is recommended that software and compilers align such data to even-numbered registers to guarantee the best performance.

INTEGERS:

SIGNED BYTE

```
31                                      8 7        0
S S S S S S S S S S S S S S S S S S S S S S S S |  BYTE
```

UNSIGNED BYTE

```
31                                      8 7        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 |  BYTE
```

SIGNED HALF WORD

```
31                      16 15                    0
S S S S S S S S S S S S S S S S |    HALF WORD
```

UNSIGNED HALF WORD

```
31                      16 15                    0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |    HALF WORD
```

WORD

```
31                                              0
|                    WORD                        |
```

DOUBLE WORD

```
        63                                      32
N      | WORD 0 (MOST SIGNIFICANT WORD; REGISTER N)    |
N + 1  | WORD 1 (LEAST SIGNIFICANT WORD; REGISTER N + 1) |
```

BIT FIELDS:

BIT FIELD

```
                        BIT NUMBER =    BIT NUMBER =
                        OFFSET + WIDTH   OFFSET
31                                              0
|                       | BIT FIELD  |           |
                        |← WIDTH →|← OFFSET →|
```

FLOATING-POINT OPERANDS:

SINGLE-PRECISION FLOATING POINT

```
31        24 23                                 0
| S | EXPONENT |           MANTISSA              |
```

DOUBLE-PRECISION FLOATING POINT

```
        63 62        52 51              -       32
N      | S | EXPONENT  | HIGH-ORDER BITS OF MANTISSA |
N + 1  |        LOW-ORDER BITS OF MANTISSA           |
        31                                      0
```

GRAPHICS OPERANDS:

32-BIT PACKED NIBBLES

```
31  28 27  24 23  20 19  16 15  12 11  8 7  4 3  0
|NIBBLE|NIBBLE|NIBBLE|NIBBLE|NIBBLE|NIBBLE|NIBBLE|NIBBLE|
```

32-BIT PACKED BYTES

```
31      24 23      16 15      8 7        0
|  BYTE  |  BYTE  |  BYTE  |  BYTE  |
```

64-BIT PACKED BYTES

```
63      56 55      48 47      40 39       32
|  BYTE  |  BYTE  |  BYTE  |  BYTE  |
31      24 23      16 15      8 7        0
|  BYTE  |  BYTE  |  BYTE  |  BYTE  |
```

32-BIT PACKED HALF-WORDS

```
31              16 15              0
|   HALF-WORD    |   HALF-WORD    |
```

64-BIT PACKED HALF-WORDS

```
63              48 47              32
|   HALF-WORD    |   HALF-WORD    |
31              16 15              0
|   HALF-WORD    |   HALF-WORD    |
```

64-BIT PACKED WORDS

```
63                                32
|              WORD               |
31                                0
|              WORD               |
```

Figure 2-6. Data Organization in General Registers

2.3.3 Data Organization in Extended Registers

The XRF can contain all types of floating-point operands (see Figure 2-7). When data is placed in an extended register, the value given to unused bits is not defined; for example, if single-precision data is placed in an 80-bit extended register, then all 80 bits are overwritten, but the value of the least significant 48 bits is undefined.
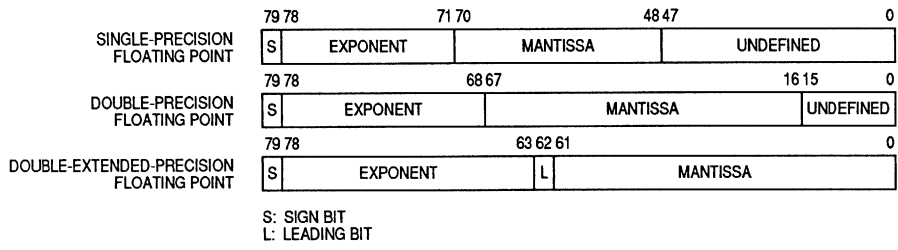
Figure 2-7. Operands in Extended Register File

2.3.4 Data Organization in Memory and Data Transfers

Data transfers are required to be aligned to the appropriately sized boundary in memory (i.e., byte, half word, or word), and bit fields are represented in memory as part of bytes, half words, and words.

Single-precision floating-point values are stored in memory on word boundaries; double-precision values are stored on even-word boundaries. Double-extended-precision values are stored on quad-word boundaries with all data left justified and all extra space filled with zeros when writing and ignored when reading to ensure that data stored in memory will be compatible with future implementations. Figure 2-8 illustrates how data is organized in memory.
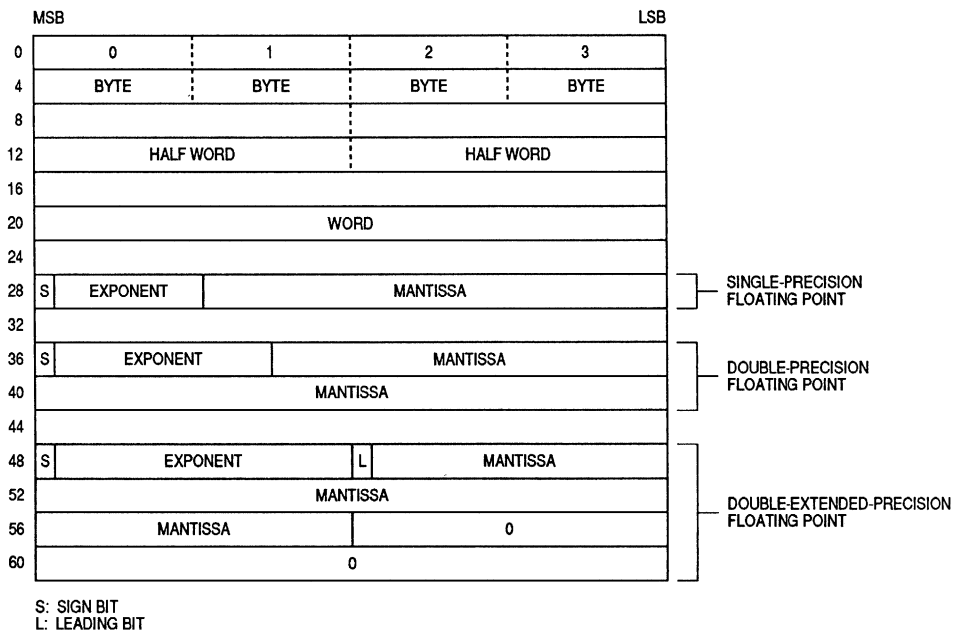
Figure 2-8. Floating-Point Memory Storage Alignment

2.3.4.1 MISALIGNED ACCESS. Attempting an incorrectly aligned data transfer will cause a misaligned reference exception if the misaligned access exception is not masked. If the Exceptions:misaligned access exception is masked and a misaligned transfer is attempted, the least significant bits of the address will be ignored, and the transfer will be performed to the next lower aligned boundary.

Figure 2-9 shows the results of several memory accesses with the misaligned access exception masked. In this illustration, the lightly shaded accesses are correctly aligned, and the darkly shaded accesses are not correctly aligned. The shaded areas in memory (light and dark) show the resulting alignment for each access. Note that in each case the arrows point to the specified memory location.
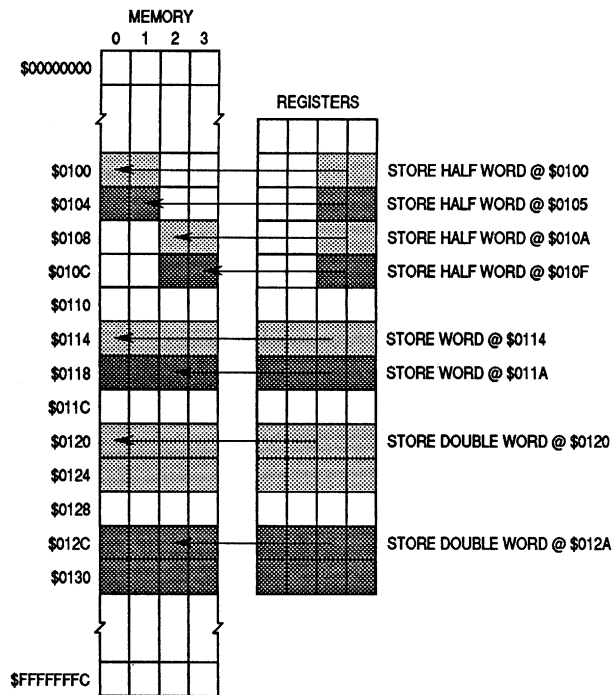
Figure 2-9. Memory Accesses with Misaligned Access Exceptions Disabled

2.3.4.2 BYTE ORDERING. The 88000 base architecture supports two byte-ordering configurations for data operands in memory: the Big-Endian configuration and the Little-Endian configuration. The processor defaults to Big-Endian byte ordering. Additionally, instruction addressing is performed in the Big-Endian configuration regardless of the byte ordering mode of the processor. Figure 2-10 illustrates the Big-Endian and Little-Endian byte-ordering memory configurations.

Note that when a Big-Endian memory system is drawn, the lowest addresses are depicted at the top of the memory system, with the addresses increasing toward the

bottom of the page. In the Big-Endian configuration, lower addresses correspond to more significant bytes, and the address of a word points to the most significant byte of the word.

When a Little-Endian memory system is drawn, the lowest address is depicted at the bottom of the page, with the addresses increasing toward the top of the page. In the Little-Endian configuration, lower addresses correspond to less significant bytes, and the address of a word points to the least significant byte of the word.
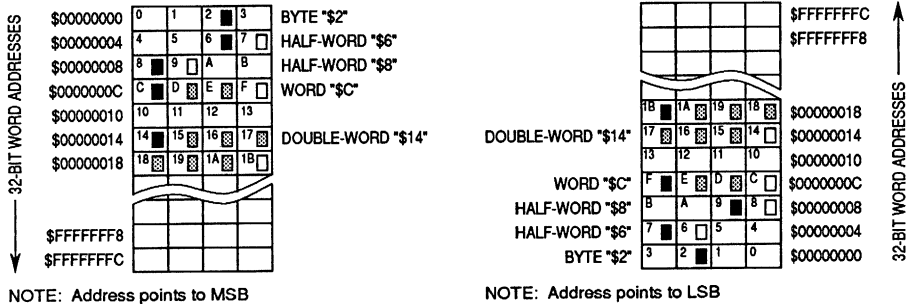
Figure 2-10. Byte-Ordering Configuration in Memory

The example byte ordering environment shown in Figure 2-11 illustrates how to interface a Little-Endian device with an MC88110 based system using a Big-Endian memory configuration.

In Figure 2-11, latches are used to transfer the data from the Little-Endian processor to the correct byte lane of the 64-bit bus. A similar circuit is used inside the MC88110, to align the bytes from the bus and write the correct data in the destination register. When the MC88110 is in Little-Endian mode, a byte-swap correction circuit is enabled which reverses the order of the bytes before they are aligned to be written to the registers.
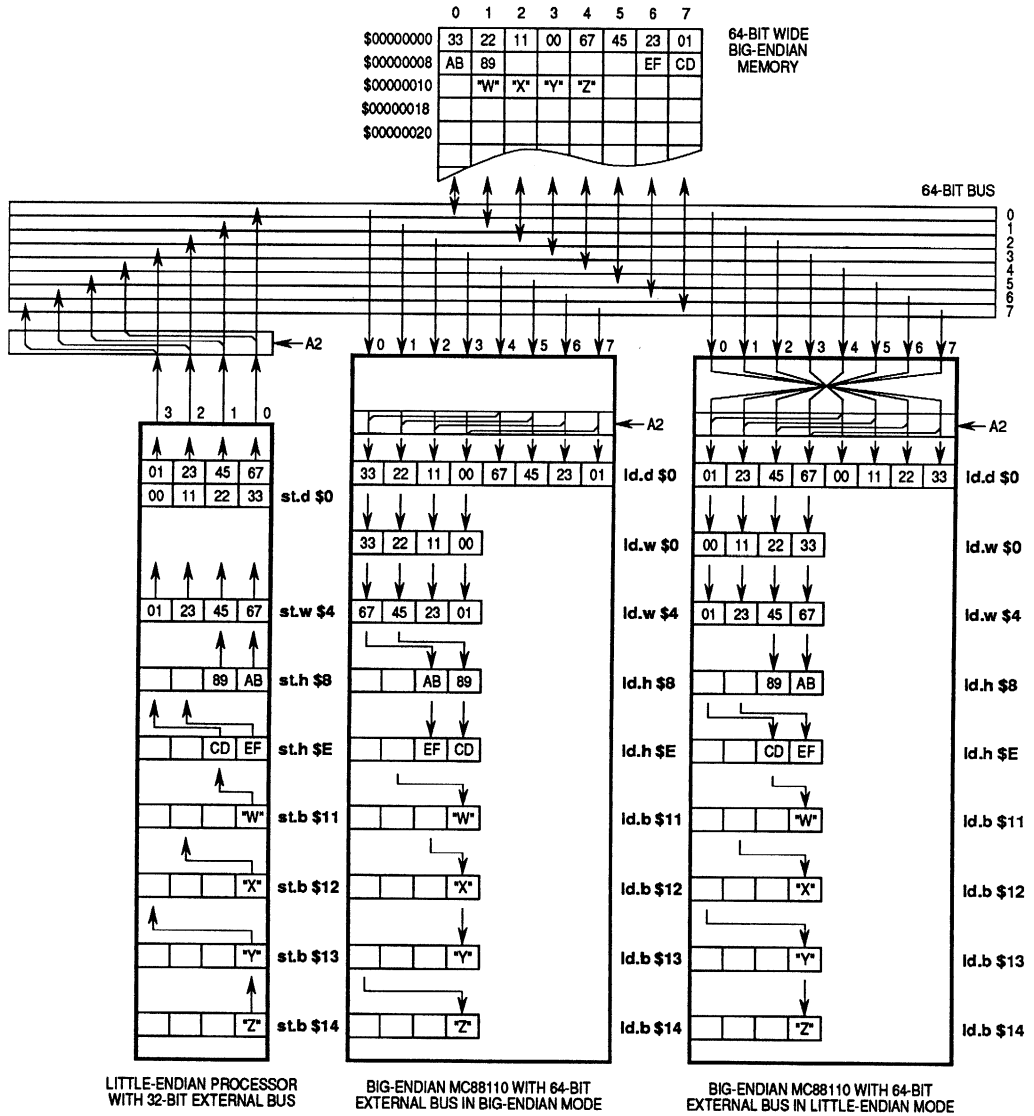
Figure 2-11. Example Byte-Ordering Environment Using Big-Endian Memory and 64-Bit Bus

Figure 2-11 shows the results of a Little-Endian processor writing a series of data to a Big-Endian Memory, and the MC88110 reading in the same data in both Big Endian and Little-Endian modes. When the Little-Endian processor stores a double-word (0123456700112233) to memory at address $0, the data is stored with the least significant byte at memory location $0, and the most significant byte at memory location $7. When the double-word at memory address $0 is read in by the MC88110 in Big-Endian mode, the data in byte $0 is considered to be the most significant byte, the data in byte $7 is considered to be the least significant byte, and the entire double-word is loaded into the register backwards. However, if the MC88110 is placed in Little-Endian mode, a byte-swap correction is applied to the data as it is read into the registers, and the correct integer is loaded into the registers.

Figure 2-11 also illustrates the storage of words, half-words, and bytes. Notice that whenever the data being transferred is more than one byte wide, the data read by the Big-Endian processor is backwards when compared to the data written to memory from the Little-Endian processor. In each of these cases, however, the problem is solved by placing the Big-Endian processor in Little-Endian mode. Also notice that the data being transferred on the bus is the same, regardless of the size of the transfer and regardless of the byte ordering mode.

Figure 2-12 shows how the previous example is affected by replacing the Big-Endian memory system with a Little-Endian memory system and replacing the 64-bit bus with a 32-bit bus. In this case, external circuitry is required to interface the 64-bit bus of the MC88110 with the 32-bit bus. Note that the configuration of the actual memory system has no effect on the operation of the processors.
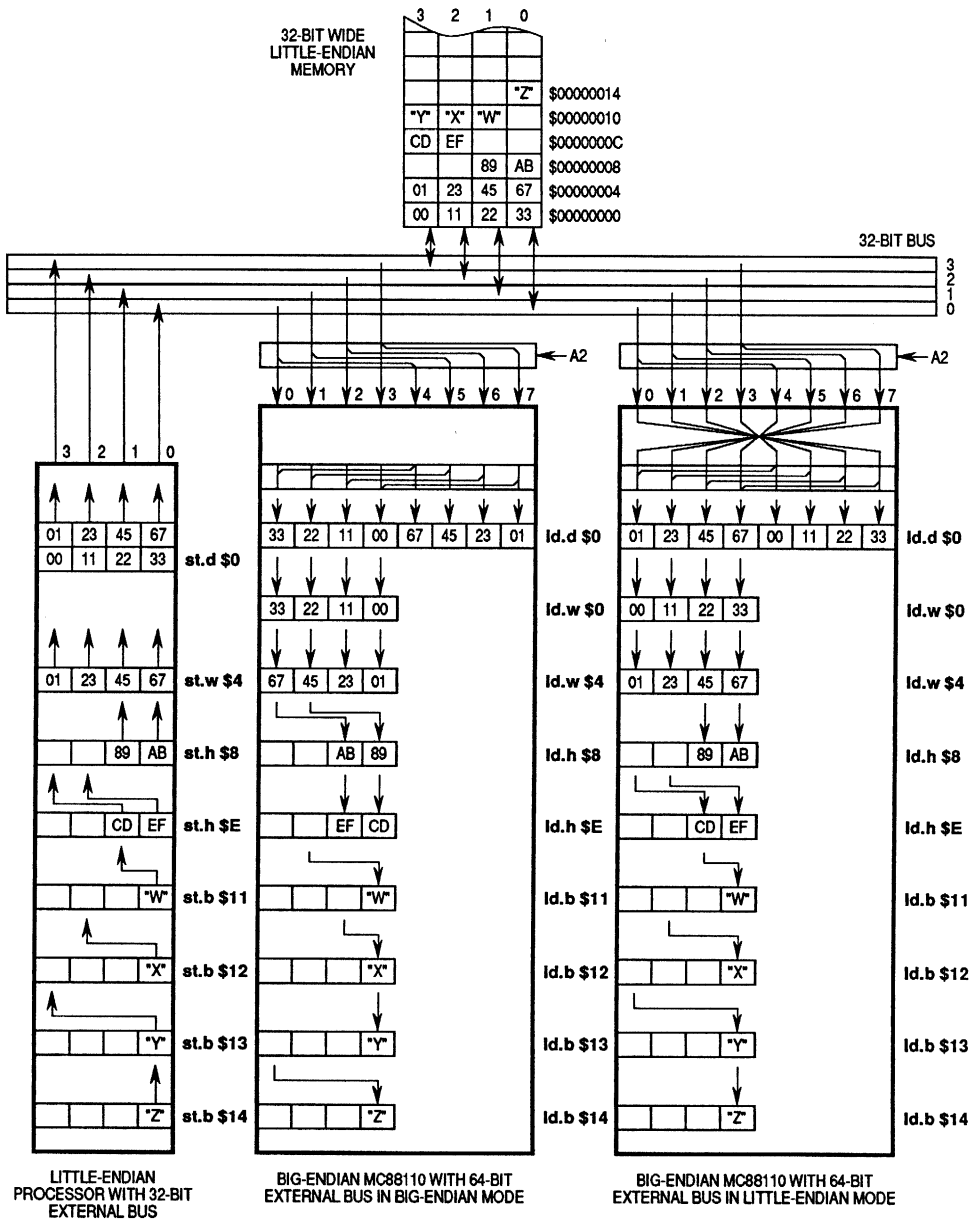
Figure 2-12. Example Byte-Ordering Environment Using Little-Endian Memory and 32-Bit Bus

SECTION 3
ADDRESSING MODES AND INSTRUCTION SET SUMMARY

3

This section describes the addressing modes available in the MC88110 and gives a summary of the instruction set. For complete instruction descriptions, including the exceptions caused by each instruction, refer to **Section 10 Instruction Set**.

The MC88110 instruction set is divided into seven categories, as shown in Figure 3-1: integer arithmetic, logical, bit-field, floating-point, graphics, flow control, and load/store/exchange instructions. The MC88110 addressing modes are defined in terms of three types of instructions: computational, load/store/exchange, and flow control instructions. Computational instructions include the integer arithmetic, logical, bit-field, floating-point, and graphics instructions.

Integer Arithmetic Instructions

Mnemonic	Description
add	Signed Add
addu	Unsigned Add
cmp	Integer Compare
divs	Signed Divide
divu	Unsigned Divide
muls	Signed Multiply
mulu	Unsigned Multiply
sub	Signed Subtract
subu	Unsigned Subtract

Bit-Field Instructions

Mnemonic	Description
clr	Clear Bit Field
ext	Extract Bit Field
extu	Unsigned Extract Bit Field
ff0	Find First Bit Clear
ff1	Find First Bit Set
mak	Make Bit Field
rot	Rotate Register
set	Set Bit Field

Logical Instructions

Mnemonic	Description
and	And
mask	Logical Mask Immediate
or	Or
xor	Exclusive Or

Graphics Instructions

Mnemonic	Description
padd	Pixel Add
padds	Pixel Add and Saturate
pcmp	Pixel Compare
pmul	Pixel Multiply
ppack	Pixel Truncate, Insert, and Pack
prot	Pixel Rotate Left
psub	Pixel Subtract
psubs	Pixel Subtract and Saturate
punpk	Pixel Unpack

Flow Control Instructions

Mnemonic	Description
bb0	Branch on Bit Clear
bb1	Branch on Bit Set
bcnd	Conditional Branch
br	Unconditional Branch
bsr	Branch to Subroutine
illop	Illegal Operation
jmp	Unconditional Jump
jsr	Jump to Subroutine
rte	Return from Exception
tb0	Trap on Bit Clear
tb1	Trap on Bit Set
tbnd	Trap on Bounds Check
tcnd	Conditional Trap

Load/Store/Exchange Instructions

Mnemonic	Description
ld	Load Register From Memory
lda	Load Address
ldcr	Load from Control Register
st	Store Register to Memory
stcr	Store to Control Register
xcr	Exchange Control Register
xmem	Exchange Register with Memory

Floating-Point Instructions

Mnemonic	Description
fadd	Floating-Point Add
fcmp	Floating-Point Compare
fcmpu	Unordered Floating-Point Compare
fcvt	Convert Floating-Point Precision
fdiv	Floating-Point Divide
fldcr	Load from Floating-Point Control Register
flt	Convert Integer to Floating-Point
fmul	Floating-Point Multiply
fsqrt	Floating-Point Square Root
fstcr	Store to Floating-Point Control Register
fsub	Floating-Point Subtract
fxcr	Exchange Floating-Point Control Register
int	Round Floating-Point to Integer
mov	Register-to-Register Move
nint	Round Floating-Point to Nearest Integer
trnc	Truncate Floating-Point to Integer

Figure 3-1. MC88110 Instruction Set

3.1 ADDRESSING MODES

The MC88110 addressing modes are defined in terms of three types of instructions: computational, load/store/exchange, and flow control instructions. The computational instructions manipulate data stored in the general or extended registers. The load/store/exchange instructions can load data into the general and extended registers, store data to memory, exchange a memory location with a general or extended register, or compute effective addresses. Flow control instructions alter the sequential flow of instructions through the processor.

Each instruction type has unique addressing capabilities. Computational instructions access data in the general-purpose registers, the extended registers, or in certain cases, the control registers. Load/store/exchange instructions use the data unit to access data in main memory. Flow control instructions use the instruction unit to reference instructions in main memory.

The following paragraphs describe the addressing modes and instruction formats available for the MC88110.

3.1.1 Computational Addressing Modes

The MC88110 supports three types of addressing modes for computational instructions: triadic register, immediate, and control register addressing. These addressing modes are described in the following paragraphs.

3.1.1.1 TRIADIC REGISTER ADDRESSING. Triadic register addressing uses three 5-bit fields encoded in the instruction word to specify two source registers (rS1 and rS2) and a destination register (rD). This addressing mode is common to all computational instructions, but some instructions do not use all three register selection fields. All bits in unused fields must be zero for upward compatibility. The following paragraphs explain triadic register addressing for ALU instructions, floating-point instructions, and graphics instructions.

3.1.1.1.1 ALU Instructions. The ALU instructions consist of the integer arithmetic, logical, and bit-field instructions. For the integer arithmetic and logical instructions, the data in rS1 and rS2 is processed by an integer unit and the result is placed in rD. The arithmetic and logical instructions are **add, addu, and, cmp, divs, divu, muls, mulu, or, sub, subu**, and **xor**.

All bit-field instructions except the bit-scan instructions (**ff1** and **ff0**) use the triadic register addressing mode by designating a bit-field operand in rS1. This operand is defined by two 5-bit values contained in the lower 10 bits of rS2: the 5-bit value contained in bits 9–5 of rS2 specifies the width of the bit field, and the 5-bit value contained in bits 4–0 of rS2 specifies the offset of the bit field from bit 0 of rS1. The upper 22 bits of rS2 are ignored. For the **rot** instruction, bits 9–5 are also ignored, but they must be zero to ensure upward compatibility. The bit-field operand in rS1 is processed by the integer unit according to the specified instruction and the result is placed in rD. The bit-field instructions are **clr, ext, extu, mak, rot**, and **set**. The width

and offset values for bit-field instructions can also be specified as immediate operands, as described in **3.1.1.2.2 Register with 10-Bit Immediate Addressing**.

For bit-scan instructions (**ff1** and **ff0**), the operand in **rS2** is searched by the integer unit to find either the first bit set (**ff1**) or the first bit clear (**ff0**). The register is scanned from most significant bit (bit 31) to least significant bit (bit 0). The result is returned to **rD**. The S1 field is not used and must contain zeros.

```
                                         31                        0
                             rS1        ┌──────────────────────────┐
                                        │    SOURCE 1 REGISTER      │
                                        └──────────────────────────┘
                   31                    0
      rS2         ┌──────────────────────┐
                  │   SOURCE 2 REGISTER   │
                  └──────────────────────┘
                                        ┌──────────────────────────┐
                                        │ INTEGER ARITHMETIC OPERATION │
                                        │   BIT-FIELD OPERATION     │
                                        │   LOGICAL OPERATION       │
                                        └──────────────────────────┘
                                         31                        0
      rD                                ┌──────────────────────────┐
                                        │   DESTINATION REGISTER    │
                                        └──────────────────────────┘
```
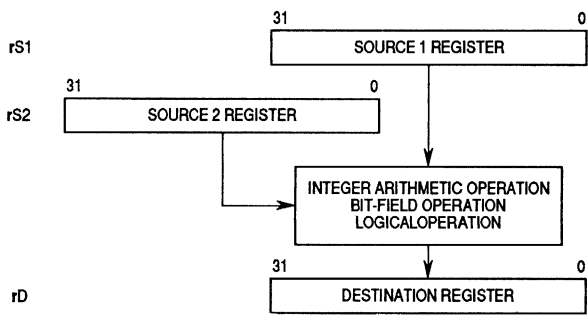
The following is the instruction format for arithmetic, logical, and bit-field instructions using triadic register addressing:
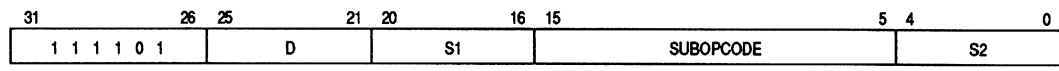
31 26	25 21	20 16	15 5	4 0
1 1 1 1 0 1	D	S1	SUBOPCODE	S2

Field	Description
D	Specifies the destination register, rD.
S1	Specifies the source 1 register, rS1. For the bit-scan instructions, this field is not used.
SUBOPCODE	Identifies the operation to be performed (**add, addu, and, clr, cmp, divs, divu, ext, extu, ff1, ff0, mak, muls, mulu, or, rot, set, sub, subu,** or **xor**).
S2	Specifies the source 2 register, rS2.

3.1.1.1.2 Floating-Point Instructions. The operands for floating-point operations can be taken from the general register file or the extended register file. The extended register file contains single-, double-, or double-extended-precision numbers. For instructions using the extended register file, the source 1, source 2, and destination registers are denoted **xS1, xS2,** and **xD,** respectively.

For floating-point instructions, the data in the source 1 registers (**rS1** or **xS1**) and source 2 registers (**rS2** or **xS2**) is processed by the floating-point unit (FPU), and the result is placed in the destination register (**rD** or **xD**). The floating point-instructions include **fmul,**

fadd, fsub, fcmp, fcmpu, and **fdiv.** In addition, the **fcvt, flt, fsqrt, int, nint, mov,** and **trnc** instructions use floating-point triadic register addressing, but the S1 field is not used by these instructions and must be filled with zeros.

The source 1 and source 2 operands must always originate from the same register file; however, depending on the instruction, the destination register may or may not have to be located in the same register file as the source registers. For the **fmul, fcvt, fadd, fsub, fsqrt,** and **fdiv** instructions, the source and destination registers must always be in the same register file. The destination for the **fcmp, fcmpu, int, nint,** and **trnc** instructions must always be in the general register file, but the source operands can be from either the general or extended register files. For the **mov** instruction, the source and destination registers cannot both be in the general register file; however, any other combination is allowed.
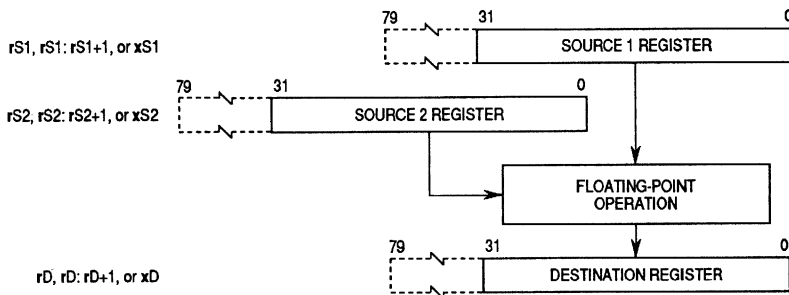
The following is the instruction format for floating-point instructions using triadic register addressing:

31	26	25	21	20	16	15	5	4	0
1 0 0 0 0 1		D		S1		SUBOPCODE		S2	
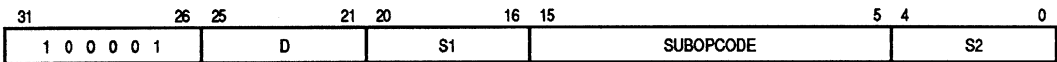
Field	Description
D	Specifies the destination register, rD or xD.
S1	Specifies the source 1 register, rS1 or xS1. For the **fcvt, fsqrt, mov, int, nint, flt,** and **trnc** instructions, S1 must be zero.
SUBOPCODE	Identifies the operation to be performed (**fmul, fadd, fsub, fcmp, fcmpu, fcvt, fdiv, flt, fsqrt, int, mov, nint,** or **trnc**).
S2	Specifies the source 2 register, rS2 or xS2.

3.1.1.1.3 Graphics Instructions. Graphics data is more efficiently processed in double-words, so the operands for most graphics instructions are contained in register pairs. The data in the source 1 (rS1:rS1+1) and source 2 (rS2:rS2+1) register pairs is processed by the graphics unit, and the result is placed in the destination register pair (rD:rD+1). The graphics instructions which process double-word data are **padd, padds, psub, psubs, pcmp, prot,** and **ppack.**

The source 1 operands for the **pmul** and **punpk** instructions are only one word in length. For the **pmul** instruction, the 64-bit value in the source 1 register pair (rS1:rS1+1) is multiplied by the 32-bit value in the source 2 register and the 64 least significant bits of the product are placed in the 64-bit destination register pair (rD:rD+1). For the **punpk** instruction, nibble, byte, or half-word fields from rS1 are placed into fields of twice their size and zero-extended. These fields are concatenated to form a 64-bit result which is placed in the destination register pair (rD:rD+1). The **punpk** instruction does not use the S2 field, so it must be filled with zeros.

The source 2 operand of the **prot** instruction is only one word in length. For this instruction, the value in the source 1 register pair (rS1:rS1+1) is rotated to the left by the number of bits specified by bits 5-2 of rS2, and the result is placed in the destination register pair (rD:rD+1). The number of bits to be rotated can also be specified as immediate operands, as described in **3.1.1.2.1 Register with 6-Bit Immediate Addressing.**

For the **pcmp** instruction, the value in the source 1 register pair (rS1:rS1+1) is compared to the value in the source 2 register pair (rS2:rS2+1) and the resulting bit string is returned to the destination register (rD).
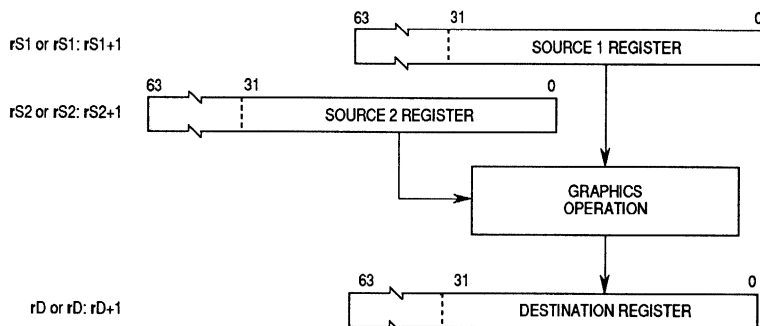
The following is the instruction format for graphics instructions using triadic register addressing:

31	26	25	21	20	16	15	5	4	0
1 0 0 0 1 0		D		S1		SUBOPCODE		S2	

Field	Description
D	Specifies the destination register, rD.
S1	Specifies the source 1 register, rS1.
SUBOPCODE	Identifies the operation to be performed (**padd, padds, pcmp, pmul, ppack, prot, psub, psubs,** or **punpk**).
S2	Specifies the source 2 register, rS2.

3.1.1.2 IMMEDIATE ADDRESSING. This type of addressing is used by instructions which require an immediate source value. The following paragraphs describe the 6-bit immediate, 10-bit immediate, 16-bit signed immediate, and 16-bit unsigned immediate addressing modes.

3.1.1.2.1 Register with 6-Bit Immediate Addressing. Register with 6-bit immediate addressing is used by the **prot** instruction. For this instruction, the value in the source 1 register pair (**rS1:rS1+1**) is rotated to the left by the number of bits specified by the immediate value in the offset (O6) field, and the result is placed in the destination register pair (**rD:rD+1**). The S2 field is not used and must be filled with zeros.

The O6 field is made up of the 4-bit rotate (R) field concatenated with the zeros in opcode bits 5 and 6. Concatenating the R field with the zeros in opcode bits 5 and 6 effectively multiplies the R field by four; therefore, the value in the O6 field is the value in the R field times four. Bits 5 and 6 of the instruction word must be zero for upward compatibility.
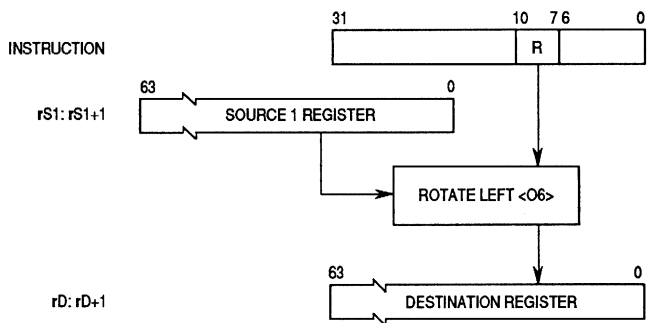
The following is the instruction format for the **prot** instruction using register with 6-bit immediate addressing:
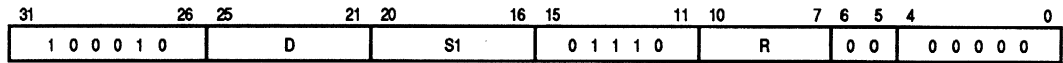
31	26	25	21	20	16	15	11	10	7	6	5	4	0
1 0 0 0 1 0		D		S1		0 1 1 1 0		R		0 0		0 0 0 0 0	

Field	Description
D	Specifies the destination register, rD.
S1	Specifies the source 1 register, rS1.
R	Specifies the number of bits to be rotated divided by four; therefore, the number of bits to be rotated equals R times 4.

3.1.1.2.2 Register with 10-Bit Immediate Addressing.
This mode of addressing is used by bit-field instructions (**clr, ext, extu, mak, rot, set**). The bit field is contained in rS1 and is defined by a 10-bit immediate field in the instruction. The 10-bit immediate field consists of two 5-bit fields which define the width and offset of the bit field from bit 0 of rS1. The bit field is processed according to the specified instruction, and the result is placed in rD.

The following is the instruction format for instructions using register with 10-bit immediate addressing:
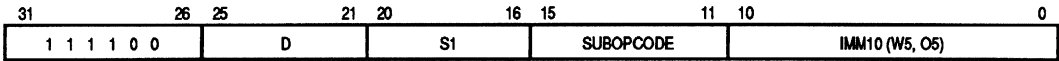
31	26	25	21	20	16	15	11	10	0
1 1 1 1 0 0		D		S1		SUBOPCODE		IMM10 (W5, O5)	

Field	Description
D	Specifies the destination register, rD.
S1	Specifies the source 1 register, rS1.
SUBOPCODE	Identifies the operation to be performed (**clr, ext, extu, mak, rot, set**).
IMM10	Contains a ten-bit immediate value which defines the width and offset of the bit field in rS1:
(W5)	Bits 9–5 define the width of the bit field
(O5)	Bits 4–0 define the offset of the bit field from bit 0 of rS1

3.1.1.2.3 Register with 16-Bit Signed Immediate Addressing. This form of addressing is used by signed arithmetic instructions which require an immediate source value. In this addressing mode, the data in rS1 and the 16-bit immediate operand are processed by an integer unit (for **add**, **sub**, and **cmp**), the multiply unit (for **muls**), or the divide unit (for **divs**), and the result is placed in rD.

The processor either sign- or zero-extends the immediate operand based on the SGN bit (bit 26) in the processor status register. If the SGN bit is clear, the processor is operating in unsigned mode and the immediate operands are zero-extended to 32-bits before being used. If the SGN bit is set, the processor is operating in signed-immediate mode and the immediate operands are sign-extended to 32-bits.
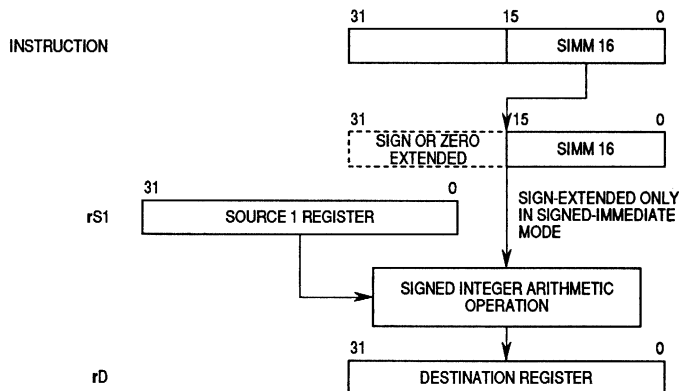
The following is the instruction format for instructions using register with 16-bit signed immediate addressing:

31	26	25	21	20	16	15	0
OPCODE		D		S1		SIMM16	
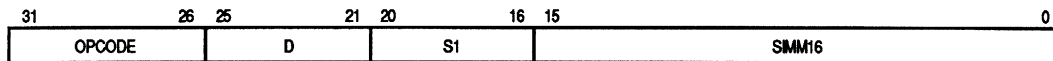
Field	Description
OPCODE	Identifies the operation to be performed (**add**, **cmp**, **divs**, **muls**, or **sub**).
D	Specifies the destination register, rD.
S1	Specifies the source 1 register, rS1.
SIMM16	Contains a 16-bit immediate value.

3.1.1.2.4 Register with 16-Bit Unsigned Immediate Addressing. This form of addressing is used by logical and unsigned arithmetic instructions which require an immediate source value.

In this addressing mode, the data in **rS1** and the 16-bit immediate operand are processed by an integer unit (for **addu, and, mask, or, subu,** and **xor**), the multiply unit (for **mulu**), or the divide unit (for **divu**), and the result is placed in **rD**. Unsigned arithmetic instructions operate identically in both the signed and unsigned immediate modes. The operands for unsigned integer arithmetic operations are zero-extended to 32-bits before being used. The operands for logical instructions are contained in the lower 16 bits of **rS1** and do not need to be extended regardless of the processor mode.
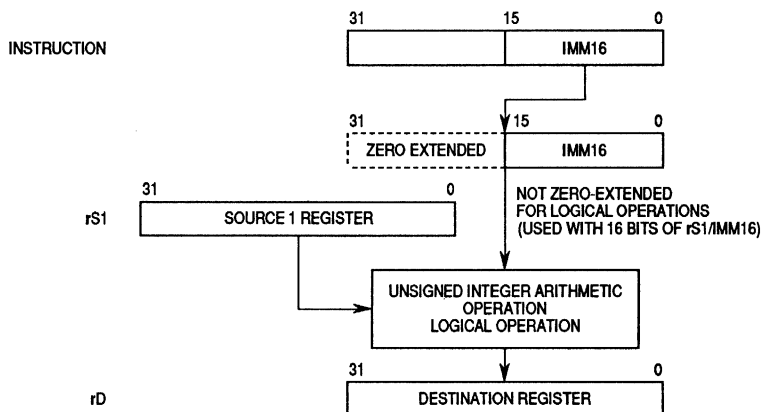
The following is the instruction format for instructions using register with 16-bit unsigned immediate addressing:

31	26	25		21	20		16	15		0
OPCODE		D			S1			IMM16		
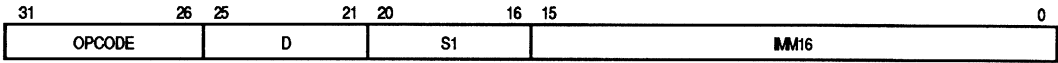
Field	Description
OPCODE	Identifies the operation to be performed (**addu, and, divu, mask, mulu, or, subu,** or **xor**).
D	Specifies the destination register, **rD**.
S1	Specifies the source 1 register, **rS1**.
IMM16	Contains a 16-bit unsigned immediate value.

3.1.1.3 CONTROL REGISTER ADDRESSING.
Control register addressing is used for referencing the general control registers and FPU control registers. In this addressing mode, general-purpose registers are loaded from, stored to, or exchanged with the control registers using the **ldcr**, **stcr**, **xcr**, **fldcr**, **fstcr**, and **fxcr** instructions.
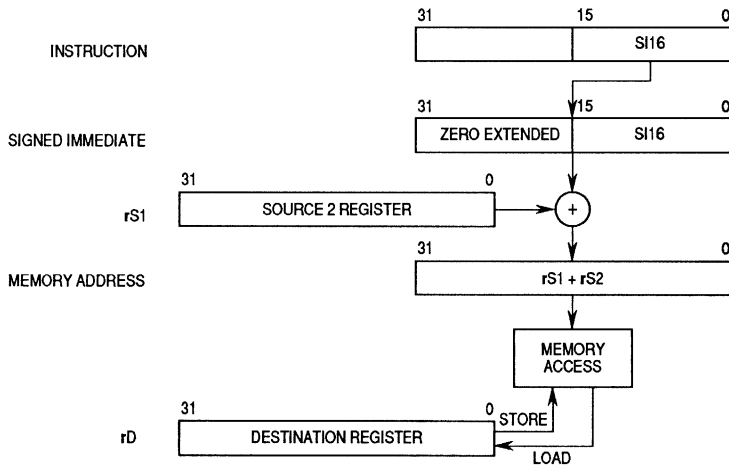
The following is the instruction format for instructions using control register addressing:

31	26	25		21	20		16	15	14	13		11	10		5	4		0
1 0 0 0 0 0		D			S1			OP		SFU			CRS/CRD			S2		
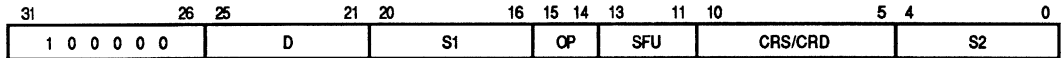
Field	Description
D	For load and exchange instructions, specifies the general register that is to be loaded with the contents of the specified control register. Must be zero for store instructions.
S1	For store and exchange instructions, specifies the general register containing the data to be transferred to the specified control register. Must be zero for load instructions.
OP	Identifies whether a load, store or exchange is to be performed (**ldcr**, **stcr**, **xcr**, **fldcr**, **fstcr**, or **fxcr**).
SFU	This field specifies which special function unit (SFU) registers are to be accessed by the instruction: a value of zero specifies the integer unit control registers; a value one specifies the floating-point unit control registers. A value of two through seven in this field causes an SFU exception for the addressed SFU.
CRS/CRD	Specifies the control register to be used. For load instructions, the control register is the source; for store instructions, the control register is the destination.
S2	Must contain the same value as the S1 field. Serves the same purpose as the S1 field.

3.1.2 Load/Store/Exchange Addressing Modes

The MC88110 supports three addressing modes for accessing data memory: register indirect with immediate index addressing, register indirect with index addressing, and register indirect with scaled index addressing. Each of these addressing modes can load data from or store data to the general register file or the extended register file. Overflow conditions in the address calculations are not detected, and results are truncated to 32 bits.

The **ld** and **st** instructions can access either the general register file (GRF) or the extended register file (XRF), as specified in the opcode. If the memory access involves data in the GRF, the operand can be a byte, half-word, word, or double-word. If the memory access involves data in the XRF, the operand can be a word, double-word, or quad-word.

3.1.2.1 REGISTER INDIRECT WITH IMMEDIATE INDEX ADDRESSING. For this type of addressing, a 16-bit immediate index is contained in the instruction. When the processor is in unsigned mode, the index is zero-extended to 32-bits, and when the processor is in signed-immediate mode, it is sign-extended. The extended immediate index is then added to the contents of rS1 and the result is truncated to 32 bits, resulting in a data memory address. For load instructions, the data at the calculated address is loaded into rD. For store instructions, the data in rD is stored to the calculated address.
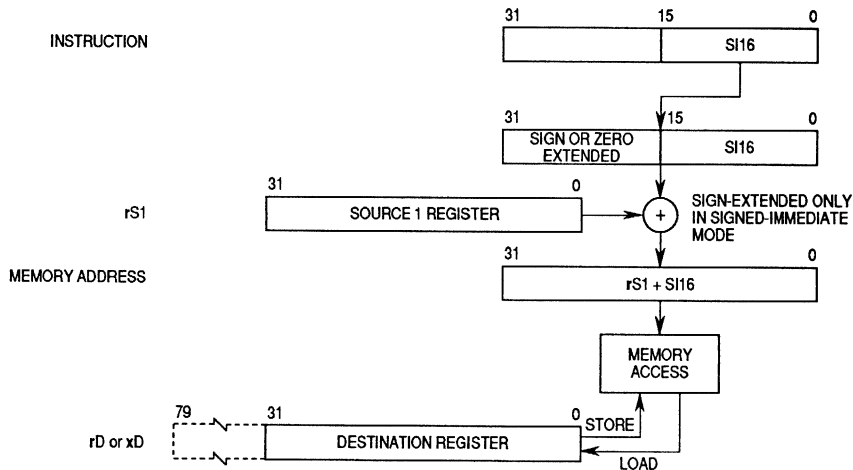
The following is the instruction format for instructions using register indirect with extended immediate index addressing:

31	26	25	21	20	16	15	0
OPCODE		D		S1		SI16	
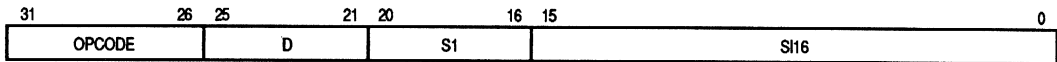
Field	Description
OPCODE	Identifies the operation to be performed (**ld** or **st**), the register file to be used (general or extended), and the data format (unsigned, single-word, double-word, quad-word, half-word, unsigned half-word, byte, unsigned byte).
D	Specifies the destination register for load instructions and the source register for store instructions.
S1	Specifies the source 1 register, rS1.
SI16	Contains a 16-bit immediate index.

3.1.2.2 REGISTER INDIRECT WITH INDEX ADDRESSING. In this addressing mode, the contents of rS1 are added to the contents of rS2 and the result is truncated to 32 bits, resulting in a data memory address. For load instructions, the memory data from the calculated address is loaded into rD. For store instructions, the data in rD is stored to the calculated address. For **xmem** instructions, the memory data from the calculated address is exchanged with the data in rD.

The following is the instruction format for instructions using register indirect with index addressing:

31		27	26	25		21	20		16	15		5	4		0
1 1 1 1 0			R		D			S1			SUBOPCODE			S2	

Field	Description
R	Identifies the type of register file to be used (general or extended). For the **xmem** and **lda** instructions, this must be one.
D	Specifies the destination register, rD or xD; rD or xD is the destination register for load instructions and the source register for store or exchange memory instructions.
S1	Specifies the source 1 register, rS1.
SUBOPCODE	Identifies the operation to be performed (**ld**, **st**, **xmem**, **lda**), the rS2 scaling factor, the size of the data being transferred, whether the data is to be transferred using user or supervisor space, and whether the store-through option should be used.
S2	Specifies the source 2 register, rS2.

3.1.2.3 REGISTER INDIRECT WITH SCALED INDEX ADDRESSING.

In this addressing mode, the contents of rS2 are first scaled according to the size of the access (i.e., byte, half-word, word, double-word, or quad-word). The scaled contents of rS2 are then added to the contents of rS1 and the result is truncated to 32 bits, resulting in a data memory address. For load instructions, the data from the calculated address is loaded into rD. For store instructions, the data in rD is stored to the memory address. For the **lda** instruction, the calculated address is loaded into rD. For **xmem** instructions, the memory data from the calculated address is exchanged with the data in rD.

Scaling the rS2 operand causes it to shift by 0, 1, 2, 3 or 4 bits (i.e., the operand is scaled by factor of 1, 2, 4, 8 or 16) for byte, half-word, word, double-word, or quad-word

accesses, respectively. For byte accesses, the result of this type of addressing is identical to the result achieved by the register indirect with index addressing (unscaled) mode, even though the SUBOPCODE fields are distinctly different for the two addressing modes.

The following is the instruction format for instructions using register indirect with scaled index addressing:

Field	Description
R	Identifies the type of register file to be used (general or extended). For the **xmem** and **lda** instructions, R must be one.
D	Specifies the destination register, rD or xD; rD or xD is the destination register for load instructions and the source register for store or exchange memory instructions.
S1	Specifies the source 1 register, rS1.
SUBOPCODE	Identifies the operation to be performed (**ld, st, xmem, lda**), the rS2 scaling factor, the size of the data being transferred, whether the data is to be transferred using user or supervisor space, and whether the store-through option should be used.
S2	Specifies the source 2 register, rS2.

3.1.3 Flow Control Addressing Modes

Flow control instructions can address or reference instruction memory using four different addressing modes: triadic register addressing, register with 9-bit vector table index addressing, register with 16-bit displacement/immediate addressing, and 26-bit branch displacement addressing. Address calculations for flow control addressing are performed using signed arithmetic. Overflows are not detected, and results are truncated to 32 bits. The following paragraphs describe the flow control addressing modes.

3.1.3.1 TRIADIC REGISTER ADDRESSING. This addressing mode is used to specify the target for **jmp** and **jsr** or the operands for the **tbnd** instruction. These flow control instructions have the same format as computational instructions which use triadic register addressing: the instruction word has three 5-bit fields which specify two source registers and a destination register. Also, as with the computational instructions, some instructions do not use all three of the register selection fields. All bits in unused fields must be zero for upward compatibility. Triadic register addressing provides access to the entire 32-bit address space.

3.1.3.1.1 Jump Instructions (jmp, jsr). For jump instructions, rS2 contains the target address of the **jmp** or **jsr** instruction. The two least significant bits of rS2 are cleared to ensure that the address is aligned to a word boundary, and program flow is transferred to the resulting address. The S1 and D fields are not used and must be filled with zeros.

```
              31                                0
        rS2   |        SOURCE 2 REGISTER         |

              31                                0
TARGET INSTRUCTION
        ADDRESS   |   SOURCE 2 REGISTER CONTENTS   |
```
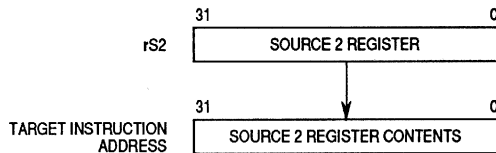
The following is the instruction format for the **jmp** and **jsr** instructions:

31 26	25 21	20 16	15 5	4 0
1 1 1 1 0 1	0 0 0 0 0	0 0 0 0 0	SUBOPCODE	S2
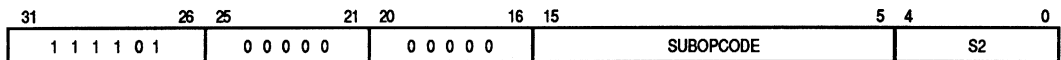
Field	Description
SUBOPCODE	Identifies the operation to be performed (**jmp, jmp.n, jsr,** or **jsr.n**).
S2	Specifies the source 2 register, rS2, which contains the target address of the **jmp** or **jsr** instruction to be executed.

3.1.3.1.2 Trap-Generating Bounds-Check Instruction (tbnd). For the **tbnd** instruction, the data in **rS1** is compared to the data in **rS2** using unsigned arithmetic, and a trap is taken if the rS1 data is greater than the rS2 data. If the trap is taken, the 20-bit address in the vector base register (VBR) is concatenated with the bounds check

exception vector and with three trailing zeros resulting in a 32-bit instruction address. Program flow begins at the resulting address. The D field is not used and must be filled with zeros.
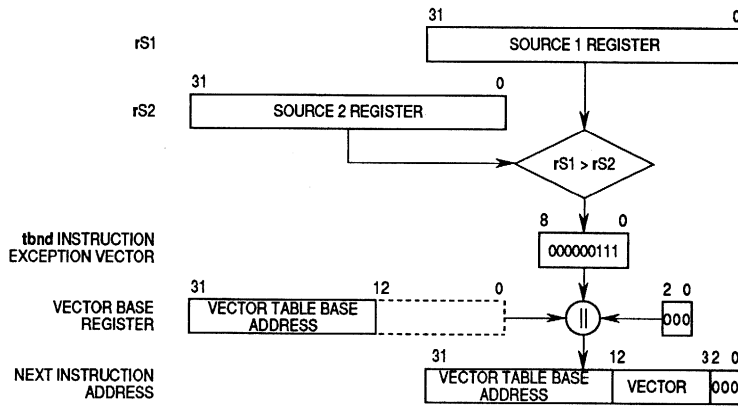
The following is the instruction format for the **tbnd** instruction when using triadic register addressing:
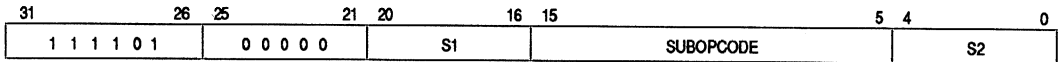
31	26	25	21	20	16	15	5	4	0
1 1 1 1 0 1		0 0 0 0 0		S1		SUBOPCODE		S2	

Field	Description
S1	Specifies the source 1 register, rS1.
SUBOPCODE	Identifies the operation to be performed (**tbnd**).
S2	Specifies the source 2 register, rS2.

3.1.3.2 REGISTER WITH 9-BIT VECTOR TABLE INDEX ADDRESSING. This addressing mode is used by the trap-generating instructions **tb0**, **tb1**, and **tcnd** (not **tbnd**).

For the bit-test trap instructions (**tb0** and **tb1**), the bit in **r**S1 specified by the B5 field of the instruction is tested for either a set or clear condition. For the conditional trap instruction (**tcnd**), **r**S1 is tested for the condition(s) specified in the M5 field of the instruction. For both instruction types, if the test condition is true, the trap is taken, the 20-bit address in the vector base register (VBR) is concatenated with the VEC9 field of the opcode and with three trailing zeros resulting in a 32-bit instruction address. Program flow begins at the resulting address.
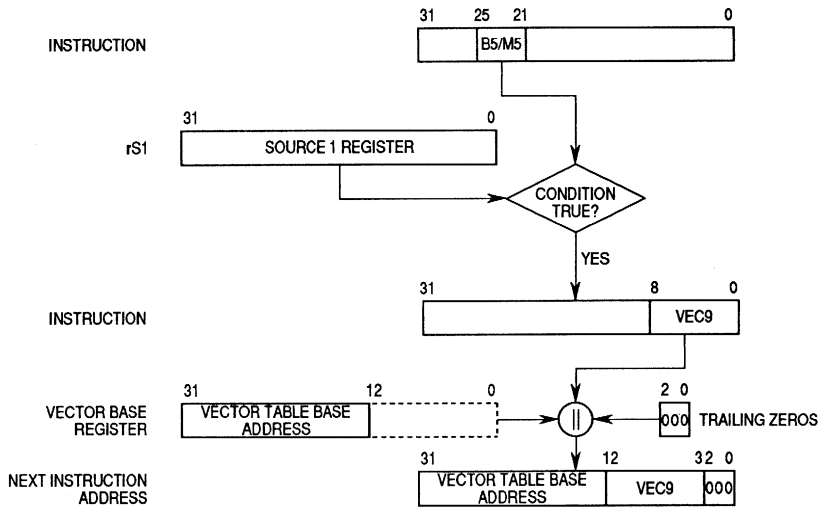
The following is the instruction format for instructions using 9-bit vector table index addressing:

31	26	25	21	20	16	15	9	8	0
1 1 1 1 0 0		B5/M5		S1		SUBOPCODE		VEC9	

Field	Description
B5/M5	For bit tests, the B5 field specifies which bit in rS1 is to be tested.
	For conditional tests, the M5 field specifies which of the following conditions for which to test the contents of rS1:
	Bit 25: Reserved; unused by the branch selection logic; must be zero for upward compatibility.
	Bit 24: Maximum negative number [Sign and Zero]
	Bit 23: Less than zero (not max) [Sign and (not Zero)]
	Bit 22: Equal to zero [(not Sign) and Zero]
	Bit 21: Greater than zero [(not Sign) and (not Zero)]
	Multiple conditions can be specified by setting more than one bit in the M5 field as shown in the following table. The most common combinations are shown, but all combinations are possible.
	Bit: 25 24 23 22 21
	eq0 (equals zero) 0 0 0 1 0
	ne0 (not equal to zero) 0 1 1 0 1
	gt0 (greater than zero) 0 0 0 0 1
	lt0 (less than zero) 0 1 1 0 0
	ge0 (greater than/equals zero) 0 0 0 1 1
	le0 (less than/equals zero) 0 1 1 1 0
S1	Specifies the source 1 register, rS1.
SUBOPCODE	Identifies the operation to be performed (tb0, tb1, tcnd).
VEC9	Contains a 9-bit vector number.

3.1.3.3 REGISTER WITH 16-BIT DISPLACEMENT/IMMEDIATE ADDRESSING.
This form of addressing is used by branch (**bb0**, **bb1**, and **bcnd**) and trap on bound (**tbnd**) instructions to generate target addresses and test conditions.

3.1.3.3.1 Bit-Test and Conditional Branch Instructions.
For the bit-test branch instructions (**bb0** and **bb1**), the bit in rS1 specified by the B5 field of the instruction is tested for either a set or clear condition. For the conditional branch instruction (**bcnd**), rS1 is tested for the condition(s) specified in the M5 field of the instruction. For both types of instructions, if the test condition is true, the 16-bit displacement specified in the instruction is shifted left two positions and sign-extended to 32 bits, and the two least significant bits are cleared to force word alignment. This 32-bit displacement value is then added to the branch instruction address, and program flow begins at the resulting address.
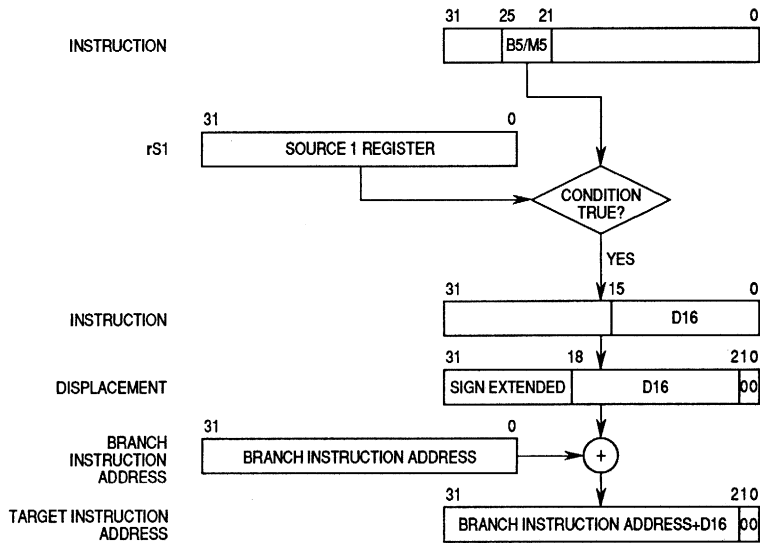
The following is the instruction format for the **bb0**, **bb1**, and **bcnd** instructions:

31	26	25	21	20	16	15	0
OPCODE		B5/M5		S1		D16	

Field	Description
OPCODE	Identifies the operation to be performed (**bb0**, **bb0.n**, **bb1**, **bb1.n**, **bcnd**, or **bcnd.n**)

B5/M5	For bit tests, the B5 field specifies which bit in rS1 is to be tested.
	
	For conditional tests, the M5 field specifies which of the following conditions for which to test the contents of rS1:
	  Bit 25: Reserved; unused by the branch selection logic; must be zero for upward compatibility.
	  Bit 24: Maximum negative number [Sign and Zero]
	  Bit 23: Less than zero (not max) [Sign and (not Zero)]
	  Bit 22: Equal to zero [(not Sign) and Zero]
	  Bit 21: Greater than zero [(not Sign) and (not Zero)]
	
	Multiple conditions can be specified by setting more than one bit in the M5 field as shown in the following table. The most common combinations are shown, but all combinations are possible.
	
	Bit: 25 24 23 22 21
	eq0 (equals zero) 0 0 0 1 0
	ne0 (not equal to zero) 0 1 1 0 1
	gt0 (greater than zero) 0 0 0 0 1
	lt0 (less than zero) 0 1 1 0 0
	ge0 (greater than/equals zero) 0 0 0 1 1
	le0 (less than/equals zero) 0 1 1 1 0

S1	Specifies the source 1 register, rS1.
D16	Specifies a signed 16-bit displacement

3.1.3.3.2 Trap-Generating Bounds-Check Instruction (tbnd). For **tbnd**, the 16-bit immediate operand (unsigned) specified in the instruction is zero-extended and then compared with the data in **rS1**. A trap is taken if the data in **rS1** is greater than the immediate operand. If the trap is taken, the 20-bit address in the VBR is concatenated with the bounds-check exception vector and three trailing zeros resulting in a 32-bit instruction address. Program flow begins at this resulting address.
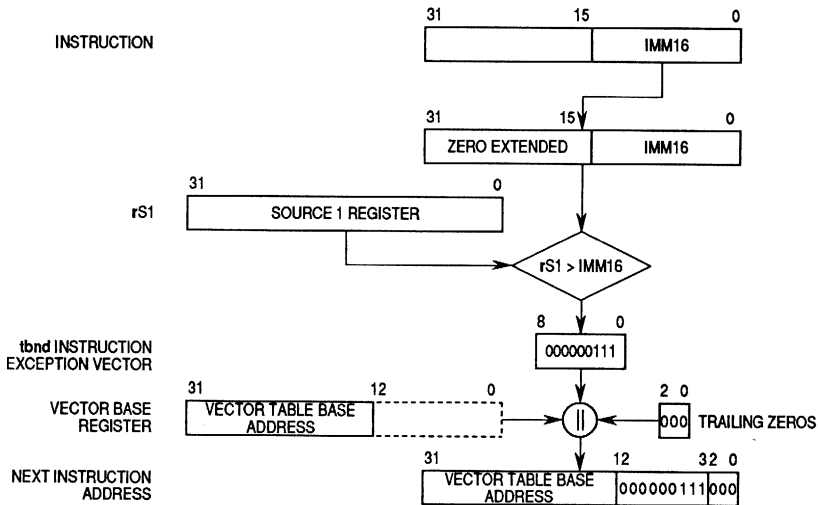
The following is the instruction format for the **tbnd** instruction when using register with 16-bit displacement/immediate addressing:

31	26 25	21 20	16 15	0
1 1 1 1 1 0	0 0 0 0 0	S1	IMM16	
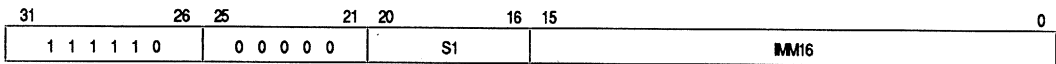
Field	Description
S1	Specifies the source 1 register, rS1.
IMM16	Specifies a 16-bit immediate operand.

3.1.3.4 26-BIT BRANCH DISPLACEMENT ADDRESSING. This form of addressing is used to specify the branch target address for unconditional branch instructions (**br** and **bsr**). The 26-bit displacement specified in the instruction word is shifted left by two bits, sign-extended to 32 bits, and added to the address of the branch. Program flow is transferred to the resulting address.
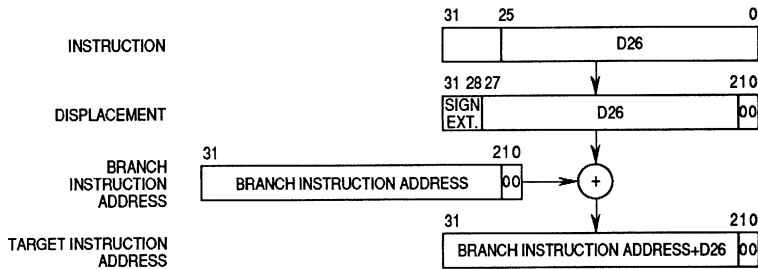
The following is the instruction format for instructions using 26-bit branch displacement addressing:
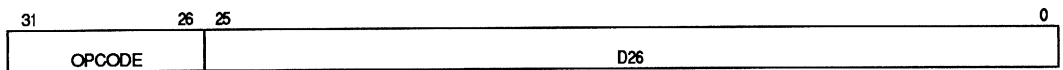
Field	Description
OPCODE	Identifies the operation to be performed (**br**, **br.n**, **bsr**, or **bsr.n**)
D26	Specifies a 26-bit displacement.

3.1.3.5 RETURN FROM EXCEPTION (rte) AND ILLEGAL OPERATION (illop) INSTRUCTION ADDRESSING.
The **rte** and **illop** instructions use an addressing mode in which no operands are specified. The **illop** instructions (**illop1**, **illop2**, and **illop3**) perform no user-visible operation but cause an unimplemented opcode exception. When the rte instruction executes, the instruction unit restores the machine state saved in the exception-time registers and resumes normal program execution.

The following is the instruction format for the **rte** instruction addressing:
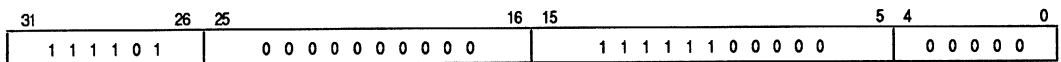
The following is the instruction format for the **illop** instruction addressing:
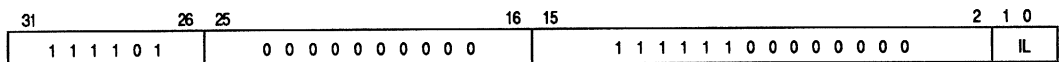
Field	Description
IL	Identifies the illegal opcode instruction
	01—illegal opcode 1
	10—illegal opcode 2
	11—illegal opcode 3

3.2 INSTRUCTION SET SUMMARY

MC88110 instructions fall into seven categories: logical, integer arithmetic, floating-point, graphics, bit-field, load/store/exchange, and flow control. The following paragraphs describe these categories and provide operand syntax and operational descriptions for the instructions in each category. Table 3-1 identifies the abbreviations and symbols used in the instruction set.

Table 3-1. Instruction Description Notations

Abbreviation/Symbol	Description
r1	General register 1
rS1	Source 1 register—General register containing the first source operand
rS2	Source 2 register —General register containing the second source operand
rD	Destination register—Register destination that will be modified by the operation or source of data on a store operation
xS1	Source 1 extended register—Extended register containing the first source operand.
xS2	Source 2 extended register —Extended register containing the second source operand
xD	Destination extended register —Extended register destination that will be modified by the operation or source of data on a store operation
crS	Source control register
crD	Destination control register
crS/D	Source and destination control registers for **xcr** instruction
fcrS	Source floating-point control register
fcrD	Destination floating-point control register
fcrS/D	Source and destination floating-point control registers for **fxcr** instruction
D16, D26	Sixteen and twenty-six bit signed instruction address displacement
IMM16	Unsigned 16-bit immediate operand
SIMM16	Signed 16-bit immediate operand; this operand is sign-extended when the processor is operating in signed mode, zero-extended when operating in unsigned mode.
SI16	Signed 16-bit immediate index; this operand is sign-extended when the processor is operating in signed mode, zero-extended when operating in unsigned mode.
VEC9	Offset from the page address contained in the vector base register
M5	Five-bit condition match field. The bits indicate the following conditions: Bit 25: Reserved Bit 24: S and Z Bit 23: S and (not Z) Bit 22: (not S) and Z Bit 21: (not S) and (not Z) S: Sign bit (bit 31 of the tested register) Z: Zero bit (logical NOR of bits 30 through 0 of the tested register)
B5	Unsigned 5-bit integer denoting a bit number within a word
O5	Unsigned 5-bit integer denoting a bit-field offset within a word
W5	Unsigned 5-bit integer denoting a bit-field width within a word (0 denotes a width of 32)

Table 3-1 Instruction Description Notations (Continued)

Abbreviation	Description
O6	Unsigned 6-bit integer denoting the number of bits to rotate a pixel
{.n}	Delayed branch option. If specified, the next sequential instruction is executed before the branch target instruction.
{.c}	Complement option. If specified, the second operand is ones-complemented before it is used in the operation.
{.d}	Double-word option. If specified for the **divu** instruction, double register rS:rS+1 is used for source 1, and rD:rD+1 is used for the destination register. If specified for the **mulu** instruction, double register rD:rD+1 is used for the destination register.
{.u}	Upper half word option. If specified, the 16-bit logical operation is performed with the upper 16 bits of the source register .
.car	Carry
{.ci}	Carry in option. If specified, includes the processor status register (PSR) carry bit in the arithmetic operation.
{.co}	Carry out option. If specified, sets or clears the PSR carry bit based on the result of the arithmetic operation.
{.cio}	Carry in/carry out option. If specified, includes the PSR carry bit in the arithmetic operation and sets or clears the carry bit based on the result.
.sz	Memory size for general register file (default = word):
.b	Byte(8 bits).
.bu	Unsigned byte (8 bits).
.h	Half word (16 bits).
.hu	Unsigned half word (16 bits).
.d	Double word (64 bits)
.xsz	Memory size for extended register file (default = word)
.d	Double word (64 bits).
.x	Quad word (128 bits).
.fsz	Floating-point operand size. The .fsz is a 3-letter designator that corresponds to the sizes of the D, S1, and S2 operands, respectively (2-letter designator for D and S2 operands for the conversion instructions). Floating-point operations support mixed operand sizes; two or three register operands can use two or three of the ".s" or ".d" qualifiers in any combination to support the operand size mix. For example: **fadd.dds r3,r5,r9** r3 and r5 are double precision, r9 is single precision, .s is single precision, .d is double precision, and .x is extended precision
.r	Graphics pack result field size:
.8	8 bits
.16	16 bits
.32	32 bits
.t	Graphics field size (default = word):
.b	Byte (8 bits)
.h	Half word (16 bits)

Table 3-1 Instruction Description Notations (Continued)

Abbreviation	Description
.x	Graphics saturation option:
.u	unsigned ± unsigned = unsigned
.s	signed ± signed = signed
.us	unsigned ± signed = unsigned
{.usr}	User memory option. This option pertains to memory access instructions, allowing the user memory space to be accessed while in the supervisor mode.
{.wt}	Store-through option. This option pertains to the store (st) instruction, forcing the store to write to the cache and to memory.
[rS2]	Scaled index
x	"Don't care" bit.
+	Add
-	Subtract
•	Multiply
::	Compare
/	Divide
‖	Concatenate
<<	Shift Left
..	Replaced by
∧	AND
∨	OR
⊕	EXCLUSIVE OR
<	Relational test; true if left operand is less than right operand
>	Relational test; true if left operand is greater than right operand
{}	Optional

3.2.1 Logical Instructions

The logical instructions provide three common logical operations: AND, OR, and XOR. An immediate mask instruction is also provided. These instructions operate on the entire rS1 operand when triadic register addressing is used or on either the lower or upper half word of the rS1 operand when register with 16-bit immediate addressing is used. In addition, when triadic register addressing is used, the logical instructions can optionally complement the rS2 operand before the operation occurs. Table 3-2 lists the logical instructions.

Table 3-2. Logical Instructions

Instruction	Name	Operand Syntax	Operation
and{.u}	Logical AND	rD,rS1,IMM16	rD← rS1 (lower or upper 16 bits) Λ IMM16. Remaining 16 bits of rS1 are copied to rD.
and{.c}	Logical AND	rD,rS1,rS2	rD ← rS1 Λ rS2 (normal or complemented)
mask{.u}	Logical Mask Immediate	rD,rS1,IMM16	rD (lower or upper 16 bits) ← rS1 (lower or upper 16 bits) Λ IMM16. Remaining bits ← zero.
or{.u}	Logical OR	rD,rS1,IMM16	rS1 (lower or upper 16 bits) V IMM16. Remaining 16 bits of rS1 are copied to rD.
or{.c}	Logical OR	rD,rS1,rS2	rD ← rS1 V rS2 (normal or complemented)
xor{.u}	Logical Exclusive Or (XOR)	rD,rS1,IMM16	rD← rS1 (lower or upper 16 bits) ⊕ IMM16. Remaining 16 bits of rS1 are copied to rD.
xor{.c}	Logical Exclusive Or (XOR)	rD,rS1,rS2	rD ← rS1 ⊕ rS2 (normal or complemented)

3.2.2 Integer Arithmetic Instructions

nteger arithmetic instructions provide the standard arithmetic operations and an integer compare operation. Signed and unsigned add and subtract, multiply and divide operations are available. Various combinations of carry bits can be specified for the add and subtract instructions. Table 3-3 lists the integer arithmetic instructions.

Table 3-3. Integer Arithmetic Instructions

Instruction	Name	Operand Syntax	Operation
add{.car}	Integer Add	rD,rS1,SIMM16 rD,rS1,rS2	rD ← rS1 + SIMM16 rD ← rS1 + rS2
addu{.car}	Unsigned Integer Add	rD,rS1,IMM16 rD,rS1,rS2	rD ← rS1 + IMM16 rD ← rS1 + rS2
cmp	Integer Compare	rD,rS1,SIMM16 rD,rS1,rS2	rD ← rS1 :: SIMM16 rD ← rS1 :: rS2
divs	Integer Divide	rD,rS1,SIMM16 rD,rS1,rS2	rD ← rS1/SIMM16 rD ← rS1/rS2
divu	Unsigned Integer Divide	rD,rS1,IMM16	rD ← rS1/IMM16
divu{.d}	Unsigned Integer Divide	rD,rS1,rS2	rD ← (rS1 or rS1:rS1+1)/rS2
muls	Integer Multiply	rD,rS1,SIMM16 rD,rS1,rS2	rD ← rS1 * SIMM16 rD ← rS1 * rS2
mulu	Unsigned Integer Multiply	rD,rS1,IMM16	rD ← rS1 * IMM16
mulu{.d}	Unsigned Integer Multiply	rD,rS1,rS2	(rD or rD:rD+1) ← rS1 * rS2
sub{.car}	Integer Subtract	rD,rS1,SIMM16 rD,rS1,rS2	rD ← rS1 - SIMM16 rD ← rS1 - rS2
subu{.car}	Unsigned Integer Subtract	rD,rS1,IMM16 rD,rS1,rS2	rD ← rS1 - IMM16 rD ← rS1 - rS2

3.2.3 Bit-Field Instructions

The bit-field instructions set, clear, make, extract, rotate, and find bit fields in the source operand. Certain bit-field instructions (**ext**, **extu**, and **mak**) can be used to perform right or left shift operations in addition to their normal functions. A bit field is defined by the width of the bit field and by the offset of the bit field from bit 0 of the source operand. Depending on the instruction, the width and offset are specified either in the instruction word or in the lower ten bits of the rS2 operand. The lower ten bits of rS2 are divided into two 5-bit fields: bits 4–0 specify the offset (<O5>) and bits 9–5 specify the width (<W5>). A width of zero specifies all 32 bits. Table 3-4 lists the bit-field instructions.

Table 3-4. Bit-Field Instructions

Instruction	Name	Operand Syntax	Operation
clr	Clear Bit Field	rD,rS1,W5<O5> rD,rS1,rS2	rD← rS1 with bit field clear. Bit field is O5 bits from bit zero, W5 bits wide.
ext	Extract Bit Field	rD,rS1,W5<O5> rD,rS1,rS2	rD ← rS1 bit field. rS1 bit field is O5 bits from bit zero, W5 bits wide, sign-extended. The resulting bit field is placed in rD starting at bit 0.
extu	Extract Bit Field Unsigned	D,rS1,W5<O5> rD,rS1,rS2	D ← rS1 bit field. rS1 bit field is O5 bits from bit zero, W5 bits wide, zero-extended. The resulting bit field is placed in rD starting at bit 0.
ff0	Find First Bit Clear	rD,rS2	rD← position of rS2 first zero bit (32 if none found). The search begins at bit 31 of rS2 (the most significant bit).
ff1	Find First Bit Set	rD,rS2	rD← position of rS2 first one bit (32 if none found). The search begins at bit 31 of rS2 (the most significant bit).
mak	Make Bit Field	rD,rS1,W5<O5> rD,rS1,rS2	rS1 bit field is W5 bits wide starting at bit zero. rD← rS1 bit field shifted left by offset O5. Remaining rD bits cleared.
rot	Rotate Register	rD,rS1,<O5> rD,rS1,rS2	rD ← rS1 rotated right by O5 bits.
set	Set Bit Field	rD,rS1,W5<O5> rD,rS1,rS2	D ← rS1 with bit field set. Bit field is O5 bits from bit zero, W5 bits wide.

3.2.4 Floating-Point Instructions

The floating-point instructions provide standard floating-point arithmetic operations and integer/floating-point conversions for various operand sizes (single-, double-, and double-extended-precision). These instructions meet the IEEE standard for binary floating-point arithmetic (ANSI-IEEE 754-1985). Included in the floating-point instruction category are instructions which access the floating-point control registers. Table 3-5 lists the floating-point instructions.

Table 3-5. Floating-Point Instructions

Instruction	Name	Operand Syntax	Operation
fadd.fsz	Floating-Point Add	rD,rS1,rS2 xD,xS1,xS2	rD ← rS1 + rS2 xD ← xS1 + xS2
fcmp.fsz	Floating-Point Compare	rD,rS1,rS2 rD,xS1,xS2	rD ← rS1 :: rS2 rD ← xS1:: xS2
fcmpu.fsz	Unordered Floating-Point Compare	rD,rS1,rS2 rD,xS1,xS2	rD ← rS1 :: rS2 rD ← xS1:: xS2
fcvt.fsz	Convert Floating-Point Precision	rD,rS2 xD,xS2	rD ← convert(rS2) xD ← convert(xS2)
fdiv.fsz	Floating-Point Divide	rD,rS1,rS2 xD,xS1,xS2	rD ← rS1/rS2 xD ← xS1/xS2
fldcr	Load From Floating-Point Control Register	rD,fcrS	rD ← fcrS
flt.fsz	Convert Integer to Floating Point	rD,rS2 xD,rS2	rD ← float(rS2) xD ← float(rS2)
fmul.fsz	Floating-Point Multiply	rD,rS1,rS2 xD,xS1,xS2	rD ← rS1*rS2 xD ← xS1*xS2
fstcr	Store to Floating-Point Control Register	rS1,fcrD	fcrD ← rS1
fsub.fsz	Floating-Point Subtract	rD,rS1,rS2 xD,xS1,xS2	rD ← rS1 - rS2 xD ← xS1 - xS2
fxcr	Exchange Floating-Point Control Register	rD,rS,fcrS/D	temp ← fcrS/D fcrS/D ← rS rD ← temp
mov{.s} mov{.d}	Register-to-Register Move	rD,xS2 xD,rS2 xD,xS2	Move the contents of rS2 (xS2) to rD (xD).
int.fsz	Round Floating Point to Integer	rD,rS2 rD,xS2	rD ← round(rS2) rD ← round(xS2)
nint.fsz	Round Floating Point to Nearest Integer	rD,rS2 rD,xS2	rD.← round_nearest(rS2) rD ← round_nearest(xS2)
trnc.fsz	Truncate Floating Point to Integer	rD,rS2 rD,xS2	rD.← trunc(rS2) rD.← trunc(xS2)

3

3.2.5 Graphics Instructions

The graphics instructions accelerate 3D graphics rendering algorithms. Multiple pixels of varying length are packed into 64-bit fields stored in register pairs. The graphics instructions process the individual fields within the 64-bit fields in parallel, avoiding the need to pull them apart and operate on them separately. Table 3-6 lists the graphics instructions.

3

Table 3-6. Graphics Instructions

Instruction	Name	Operand Syntax	Operation
padd.t	Pixel Add	rD, rS1, rS2	rD:rD+1 ← rS1:rS1+1 + rS2: rS2+1 modulo 2^t add
padds.x.t	Pixel Add and Saturate	rD, rS1, rS2	rD:rD+1 ← rS1:rS1+1 + rS2: rS2+1 modulo 2^t add and saturate
pcmp	Z-Compare	rD, rS1, rS2	rD ←rS1:rS1+1 :: rS2: rS2+1
pmul	Pixel Multiply	rD, rS1, rS2	rD:rD+1 ← rS1 * rS2: rS2+1
ppack.r.t	Pixel Truncate, Insert, and Pack	rD, rS1, rS2	rD:rD+1 ← fields of size t from rS2: rS2+1 truncated to t*r/64, packed together, and concatenated with rS1:rS1+1
prot	Pixel Rotate	rD, rS1,<O6> rD, rS1, rS2	rD:rD+1 ← rS1:rS1+1 rotated left by rS2 or O6 bits. rS2 or O6 should be an even multiple of 4
psub.t	Pixel Subtract	rD, rS1, rS2	rD:rD+1 ← rS1:rS1+1 - rS2: rS2+ modulo 2^t subtract
psubs.x.t	Pixel Subtract and Saturate	rD, rS1, rS2	rD:rD+1 ← rS1:rS1+1 - rS2: rS2+1 modulo 2^t subtract and saturate
punpk.t	Pixel Unpack	rD, rS1	rD:rD+1 ← fields of size t from rS1 are put in fields of size 2t and placed in rD:rD+1

3.2.6 Load/Store/Exchange Instructions

The load/store/exchange instructions perform memory accesses that move data of various sizes between memory and general registers. Also, this category includes the instructions that access the integer unit control registers. Table 3-7 lists the load/store/exchange instructions.

Table 3-7. Load/Store/Exchange Instructions

Instruction	Name	Operand Syntax	Operation
ld {.sz} ld {.xsz}	Load Register from Memory	rD,rS1,SI16 xD,rS1,SI16	(rD or xD) ← contents of memory location. Memory address is rS1 + SI16
ld {.sz}{.usr} ld {.xsz}{.usr}	Load Register from Memory	rD,rS1,rS2 rD,rS1,[rS2] xD,rS1,rS2 xD,rS1,[rS2]	(rD or xD) ← contents of memory location. Memory address is rS1 + rS2, or rS1 + (rS2 << scale). Scale factor = 0, 1, 2, 3, or 4 for byte, half word, word, double word, or quad word, respectively
lda (.h) lda {.xsz}	Load Address	rD,rS1,[rS2]	rD ← rS1 +(rS2 << scale) Scale factor = 1, 2, 3, or 4 for half word, word, double word, or quad word, respectively. Note that the .b size option is not available for the lda instruction
ldcr	Load from Control Register	rD,crS	rD ← crS
st {.sz} st {.xsz}	Store Register to Memory	rD,rS1,SI16 xD,rS1,SI16	Contents of memory location ← (rD or xD). Memory address is rS1 + SI16
st {.sz}{.usr}{.wt} st {.xsz}{.usr}{.wt}	Store Register to Memory	rD,rS1,rS2 rD,rS1,[rS2] xD,rS1,rS2 xD,rS1,[rS2]	Contents of memory location ← (rD or xD). Memory address is rS1 + rS2, or rS1 + (rS2 << scale). Scale factor = 0, 1, 2, 3, or 4 for byte, half word, word, double word, or quad word, respectively
stcr	Store to Control Register	rS1,crD	crD ← rS1
xmem{.bu}{.usr}	Exchange Register With Memory	rD,rS1,rS2 rD,rS1,[rS2]	rD← contents of memory location. Contents of memory location ← rD. Memory address is rS1 + rS2, or rS1 + (rS2 << scale). Scale factor = 0 or 2 for byte or word, respectively
xcr	Exchange	rD,rS,crS/D	temp← rS; rD ← crS/D; fcrS/D ← temp Control Register

3.2.7 Flow Control Instructions

The flow control instructions alter the sequential execution stream. These instructions include jump, branch and trap instructions. Table 3-8 lists the flow control instructions.

Table 3-8. Flow Control Instructions

Instruction	Name	Operand Syntax	Operation
jmp {.n}	Unconditional Jump	rS2	Program flow is transferred to the address in rS2.
jsr {.n}	Jump to Subroutine	rS2	Program flow is transferred to the address in rS2, and the address of the first instruction after the jsr (second if .n) is written to r1.
bb0 {.n}	Branch on Bit Clear	B5,rS1,D16	If bit B5 of rS1 clear, (D16 << 2) is sign-extended and added to the branch instruction address. Program flow is transferred to the resulting address.
bb1 {.n}	Branch on Bit Set	B5,rS1,D16	If bit B5 of rS1 set, (D16 << 2) is sign-extended and added to the branch instruction address. Program flow is transferred to the resulting address.
bcnd {.n}	Conditional Branch	M5,rS1,D16	If rS1 meets condition(s) M5, (D16 << 2) is sign-extended and added to the branch instruction address. Program flow is transferred to the resulting address.
br {.n}	Unconditional Branch	D26	(D26 << 2) is sign-extended and added to the branch instruction address. Program flow is transferred to the resulting address.
bsr {.n}	Branch to Subroutine	D26	The address of the first instruction after the bsr (second if .n) is written to r1. (D26 << 2) is sign-extended and added to the branch instruction address. Program flow is transferred to the resulting address.
Illop1 Illop2 Illop3	Illegal Operation	none	An unimplemented opcode exception is unconditionally taken.
tb0	Trap on Bit Clear	B5,rS1,VEC9	If bit B5 of rS1 clear, save execution context; program flow is transferred to VBR ‖ VEC9 ‖ 3 trailing zeros
tb1	Trap on Bit Set	B5,rS1,VEC9	If bit B5 of rS1 set, save execution context; program flow is transferred to VBR ‖ VEC9 ‖ 3 trailing zeros
tbnd	Trap on Bounds	rS1,rS2 rS1,IMM16	If rS1 > IMM16 or rS1 > rS2 (unsigned rS1,rS2 comparison) save execution context; program flow is transferred to VBR ‖ bounds check vector ‖ 3 trailing zeros
tcnd	Conditional Trap	M5,rS1,VEC9	If rS1 meets condition(s) M5, save execution context; program flow is transferred to VBR ‖ VEC9 ‖ 3 trailing zeros
rte	Return from Exception	none	Restore saved context

SECTION 4
FLOATING-POINT IMPLEMENTATION

This section describes the MC88110 floating-point function unit (FPU), implemented as special function unit one (SFU1), and how it conforms to the ANSI/IEEE Standard 754-1985 for binary floating-point arithmetic. Floating-point numeric representations, floating-point status and control registers, and exception handling for floating-point instructions are discussed. For more information on the specific operation of floating-point instructions and their timing, refer to **Section 9 Instruction Timing and Code Scheduling Considerations** and **Section 10 Instruction Set**.

NOTE

The MC88110 provides the capability to conform to ANSI/IEEE Standard 754-1985. Although the information presented in the following paragraphs will aid in understanding the MC88110 floating-point implementation, this information is not intended as a complete definition of the ANSI/IEEE floating-point functionality. The ANSI/IEEE standard is the governing document for this information.

The MC88110 completely conforms to the ANSI/IEEE standard when used with the software envelope supplied by Motorola. In addition to providing full conformance with the exception specification of the ANSI/IEEE standard, the software envelope implements those features of the ANSI/IEEE standard that are important functionally, but occur rarely in practice (e.g. NaNs, denormalized numbers). However, the MC88110 floating-point implementation has many features, such as support for mixed-mode arithmetic, that extend beyond the IEEE standard.

For applications that do not require strict adherence to the IEEE standard, there also is a time-critical floating-point (TCFP) mode that may be selected that provides default results for conditions that otherwise cause exceptions. For more information on exception processing with the MC88110, refer to **Section 7 Exceptions**. For a complete description of the software envelope and its interaction with the system software, refer to the *MC88110 Floating-Point Exception Envelope (FPEE) User's Guide*.

4.1 FLOATING-POINT NUMERIC REPRESENTATION

The following paragraphs describe floating-point numeric representations in the MC88110. Numeric formats, denormalized numbers, unnormalized double-extended-precision numbers, and not-a-numbers (NaNs) are discussed.

4.1.1 Floating-Point Numeric Formats

The MC88110 architecture supports three IEEE 754 standard floating-point data formats (see Figure 4-1): single-precision, double-precision, and double-extended-precision. In all three formats, numbers are encoded with the following four fields:

1. Sign field—a one-bit field which is 0 for positive numbers and 1 for negative numbers.

2. Exponent field—a bit field which represents the exponent of the floating-point number. The exponent is contained in 8 bits for single-precision numbers, 11 bits for double-precision numbers, and 15 bits for double-extended-precision numbers. The exponent is represented in excess 127 notation for single-precision numbers, in excess 1023 notation for double-precision numbers and in excess 16,383 notation for double-extended-precision numbers. Exponents are converted to excess 127, 1023, or 16,383 notation by adding a bias of 127, 1023, or 16,383, respectively, to the true exponent of the number.

   Two exponent values are reserved for special representations. A biased exponent value of zero indicates that the floating-point number is a denormalized number (mantissa nonzero or mantissa zero and leading bit one) or zero (mantissa zero and leading bit zero). A biased exponent value of all ones (binary) indicates infinity (mantissa zero) or a NaN (mantissa nonzero).

3. Leading Bit—a bit which represents the integer portion of the floating-point number. For single- and double-precision numbers this bit is implied and is referenced as the hidden bit. When the exponent is a nonzero number (but not all ones), then the leading bit is one for normalized numbers and zero for unnormalized numbers (see **4.1.4 Unnormalized Double-Extended-Precision Numbers**). When the exponent and the mantissa are zero and the leading bit is zero, the value is zero; however, if the leading bit is one, the value is denormalized (see **4.1.3 Denormalized Numbers**). For single- and double-precision numbers, the hidden bit is assumed to be a one when the exponent is a nonzero number and a zero when the exponent is zero.

4. Mantissa—a bit field which represents the fractional binary portion of both normalized and unnormalized floating-point numbers. The mantissa is contained in 23 bits for single-precision numbers, 52 bits for double-precision numbers, and 63 bits for double-extended-precision numbers. In addition, the most significant bit (MSB), which is the left-most bit, of the mantissa also distinguishes between signaling and nonsignaling NaNs (**see 4.1.5 Not-a-Numbers (NaN's)**).
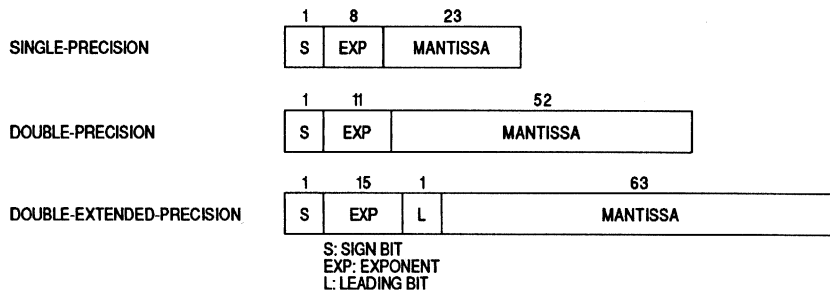
Figure 4-1. Floating-Point Data Formats

Table 4-1 contains a summary of biased exponent values and Table 4-2 contains a summary of the floating-point numbers which are recognized by the MC88110.

4

Table 4-1. Biased Exponent Value Summary

Exponent	Single-Precision	Double-Precision	Double-Extended-Precision
Maximum Exponent (Unbiased)	+127	+1023	+16,383
Minimum Exponent (Unbiased)	−126	−1022	−16,382
Exponent Bias	+127	+1023	+16,383
Exponent Width	8 bits	11 bits	15 bits

Table 4-2. Recognized Floating-Point Number Summary

Sign Bit	Exponent (Biased)	Leading Bit	Mantissa	Value
0	Maximum	x	Nonzero	+NaN
0	Maximum	x	Zero	+Infinity
0	0 < Exponent < Maximum	0	Nonzero	+Unnormalized
0	0 < Exponent < Maximum	1*	Nonzero	+Normalized
0	0	x	Nonzero	+Denormalized
0	0	1	Zero	+Denormalized
0	0	0*	Zero	+0
1	0	0*	Zero	−0
1	0	1	Zero	−Denormalized
1	0	x	Nonzero	−Denormalized
1	0 < Exponent < Maximum	0	Nonzero	−Unnormalized
1	0 < Exponent < Maximum	1*	Nonzero	−Normalized
1	Maximum	x	Zero	−Infinity
1	Maximum	x	Nonzero	−NaN

x: don't care
* not visible for single- and double-precision numbers (hidden)

NOTE

All floating-point operands should be explicitly converted to the desired precision before use. Explicit conversion does not carry an associated performance penalty since all floating-point instructions support full mixed-mode operations. Specifying a precision for an operand that is different from the precision used to originally generate the operand, without explicit conversion, is a programming error.

Table 4-3 summarizes the values of the numbers generated by the MC88110 that differ from the representations that are recognized by the MC88110. All values in Table 4-3, except for positive and negative infinity, are generated by the MC88110 only in TCFP mode (see **4.3.3 Time-Critical Floating-Point (TCFP) Mode**).

Table 4-3. Summary of Results Generated by MC88110

Sign Bit	Exponent (Biased)	Leading Bit	Mantissa	Results
0	Maximum	1*	110...0	+Universal NaN (nonsignaling)
0	Maximum	0	Zero	+Infinity
0	N/A**	N/A**	100...0	+Large Integer
1	Maximum	1*	110...0	−Universal NaN (nonsignaling)
1	Maximum	0	Zero	−Infinity
1	N/A**	N/A**	Zero	−Large Integer

* not visible for single- and double-precision numbers (hidden)

** not applicable because the result is an integer, i.e., the +large integer is 01000...0 and the − large integer is 10000...0

4.1.2 Normalized Floating-Point Numbers

The positive and negative normalized number formats are used to represent real floating-point numbers. The four fields that define normalized floating-point numbers are derived from the floating-point value as shown in Example 1. This example shows the normalized representation of the number 1.0_{10} in single-precision format (see Figure 4-2). Note that the mantissa represents all digits to the right of the binary point.

Example 1:

Value = 1.0_{10} = 1.0_2 = $1.0 * 2^0$

Sign Bit = 0

Exponent = 0

Biased Exponent (= exponent+127) = +127

Hidden Bit = 1

Mantissa = 0
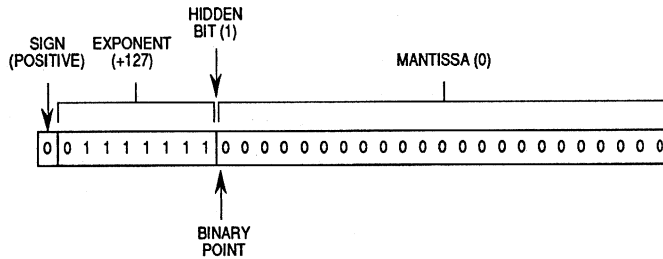
Figure 4-2. Single-Precision Floating-Point Representation of 1.0

Example 2 shows the normalized representation of the number 1/8 (0.125) in single-precision format (see Figure 4-3):

Value = 0.125_{10} = 0.001_2 = $1.0 * 2^{-3}$

Sign bit = 0

Exponent = -3

Biased Exponent (= exponent +127) = +124

Hidden Bit = 1

Mantissa = 0
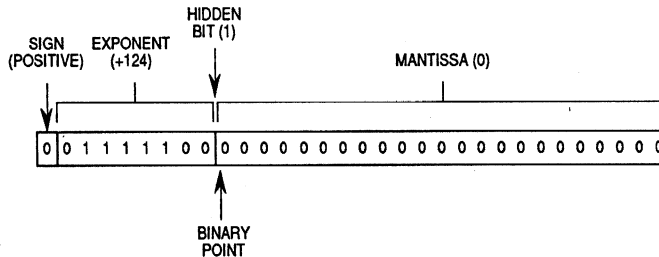
Figure 4-3. Single-Precision Floating-Point Representation of 1/8 (.125)

4.1.3 Denormalized Numbers

Denormalization occurs when a number is too small to be represented as a normalized number in the specified format. For example, the smallest single-precision normalized number that can be normally represented is $1.0 * 2^{-126}$. If this number is divided by four, the result cannot be represented as a single-precision normalized number.

Denormalized numbers are represented by a biased exponent of zero with a nonzero mantissa. Also, the double-extended-precision number with the biased exponent zero,

the mantissa zero, and the leading bit one is treated as a denormalized number. The value of the denormalized number can be calculated from the nonzero mantissa using the following equation:

$$\text{denormalized number} = \text{leading bit.<mantissa>} \times 2^{-\text{minimum exponent}}$$

where the leading bit is zero for single- and double-precision numbers.

Therefore, to represent $1.0 * 2^{-126} \div 4$, the following conversion is made:

$$(1.0 * 2^{-126}) \div (1.0 * 2^2) = (1.0 * 2^{-128}) = (0.01 * 2^{-126})$$

The denormalized result of the preceding calculation is represented with a sign bit of zero, an exponent of zero (indicating a denormalized number), and a mantissa of $.01_2$ (see Figure 4-4). Since the mantissa is 2^{-2} ($.01_2$), the format indicates that the desired result was 2^{-128}.
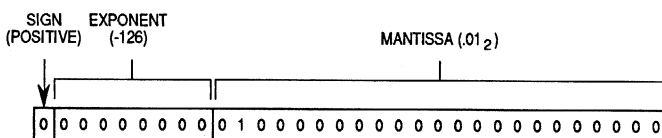
Figure 4-4. Example of a Denormalized Number

When the MC88110 is not operating in TCFP mode and an instruction specifies a denormalized source operand, a floating-point reserved operand exception occurs when the instruction begins execution. If the exception handler provided in the software envelope is used, the handler performs the operation and returns the result to the destination register of the instruction that caused the exception. The denormalized source is not affected by this process and remains denormalized.

When the result of an operation is too small to be represented as a normalized number in the specified format, a floating-point underflow exception occurs upon completion of the instruction. Refer to **4.3 Floating-Point Exceptions** for a definition of IEEE exception conditions and descriptions of the functions performed by the software envelope for the various exception conditions.

When the MC88110 is operating in TCFP mode and a denormalized number is specified as a source operand, a nonsignaling NaN is returned to the destination register.

4.1.4 Unnormalized Double-Extended-Precision Numbers

Because double-extended-precision numbers have an explicit leading bit of either 1 or 0, there is the possibility of more than one encoding for a given number. For example:

$$1.1001 * 2^{011} = 0.1101 * 2^{100}$$

where the first number is normalized and the second number is unnormalized. Note that unnormalized numbers are distinguished from denormalized numbers by the fact that unnormalized numbers have a nonzero biased exponent.

The IEEE standard requires that redundant encodings either be disallowed or that they be indistinguishable from each other. The MC88110 accommodates the second alternative. When an instruction specifies an unnormalized source operand, a floating-point reserved operand exception occurs when the instruction begins execution. The exception handler provided in the software envelope then normalizes the number, performs the operation, and returns the result to the destination register of the instruction that caused the exception. The unnormalized source is not affected by this process and remains unnormalized. The MC88110 never generates unnormalized results.

4.1.5 Not-a-Numbers (NaNs)

The IEEE standard provides for the representation of NaNs. There are two types of NaNs: signaling and nonsignaling. When an instruction specifies either type of NaN as a source operand, a floating-point reserved operand exception occurs when the instruction begins to execute; however, signaling NaNs cause the IEEE invalid operation user exception handler to be invoked when enabled (see **4.3 Floating-Point Exceptions**).

Signaling NaNs are useful for representing uninitialized variables and uninitialized memory. The MSB of the mantissa contains a zero for signaling NaNs. Nonsignaling NaNs are useful for representing the results of invalid operations such as 0/0. The MSB of the mantissa contains a one for nonsignaling NaNs. The MC88110 only generates NaNs while in TCFP mode (see **4.3.3 Time-Critical Floating-Point (TCFP) Mode**) and these NaNs are nonsignaling.

4.2 ROUNDING

The FPU supports four rounding modes that can be used for floating-point calculations: round-to-nearest, round-toward-zero, round-toward-negative-infinity, and round-toward-positive-infinity. Bits 15 and 14 in the floating-point control register (FPCR) (see **4.3.1.2 Floating-Point Control Register (FPCR)**) are used to select the desired rounding mode as shown in Table 4-4. To determine the outcome of a rounding operation, the rounding modes rely on three extra bits of precision which are generated from the floating-point result being rounded. The rounding modes and extra bits of precision are consistent with the IEEE standard. The **nint** and **trnc** instructions always round as specified in the instruction description (see **Section 10 Instruction Set**), regardless of the current rounding mode.

Table 4-4. Rounding Modes

FPCR Bits		Rounding Modes
15	**14**	
0	0	Round-to-Nearest—The result is rounded up to the next higher number when the extra bits of precision make the result closer to the higher number than to the original result. If there is a tie, round to even.
0	1	Round-toward-Zero—Extra bits of precision are truncated.
1	0	Round-toward-Negative-Infinity—A negative result is rounded down to the next more negative number if any of the extra bits of precision are set. Positive results are truncated.
1	1	Round-toward-Positive-Infinity—A positive result is rounded up to the next more positive number if any of the extra bits of precision are set. Negative results are truncated.

The three extra bits of precision are defined as follows (see Figure 4-5):

1. The Guard Bit (G)—The bit immediately to the right of the least significant bit (LSB) of the number being rounded.
2. The Round Bit (R)—The bit immediately to the right of the guard bit.
3. The Sticky Bit (S)—The logical OR of all the bits that would be to the right of the round bit if the result was infinitely precise.

Note that these bits are not visible in the MC88110 programming model.
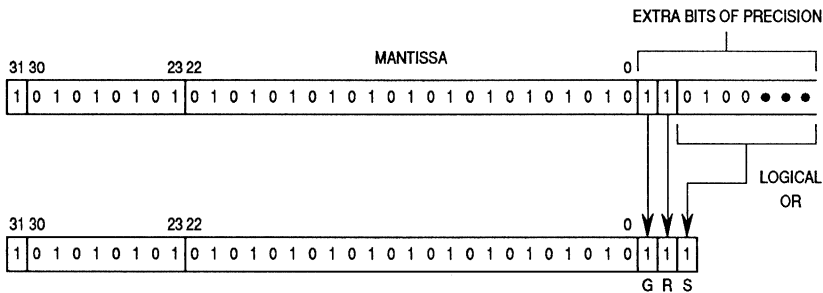
Figure 4-5. The Guard, Round, and Sticky Bits

NOTE

Note that mixed-mode operations, except in the round-toward-zero mode, can produce more accurate results than specified by the IEEE standard. Therefore, the round-toward-zero mode should be used when strict compliance with the IEEE standard is required.

4.2.1 Round-to-Nearest

Round-to-nearest is the default rounding mode after the MC88110 is reset. In this mode, a result is rounded up to the next higher number when the guard, round, and sticky bits make the result closer to the higher number than to the intermediate result.

A tie situation occurs when the guard bit is one and the round and sticky bits are zero. In the case of a tie, rounding depends on the LSB of the result: the result is rounded up if the LSB is one and is unchanged if the LSB is zero (G, R, and S are truncated).

The following statements summarize the round-to-nearest rounding mode:

If G=0—Do Not Round

If G=1 and (R=1 and/or S=1)—Round Up

If G=1, R=0, and S=0

and LSB = 0—Do Not Round

and LSB = 1—Round Up

4.2.2 Round-toward-Zero

When the round-toward-zero rounding mode is selected, the guard, round, and sticky bits are truncated.

4.2.3 Round-toward-Positive-Infinity

In the round-toward-positive-infinity mode, only positive results require the use of the extra bits of precision. If a result is positive and any of the extra bits of precision are set, the result is rounded up to the next higher number. After rounding, the guard, round, and sticky bits are discarded. Negative numbers are truncated in this mode.

4.2.4 Round-toward-Negative-Infinity

In the round-toward-negative-infinity mode, only negative results require the use of the extra bits of precision. If a result is negative and any of the extra bits of precision are set, the result is rounded down to the next lower number. After rounding, the guard, round, and sticky bits are discarded. Positive numbers are truncated in this mode.

4.3 FLOATING-POINT EXCEPTIONS

There are three definitions of floating-point exceptions that are referenced in this manual: (1) the SFU1 exception, (2) floating-point exceptions, and, (3) IEEE exception conditions. First, the MC88110 hardware automatically uses one exception vector, defined by the exception vector table to be the SFU1 exception vector (see **Section 7 Exceptions**), for all floating-point exceptions detected by the MC88110. Second, floating-point exceptions are the eight events (privilege violation, underflow, overflow, etc.) that cause the SFU1 exception to occur under the default operation (out of reset) of the MC88110. The program residing at the location of the SFU1 exception vector may

then use the bits in the floating-point exception cause register (FPECR) to explicitly branch to the appropriate routine for each of the eight floating-point exceptions. Third, there are five exception conditions that are defined by the IEEE standard and are referenced as IEEE exception conditions.

The software envelope maps seven of the eight floating-point exceptions into the five IEEE exception conditions as shown in Figure 4-6 in order to provide IEEE conformance. Note from Figure 4-6 that the software envelope maps the floating-point exceptions depicted with a dashed line to the corresponding IEEE exception conditions only in certain cases, whereas it always maps the floating-point exceptions depicted with a solid line to the specified IEEE exception conditions.

The floating-point privilege violation exception is specific to the MC88110 and does not map into an IEEE exception condition. The floating-point unimplemented opcode and floating-point reserved operand exceptions may or may not map to IEEE exception conditions, depending on the cause. However, handlers for the floating-point privilege violation, floating-point unimplemented opcode, and floating-point reserved operand exceptions are also provided in the software envelope.

FLOATING-POINT UNIMPLEMENTED OPCODE	———————	COULD MAP TO ANY
FLOATING-POINT PRIVILEGE VIOLATION	———————	NEVER MAPS TO ANY
FLOATING-POINT INTEGER CONVERSION OVERFLOW	———————>	IEEE INVALID OPERATION
FLOATING-POINT RESERVED OPERAND	———————	COULD MAP TO ANY
FLOATING-POINT DIVIDE-BY-ZERO	———————>	IEEE DIVIDE-BY-ZERO
FLOATING-POINT UNDERFLOW	- - - - - - - ->	IEEE UNDERFLOW
FLOATING-POINT INEXACT	———————>	IEEE INEXACT
FLOATING-POINT OVERFLOW	———————>	IEEE OVERFLOW

LEGEND:
———> ALWAYS MAPS
- - -> SOMETIMES MAPS

Figure 4-6. Mapping of Floating-Point Exceptions to IEEE Exception Conditions

The MC88110 has the ability to enable user-specified handlers for each of the five IEEE exception conditions. The software envelope explicitly checks the bits in the FPCR and passes parameters to the system software for branches to the appropriate user routine when it is enabled and the corresponding IEEE exception condition occurs. The system software should then perform the branch to the user handler. These routines are referenced as user routines in this section, but this does not imply that they necessarily

execute in user mode as defined by the supervisor/user mode bit of the PSR (see **Section 2 Programming Model**).

The system software can use the eight floating-point exceptions and the software envelope to provide full binary floating-point exception compatibility with the IEEE standard. However, supervisor software can also enable time-critical floating-point (TCFP) mode when strict IEEE conformance is not required. In TCFP mode, the hardware generates default results instead of taking the SFU1 exception when IEEE exception conditions occur. The software envelope is invoked only if non-IEEE exception conditions cause the exception.

When the SFU1 exception occurs, the MC88110 suspends all operations, signals the floating-point exception in the FPECR, and branches to the address specified by the vector base register and exception vector table (see **Section 7 Exceptions**). The software envelope can then be invoked to process the exception in a predefined way.

Table 4-5 depicts a summary of all the floating-point instructions of the MC88110 and the exceptions that each of these instructions can cause. The exceptions are itemized by setting the corresponding bit in the FPECR. Refer to **Section 7 Exceptions** for a more detailed description of exception processing for all exceptions.

Table 4-5. Exceptions Caused by Floating-Point Instructions

Instructions	FIOV	FUNIMP	FROP	FDVZ	FUNF	FOVF	FINX	FPRV
fmul		SFU1 disabled, odd reg. pair	NaN, invalid, denorm, or unnorm		Underflow	Overflow	Inexact	
fadd		SFU1 disabled, odd reg. pair	NaN, invalid, denorm, or unnorm		Underflow	Overflow	Inexact	
fsub		SFU1 disabled, odd reg. pair	NaN, invalid, denorm, or unnorm		Underflow	Overflow	Inexact	
fcvt		SFU1 disabled, odd reg. pair	NaN, denorm, or unnorm		Underflow	Overflow	Inexact	
fcmp		SFU1 disabled, odd reg. pair	NaN, denorm, or unnorm					
fcmpu		SFU1 disabled, odd reg. pair	NaN, denorm, or unnorm					
flt		SFU1 disabled, odd reg. pair					Inexact	

Table 4-5. Exceptions Caused by Floating-Point Instructions (Continued)

Instructions	FIOV	FUNIMP	FROP	FDVZ	FUNF	FOVF	FINX	FPRV
int	$rS2 < -2^{31}$, $rS2 > 2^{31}-1$	SFU1 disabled, odd reg. pair	NaN, denorm, or unnorm				Inexact	
nint	$rS2 < -2^{31}$, $rS2 > 2^{31}-1$	SFU1 disabled, odd reg. pair	NaN, denorm, or unnorm				Inexact	
trnc	$rS2 < -2^{31}$, $rS2 > 2^{31}-1$	SFU1 disabled, odd reg. pair	NaN, denorm, or unnorm				Inexact	
fdiv		SFU1 disabled, odd reg. pair	NaN, invalid, denorm, or unnorm	$rS2 = 0$	Underflow	Overflow	Inexact	
fsqrt		Always						
mov		SFU1 disabled, odd reg. pair						
fldcr, fstcr, fxcr		SFU1 disabled ***						*
other FP**		Always						

* FPRV set when any of these instructions specify any of fcr0–fcr61 while operating in the user mode (as determined by supervisor/user mode bit of PSR—see **Section 2 Programming Model**). Note that when fcr1–fcr61 are referenced while operating in user mode, the FPRV bit is set but the FUNIMP bit is not.

** "Other FP" refers to all other opcodes (not described above) that map into the SFU1 opcode space.

*** FUNIMP set when any of these instructions specify any of fcr1–fcr61 while operating in the supervisor mode (as determined by the supervisor/user mode bit of PSR—see **Section 2 Programming Model**).

The following paragraphs describe the floating-point registers, the handling of floating-point exceptions by the software envelope and the operation of TCFP mode.

4.3.1 Floating-Point Control Registers

The MC88110 implements three floating-point control registers as follows:

fcr0—floating-point exception cause register (FPECR)

fcr62—floating-point status register (FPSR)

fcr63—floating-point control register (FPCR)

The floating-point control registers are accessed using the **fldcr, fstcr, fxcr** instructions (see **Section 10 Instruction Set**).

4.3.1.1 FLOATING-POINT EXCEPTION CAUSE REGISTER (FPECR). The FPECR is written by the hardware whenever floating-point exceptions occur to indicate which floating-point exception has occurred when the SFU1 exception is taken. Each of the possible eight MC88110 floating-point exceptions has a corresponding bit in the FPECR which is set by the hardware when that exception occurs. Some exceptions, such as overflow and inexact, occur simultaneously and thus multiple bits may be set in

the FPECR. If the floating-point unimplemented instruction bit is set, then all other bits in the FPECR are undefined.

The FPECR is read by the software envelope to determine which floating-point exception occurred. The FPECR has read/write access and is accessible from supervisor mode only. The FPECR and its defined bits are shown in Figure 4-7. Refer to section **4.3.2 IEEE Exceptions Conformance** for more detail on the causes of these eight exceptions and actions performed by the software envelope in response to these various conditions. In the paragraphs that follow, an asterisk (*) denotes the default state after reset.

31		8	7	6	5	4	3	2	1	0
			FIOV	FUNIMP	FPRV	FROP	FDVZ	FUNF	FOVF	FINX

☐ RESERVED FOR FUTURE USE
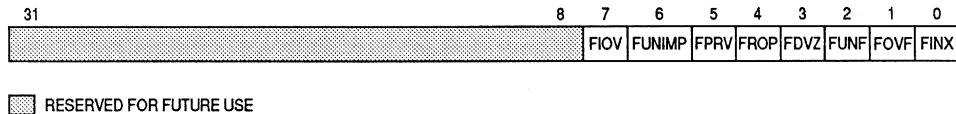
Figure 4-7. Floating-Point Exception Cause Register

Bits 31–8—Reserved

Always read as zero but not guaranteed to be zeros in future implementations; writes are ignored.

FIOV—Floating-Point to Integer Conversion Overflow

This bit is set by the MC88110 to indicate that the exception was caused by a floating-point to integer conversion overflow.

0—No floating-point to integer conversion overflow*
1—Exception caused by floating-point to integer conversion overflow

FUNIMP—Floating-Point Unimplemented Instruction

This bit is set by the MC88110 to indicate that the exception was caused by a floating-point instruction opcode that is unimplemented in the MC88110 hardware. In addition, when SFU1 opcodes are attempted to be executed when SFU1 is disabled in the PSR (see **Section 2 Programming Model**), the FUNIMP bit is set.

0—No floating-point unimplemented instruction*
1—Exception caused by a floating-point unimplemented instruction

FPRV—Floating-Point Privilege Violation

This bit is set by the MC88110 to indicate that the exception was caused by an attempt to access a privileged (implemented or unimplemented) floating-point control register while in user mode.

0—No floating-point privilege violation*
1—Exception caused by a floating-point privilege violation

FROP—Floating-Point Reserved Operand

This bit is set by the MC88110 to indicate that the exception was caused by either a floating-point reserved operand check (nonsignaling NaN, denormalized operand, or double-extended-precision unnormalized operand was specified) or by an invalid operation with zero, infinity, or signaling NaN (see **4.3.2.4 Floating-Point Reserved Operand**).

0—No floating-point reserved operand*
1—Exception caused by a floating-point reserved operand

FDVZ—Floating-Point Divide-by-Zero

This bit is set by the MC88110 to indicate that the exception was caused by the division of a normalized nonzero number by zero or the division of infinity by zero. Note that the division of zero by zero and division of NaN by zero do not cause the FDVZ bit to be set, but instead cause the FROP bit of FPECR to be set.

0—No floating-point divide-by-zero*
1—Exception caused by floating-point divide-by-zero

FUNF—Floating-Point Underflow

This bit is set by the MC88110 to indicate that the exception was caused by a floating-point underflow.

0—No floating-point underflow*
1—Exception caused by floating-point underflow

FOVF—Floating-Point Overflow

This bit is set by the MC88110 to indicate that the exception was caused by a floating-point overflow.

0—No floating-point overflow*
1—Exception caused by floating-point overflow

FINX—Floating-Point Inexact

This bit is set by the MC88110 to indicate that the exception was caused by a floating-point inexact condition. A floating-point overflow condition also causes this bit to be set.

0—No floating-point inexact condition*
1—Exception caused by floating-point inexact condition

4.3.1.2 FLOATING-POINT CONTROL REGISTER (FPCR). The FPCR is used to specify the desired rounding mode and to specify which IEEE floating-point exception conditions should branch to a user software exception handler. The FPCR defines one bit for each of the five user-enabled IEEE floating-point exception conditions, two bits for specifying the rounding mode, and three bits for enabling TCFP mode (see **4.3.3 Time-Critical Floating-Point (TCFP) Mode**). The FPCR has read/write access and is accessible from both user and supervisor modes. The FPCR and its defined bits are shown in Figure 4-8. In the following paragraphs, an asterisk (*) denotes the default state after reset.

31	22	21	20	19	18	17	16	15	14	13	5	4	3	2	1	0
		TCFP			TCFPUNF	TCFPOVF		RM	RM			EFINV	EFDVZ	EFUNF	EFOVF	EFINX
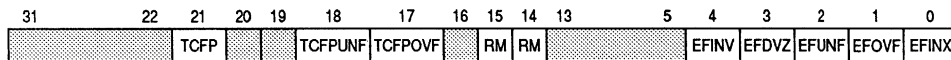
▨ RESERVED FOR FUTURE USE

Figure 4-8. Floating-Point Control Register

Bits 31–22— Reserved

Always read as zero but not guaranteed to be zeros in future implementations; writes are ignored.

TCFP—Time-Critical Floating-Point Mode

This bit enables TCFP mode. If this bit is set, the TCFPUNF and TCFPOVF bits are ignored.

    0—Take SFU1 exception for all IEEE floating-point exception conditions*
    1—Return TCFP mode default results for all IEEE floating-point exception conditions and do not cause SFU1 exception

Bits 20,19—Reserved

Always read as zero but not guaranteed to be zeros in future implementations; writes are ignored.

TCFPUNF—Time-Critical Floating-Point Underflow

This bit enables TCFP mode for underflow conditions; it is ignored if the TCFP bit is set.

    0—Take SFU1 exception for floating-point underflow condition*
    1—Return correctly signed zero for floating-point underflow and do not cause SFU1 exception

TCFPOVF—Time-Critical Floating-Point Overflow

This bit enables TCFP mode for overflow conditions; it is ignored if the TCFP bit is set.

    0—Take SFU1 exception for floating-point overflow conditions*
    1—Return correctly signed infinity for floating-point overflow and do not cause SFU1 exception

Bit 16—Reserved

Always read as zero but not guaranteed to be zeros in future implementations; writes are ignored.

RM—Rounding Mode

These two bits are used to by the hardware for rounding floating-point calculations.

    00—Round-to-nearest*
    01—Round-toward-zero
    10—Round-toward-negative-infinity
    11—Round-toward-positive-infinity

Bits 13–5—Reserved

Always read as zero but not guaranteed to be zeros in future implementations; writes are ignored.

EFINV—Enable Invalid Operation User Exception Handler

0—Disable invalid operation user exception handler*
1—Enable invalid operation user exception handler

EFDVZ—Enable Divide-by-Zero User Exception Handler

0—Disable divide-by-zero user exception handler*
1—Enable divide-by-zero user exception handler

EFUNF—Enable Underflow User Exception Handler

0—Disable underflow user exception handler*
1—Enable underflow user exception handler

EFOVF—Enable Overflow User Exception Handler

0—Disable overflow user exception handler*
1—Enable overflow user exception handler

EFINX—Enable Inexact Exception Handler

0—Disable inexact user exception handler*
1—Enable inexact user exception handler

4.3.1.3 FLOATING-POINT STATUS REGISTER (FPSR). Each of the five IEEE exception conditions has a corresponding bit in the FPSR that is set by the software envelope (except for the inexact bit which can also be set by the hardware) when the exception occurs. The FPSR also defines the XMOD bit which is set by hardware to indicate that the extended register file has been modified. Neither the hardware nor the software envelope clear bits in the FPSR; the bits must be cleared by user software. The FPSR has read/write access and is accessible from both user and supervisor modes. The FPSR and its defined bits are shown in Figure 4-9. In the following paragraphs, an asterisk (*) denotes the default state after reset.

31		17	16	15		5	4	3	2	1	0
			XMOD				AFINV	AFDVZ	AFUNF	AFOVF	AFINX

▒ RESERVED FOR FUTURE USE
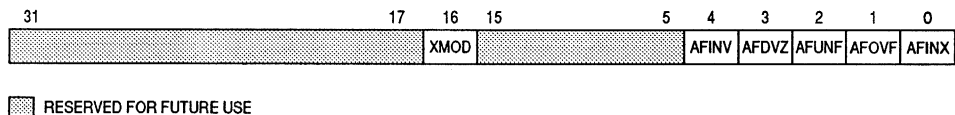
Figure 4-9. Floating-Point Status Register

Bits 31–17—Reserved

Always read as zero but not guaranteed to be zeros in future implementations; writes are ignored.

XMOD—Extended Register File Modified

This bit is used by the MC88110 to indicate that the extended register file has been modified.

    0—Extended register file not modified*
    1—Extended register file modified

Bits 15–5—Reserved

Always read as zero but not guaranteed to be zeros in future implementations; writes are ignored.

AFINV—Accumulated Invalid Operation Flag

This bit is set by the software envelope to indicate that an IEEE invalid operation exception condition has occurred.

    0—No IEEE invalid operation exception condition*
    1—IEEE invalid operation exception condition

AFDVZ—Accumulated Divide-by-Zero Flag

This bit is set by the software envelope to indicate that an IEEE divide-by-zero exception condition has occurred.

    0—No IEEE divide-by-zero exception condition*
    1—IEEE divide-by-zero exception condition

AFUNF—Accumulated Underflow Flag

This bit is set by the software envelope to indicate that an IEEE underflow exception condition has occurred.

    0—No IEEE underflow exception condition*
    1—IEEE underflow exception condition

AFOVF—Accumulated Overflow Flag

This bit is set by the software envelope to indicate that an IEEE overflow exception condition has occurred.

    0—No IEEE overflow exception condition*
    1—IEEE overflow exception condition

AFINX—Accumulated Inexact Flag

This bit is set by the hardware to indicate that an IEEE inexact exception condition has occurred. In addition, the software envelope sets this bit as well as the AFOVF bit when both the overflow and inexact user handlers are disabled and an overflow exception condition occurs.

    0—No IEEE inexact exception condition*
    1—IEEE inexact exception condition

4

4.3.2 IEEE Exceptions Conformance

In addition to providing full conformance with the IEEE exception specification, the software envelope implements those features of the IEEE standard that are important functionally, but occur rarely in practice (e.g. NaNs and denormalized numbers). For applications that do not require strict adherence to the IEEE standard, TCFP mode may be selected (see **4.3.3 Time-Critical Floating-Point (TCFP) Mode**).

When a floating-point exception occurs, the hardware records the exception by setting the appropriate bit in the FPECR and takes the SFU1 exception. The software envelope then determines if the floating-point exception can be mapped into the IEEE exception conditions as shown in Figure 4-6.

The software envelope signals an IEEE exception condition to the user by either causing a branch (software envelope passes parameters to the system software so that the system software actually performs the branch) to the user exception handler for that condition if it is enabled, or by setting the accumulated flag in the FPSR and returning the IEEE default result. The software envelope first checks the FPCR to see if the corresponding user exception handler is enabled. If the user handler is enabled, then information for the branch is passed to the system software. If the user handler is disabled, the software envelope sets the appropriate accumulated flag in the FPSR and then calculates the IEEE designated result and returns this result to the destination register of the instruction that generated the SFU1 exception.

The following paragraphs discuss the eight floating-point exceptions that generate the SFU1 exception (each one having a corresponding bit in the FPECR), the conditions that cause them, and how the software envelope responds to them. For a complete description of the software envelope and its interaction with the system software, refer to the *MC88110 Floating-Point Exception Envelope (FPEE) User's Guide* .

4.3.2.1 FLOATING-POINT UNIMPLEMENTED INSTRUCTION. When this floating-point exception occurs, the hardware sets the FUNIMP bit in the FPECR and takes the SFU1 exception. This floating-point exception does not directly map into the IEEE exception conditions; therefore, there are no corresponding accumulated flags to be set or user handlers to be enabled. The causes of this floating-point exception and the manner in which the software envelope responds to each are as follows:

1. If a floating-point operation is attempted when SFU1 is disabled (see **Section 2 Programming Model**), the software envelope signals to the system software that SFU1 is disabled.

2. If there is an attempt to execute the **fsqrt** instruction, the software envelope calculates the square root and returns the result to the destination register. If an IEEE exception condition is encountered while calculating the square root, then the software envelope checks the appropriate FPCR bit and branches to the user handler if the user trap is enabled. If the user trap is disabled, the software envelope sets the appropriate accumulated flag in the FPSR.

3. If there is an attempt to execute an unimplemented floating-point opcode, the software envelope signals to the system software that an unimplemented opcode was attempted to be executed.

4. If there is an attempt from supervisor mode to access an unimplemented floating-point control register, the software envelope signals to the system software that an access violation was attempted.

5. If there is an attempt to access a double-precision floating-point number which is aligned on an odd-numbered register pair (i.e., **r5:r6** instead of **r4:r5**) in the general register file, the software envelope transfers the operands to an even register pair, performs the operation, and returns the result to the destination register. If an IEEE exception condition is encountered while the software envelope is performing these actions, then it checks the appropriate FPCR bit and branches to the user handler if it is enabled. If the user handler is disabled, the software envelope sets the appropriate accumulated flag in the FPSR.

4.3.2.2 FLOATING-POINT PRIVILEGE VIOLATION. This exception occurs whenever there is an attempt to access a privileged (implemented or unimplemented) floating-point control register from user mode. When this happens, the hardware sets the FPRV bit in the FPECR and takes the SFU1 exception. The software envelope then signals to the system software that a privilege violation was attempted. This floating-point exception does not map into the IEEE exception conditions; therefore, there are no corresponding accumulated flags to be set or user handlers to be enabled. Note that the only floating-point registers that are not privileged are the FPSR (**fcr62**) and the FPCR (**fcr63**). The FPECR (**fcr0**) and the unimplemented floating-point registers (**fcr1–fcr61**) are all privileged.

4.3.2.3 FLOATING-POINT TO INTEGER CONVERSION OVERFLOW. This exception occurs when the source operand of a floating-point to integer conversion operation (**int**, **nint**, or **trnc** instruction) is too large to be represented as a signed 32-bit integer. When this happens, the hardware sets the FIOV bit in the FPECR and takes the SFU1 exception. The software envelope then maps this exception to the IEEE invalid operation exception condition. If the EFINV bit in the FPCR is set, the system software is notified so that it can perform a branch to the user handler. If the EFINV bit is clear, the software envelope sets the AFINV bit in the FPSR and returns the processor to normal instruction execution.

4

4.3.2.4 FLOATING-POINT RESERVED OPERAND. When this floating-point exception occurs, the hardware sets the FROP bit in the FPECR and takes the SFU1 exception. The causes of this floating-point exception and the manner in which they are handled by the software envelope are listed below and encompass both reserved operand conditions and invalid operation conditions. Notice that causes 1, 2, and 3 are resolved by the software envelope without mapping into IEEE exception conditions; therefore, there are no corresponding accumulated flags set or user handlers to be taken. Note also that the MC88110, unlike the MC88100, does not treat infinity as a reserved operand. Infinity arithmetic is performed directly in hardware except in the invalid operation cases described in cause 4.

1. If a nonsignaling NaN is specified as a source operand for any instruction which returns a floating-point quantity, the software envelope returns a nonsignaling NaN to the destination register as defined by the IEEE standard and the *MC88110 Floating-Point Exception Envelope (FPEE) User's Guide.*

2. If a nonsignaling NaN is specified as a source operand for **fcmp**, then the software envelope returns the result string, with all of the unordered bits set, to the destination register.

3. If a denormalized number or an unnormalized number is specified as a source operand, the software envelope performs the operation and returns the result to the destination register.

4. If any of the following occur:
   (a) signaling NaN is specified as a source operand
   (b) the four combinations of magnitude subtraction of infinities ($\infty - \infty$, $-\infty + \infty$, $\infty + (-\infty)$, and $\infty - (-\infty)$)
   (c) the multiplication of ($0 \times \infty$)
   (d) the division of ($0/0$) or (∞/∞)
   (e) nonsignaling NaN is specified as a source operand for **fcmpu**,
   (f) NaN is specified as a source for **int**, **nint**, or **trnc** instruction
   the software envelope maps this floating-point exception to the IEEE invalid operation exception condition. If the EFINV bit in the FPCR is set, a branch is caused (by signaling the system software) to the user handler. If the EFINV bit in the FPCR is clear, the software envelope sets the AFINV bit in the FPSR and delivers the IEEE designated result (the universal nonsignaling NaN) to the destination register of the instruction that caused the SFU1 exception. Note that in the case (f) above, there is no IEEE designated result and so the results of the destination register are unchanged.

4.3.2.5 FLOATING-POINT OVERFLOW. This exception occurs when the rounded result of an operation is too large to be represented as a finite normalized number in the destination format. The actions taken by the hardware and the software envelope when a floating-point overflow exception occurs are shown in Figure 4-10.

FLOATING-POINT
OVERFLOW EXCEPTION
DETECTED

HARDWARE SETS
FOVF AND FINX BITS IN FPECR

IF IEEE INEXACT CONDITION,
HARDWARE SETS AFINX BIT IN
FPSR

PROCESSOR TAKES
SFU1 EXCEPTION

HARDWARE

SOFTWARE ENVELOPE   SYSTEM SOFTWARE

RESULT
RE-COMPUTED

SCALE SOURCE OPERANDS;
PERFORM OPERATION;
(A) ← UNSCALED RESULT

EFOVF
IN FPCR
SET?   — YES →

NO

COMPUTE IEEE SCALED RESULT;
DATA BLOCK ← SCALED RESULT;
DATA BLOCK ← OVERFLOW TRAP STATUS CODE

RETURN TO
SYSTEM SOFTWARE

BRANCH TO OVERFLOW
USER HANDLER AND
RESUME PROCESSING

EFINX
IN FPCR
SET?   — YES →

NO

DATA BLOCK ← (A);
DATA BLOCK ← INEXACT TRAP STATUS CODE

RETURN TO
SYSTEM SOFTWARE

BRANCH TO INEXACT
USER HANDLER AND
RESUME PROCESSING

MAP OVERFLOW TO IEEE
OVERFLOW AND IEEE INEXACT
EXCEPTION CONDITIONS

SET AFOVF AND AFINX
BITS IN FPSR

WRITE-BACK
IEEE DESIGNATED RESULT

RESUME PROCESSING

**Figure 4-10. Default Floating-Point Overflow
Algorithm for Software Envelope**

When a floating-point overflow condition is detected, the MC88110 sets both the FOVF and FINX bits of the FPECR and takes the SFU1 exception. The software envelope then scales the source operands appropriately so that the operation can be performed without causing an overflow exception. The software envelope then recomputes the original operation. The result is then unscaled so that the overflow result is generated (without causing the exception).

If the EFOVF bit is set in the FPCR, then the software envelope scales the unscaled result (by subtracting either 192 (single-precision), 1536 (double-precision) or 24576 (double-extended-precision) to the exponent of the result) and writes it to a predefined data block in memory. This data block is the mechanism used to transfer parameters to the system software. In addition, an overflow trap status code is also written to the data block. Finally, the software envelope returns to the system software, which then should branch to the user handler for overflow.

If the EFOVF bit is not set in the FPCR, the software checks the value of the EFINX bit in the FPCR. If the EFINX bit is set, then the unscaled result recomputed by the software envelope is written to the data block. In addition, an inexact trap status code is also written to the data block. Finally, the software envelope returns to the system software, which then should branch to the user handler for inexact.

If the EFINX bit is not set in the FPCR, then the software envelope maps the floating-point exception for overflow into both the IEEE overflow and IEEE inexact exception conditions and sets both the AFOVF and AFINX bits in the FPSR. The IEEE-designated result is then written to the destination register and the original program flow continues.

The IEEE designated result is based on the rounding mode (as set in the FPCR) and the sign of the intermediate result as follows:

1. Round-to-nearest rounds all overflows to infinity with the sign of the intermediate result.
2. Round-toward-zero rounds all overflows to the format's largest finite number with the sign of the intermediate result.
3. Round-toward-negative infinity carries positive overflows to the format's largest finite number and rounds negative overflows to negative infinity.
4. Round-toward-positive infinity rounds negative overflows to the format's most negative finite number and rounds positive overflows to positive infinity.

4.3.2.6 FLOATING-POINT UNDERFLOW. This exception occurs when the rounded result of an operation is too small to be represented as a finite normalized number in the destination format. The actions taken by the hardware and the software envelope when a floating-point underflow exception occurs are shown in Figure 4-11.

FLOATING-POINT
UNDERFLOW EXCEPTION
DETECTED

HARDWARE SETS
FUNF BIT IN FPECR

IF IEEE INEXACT CONDITION,
HARDWARE SETS AFINX BIT IN
FPSR

HARDWARE

SOFTWARE ENVELOPE

SYSTEM SOFTWARE

RESULT
RE-COMPUTED

SCALE SOURCE OPERANDS;
PERFORM OPERATION;
Ⓐ ➡ UNSCALED RESULT

EFUNF
IN FPCR
SET?

YES

NO

CLEAR AFINX BIT
IN FPSR

COMPUTE IEEE SCALED RESULT;
DATA BLOCK ➡ SCALED RESULT;
DATA BLOCK ➡ UNDERFLOW TRAP STATUS CODE

RETURN TO
SYSTEM SOFTWARE

BRANCH TO UNDERFLOW
USER HANDLER AND
RESUME PROCESSING

LOSS
OF ACCURACY?
(AFINX BIT IN
FPSR SET?)

YES

NO

EFINX
IN FPCR
SET?

YES

NO

CLEAR AFINX BIT
IN FPSR

MAP UNDERFLOW TO IEEE
UNDERFLOW AND IEEE INEXACT
EXCEPTION CONDITIONS

DATA BLOCK ➡ Ⓐ;
DATA BLOCK ➡ INEXACT TRAP STATUS CODE

SET AFUNF
BIT IN FPSR

RETURN TO
SYSTEM SOFTWARE

BRANCH TO INEXACT
USER HANDLER AND
RESUME PROCESSING

WRITE-BACK
DENORMALIZED RESULT

RESUME PROCESSING

**Figure 4-11. Default Floating-Point Underflow
Algorithm for Software Envelope**

When a floating-point underflow condition is detected, the MC88110 sets the FUNF bit in the FPECR and takes the SFU1 exception. The software envelope then scales the source operands appropriately so that the operation can be performed without causing an underflow exception. The software envelope then recomputes the original operation. The result is then unscaled so that the underflow result is generated (without causing the exception).

If the EFUNF bit is set in the FPCR and precision was lost, then the software envelope clears the AFINX bit in the FPSR (if it was set), scales the unscaled result (by adding either 192 (single-precision), 1536 (double-precision) or 24576 (double-extended-precision) to the exponent of the result), and writes the scaled result to a predefined data block in memory. This data block is the mechanism used to transfer parameters to the system software. In addition, an underflow trap status code is also written to the data block. Finally, the software envelope returns to the system software, which then should branch to the user handler for underflow.

If the EFUNF bit is not set in the FPCR, the software checks to see if a loss of accuracy has occurred. If it has, the software envelope checks the value of the EFINX bit in the FPCR. If the EFINX bit is set, then the AFINX bit in the FPSR is cleared (if it was set) and the unscaled result recomputed by the software envelope is written to the data block. In addition, an inexact trap status code is also written to the data block. Finally, the software envelope returns to the system software, which then should branch to the user handler for inexact.

If the EFINX bit is not set in the FPCR, then the software envelope maps the floating-point exception for underflow into both the IEEE underflow and the IEEE inexact exception conditions and sets the AFUNF bit in the FPSR. Finally, the IEEE-designated result is written to the destination register and the original program flow continues. This last step also occurs when a loss of accuracy has not occurred.

4.3.2.7 FLOATING-POINT DIVIDE-BY-ZERO. This exception occurs when the denominator of a floating-point divide operation is zero and the numerator is a nonzero finite normalized number. When this happens, the hardware sets the FDVZ bit in the FPECR and takes the SFU1 exception. The software envelope then maps this floating-point exception to the IEEE divide-by-zero exception condition. If the EFDVZ bit in the FPCR is set, a branch is made to the user handler. If the EFDVZ bit is clear, the software envelope sets the AFDVZ bit in the FPSR and delivers the IEEE designated result to the destination register.

4.3.2.8 FLOATING-POINT INEXACT. If the result of a floating-point operation is not exact (e.g., due to loss of accuracy caused by rounding or loss of significance caused by overflow), the hardware checks the EFINX bit in the FPCR. If the EFINX bit is clear, the hardware does not take the SFU1 exception, but it signals the condition by setting the AFINX bit in the FPSR. If the EFINX bit is set, the hardware sets the FINX bit in the FPECR and takes the SFU1 exception. The software envelope then maps this floating-point exception to the IEEE inexact exception condition and a branch is made to the user handler.

4.3.3 Time-Critical Floating-Point (TCFP) Mode

Time-critical floating-point (TCFP) mode is the alternative to the default operation (out of reset) of the MC88110, which takes the SFU1 exception for IEEE exception conditions. In TCFP mode, default results are generated directly in hardware rather than taking an SFU1 exception when an IEEE exception condition occurs. TCFP mode avoids SFU1 exceptions for all but two of the floating-point exceptions (floating-point unimplemented instruction and floating-point privilege violation).

TCFP mode is selected by three control bits in the FPCR (see Figure 4-8). Setting the TCFPUNF bit selects TCFP mode operation when an IEEE underflow exception condition occurs. Setting the TCFPOVF bit selects TCFP mode operation when an IEEE overflow exception condition occurs. Setting the TCFP mode bit selects TCFP mode operation for all IEEE exception conditions regardless of the values of TCFPUNF and TCFPOVF.

The following paragraphs describe the eight floating-point exceptions and the actions taken in TCFP mode by the hardware and the software envelope when they occur.

NOTE

The eight floating-point exceptions referenced here are defined as those events that cause the SFU1 exception to occur when the MC88110 is *not* operating in TCFP mode. Although six of the eight conditions do not cause MC88110 exception processing to occur when operating in TCFP mode, they are still defined as exception conditions.

4.3.3.1 FLOATING-POINT UNIMPLEMENTED INSTRUCTION IN TCFP MODE. Since this floating-point exception does not directly map into the IEEE exception conditions, the hardware takes the SFU1 exception in TCFP mode when this floating-point exception occurs. The causes of this floating-point exception and the manner in which the software envelope responds to each are as follows:

1. If a floating-point operation is attempted when SFU1 is disabled (see **Section 2 Programming Model**), the software envelope signals to the system software that SFU1 is disabled.

2. If there is an attempt to execute the **fsqrt** instruction, the software envelope calculates the square root and returns the result to the destination register. If an IEEE exception condition is encountered while calculating the square root, then the TCFP mode default result for that condition is delivered to the destination register.

3. If there is an attempt to execute an unimplemented floating-point opcode, the software envelope signals to the system software that an unimplemented opcode was attempted to be executed.

4. If there is an attempt from supervisor mode to access an unimplemented floating-point control register, the software envelope signals to the system software that an access violation was attempted.

5. If there is an attempt to access a double-precision floating-point number which is aligned on an odd-numbered register pair (i.e., **r5:r6** instead of **r4:r5**) in the general register file, the software envelope transfers the operands to an even register pair, performs the operation, and returns the result to the destination register. If an IEEE exception condition is encountered while the software envelope is performing these actions, then the TCFP mode result for that condition is delivered to the destination register.

4.3.3.2 FLOATING-POINT PRIVILEGE VIOLATION IN TCFP MODE. The hardware and the software envelope carry out the same actions for this floating-point exception in TCFP mode as when not in TCFP mode (see **4.3.2.2 Floating-Point Privilege Violation**).

4.3.3.3 FLOATING-POINT TO INTEGER CONVERSION OVERFLOW IN TCFP MODE. This exception occurs when the source operand of a floating-point to integer conversion operation (**int, nint,** or **trnc** instruction) is too large to be represented as a signed 32-bit integer. When this happens, the hardware delivers the large properly signed integer (see Table 4-3) to the destination register instead of taking the SFU1 exception.

4.3.3.4 FLOATING-POINT RESERVED OPERAND IN TCFP MODE. The causes of this exception and the default results provided by the hardware instead of taking the SFU1 exception are as follows:

1. If a denormalized number, unnormalized number, signaling NaN, or nonsignaling NaN is specified as a source operand for an add or subtract operation, then the universal positive nonsignaling NaN (see Table 4-3) is delivered to the destination register.

2. If a denormalized number, unnormalized number, signaling NaN, or nonsignaling NaN is specified as a source operand for a multiply or divide operation, then the universal properly signed nonsignaling NaN is delivered to the destination register. The sign bit of the result is the exclusive-OR of the sign bits for the two source operands.

3. If a denormalized number, unnormalized number, signaling NaN, or nonsignaling NaN is specified as an operand for a compare instruction, then the result string with all of the unordered bits set is delivered to the destination register.

4. If a signaling NaN or nonsignaling NaN is specified as the source operand for a floating-point to integer conversion operation, then the large properly signed integer (see Table 4-3) is delivered to the destination register.

5. When an invalid operation ($\infty - \infty$, 0 x ∞, ∞/∞, or 0/0) is attempted, the universal nonsignaling NaN (see Table 4-3) is delivered to the destination register.

Table 4-6 summarizes the values generated by the MC88110 for the cases when the FROP bit is set in the FPECR (reserved operand exception) in TCFP mode.

Table 4-6. Results for Reserved Operand Exception in TCFP Mode

Operand(s)	Compare	Convert to Int.	Add/Sub	Mul/Div
±Signaling NaN	Unordered	Large ± Integer	Universal +Non-Signaling NaN	Universal ±Non-Signaling NaN
± Non-Signaling NaN	Unordered	Large ± Integer	Universal +Non-Signaling NaN	Universal ±Non-Signaling NaN
± Unnormalized	Unordered	0	Universal +Non-Signaling NaN	Universal ±Non-Signaling NaN
± Denormalized	Unordered	0	Universal +Non-Signaling NaN	Universal ±Non-Signaling NaN
Invalid ($\infty-\infty$, $0\times\infty$, ∞/∞, $0/0$)	N/A	N/A	Universal +Non-Signaling NaN	Universal ±Non-Signaling NaN

NOTE: For conversion to integer, the sign of the result is the same as the sign of the source operand. For addition and subtraction the result is correctly signed except for nonsignaling NaNs, which are always positive. For multiplication and division the result is always correctly signed—i.e., it is the exclusive-OR of the sign bits of the two source operands.

4.3.3.5 FLOATING-POINT OVERFLOW IN TCFP MODE. This exception occurs when the rounded result of an operation is too large to be represented as a finite normalized number in the destination format. When this happens in TCFP mode, the hardware delivers the properly signed infinity to the destination register instead of taking the SFU1 exception.

4.3.3.6 FLOATING-POINT UNDERFLOW IN TCFP MODE. This exception occurs when the rounded result of an operation is too small to be represented as a finite normalized number in the destination format. When this happens in TCFP mode, the hardware delivers the properly signed zero to the destination register instead of taking the SFU1 exception.

4.3.3.7 FLOATING-POINT DIVIDE-BY-ZERO IN TCFP MODE. This exception occurs when the denominator of a floating-point divide operation is zero and the numerator is a nonzero finite normalized number. When this happens in TCFP mode, the hardware delivers the properly signed infinity to the destination register instead of taking the SFU1 exception.

4.3.3.8 FLOATING-POINT INEXACT IN TCFP MODE. This exception occurs when the result of a floating-point operation is not exact (e.g., due to loss of accuracy caused by rounding or loss of significance caused by overflow). When this happens in TCFP mode, the hardware delivers the properly signed inexact result to the destination register instead of taking the SFU1 exception.

4

SECTION 5
GRAPHICS UNIT IMPLEMENTATION

The MC88110 provides dedicated instructions (executed by on-chip execution units) to accelerate the processing of special-purpose data types for graphics operations. The graphics instructions are optimized to support pixel-oriented graphics operations, such as bit-mapped display functions and three-dimensional (3D) graphics rendering algorithms. The graphics processing unit (GPU) is implemented as special function unit two (SFU2), and it is specific to the MC88110; thus it may not be supported in the same manner in future 88000 implementations.

This section describes the various operations performed by the GPU and discusses how they can be applied to accelerate fundamental two-dimensional (2D) and 3D graphics operations. The forming of useful primitive operations by combining sequences of instructions is described in this section, and examples are shown of how those primitive operations may be used in some common graphics algorithms. However, the user has the flexibility to use the instructions and algorithms that best fit the application rather than being restricted to a particular set of predefined graphics algorithms.

Data types and the behavior of specific instructions are described within the context of example graphics algorithms in this section; the complete definition of the graphics instructions is provided in **Section 10 Instruction Set.** The detailed timing for the execution of the graphics instructions is provided in **Section 9 Instruction Timing and Code Scheduling Considerations.**

5.1 GPU OVERVIEW

The operation of the GPU is architecturally compatible with all other MC88110 operations in that operands reside in the general register file and data movement to and from memory is performed using load and store instructions. Graphics instructions, which can be issued two at a time, can be intermixed with other integer and floating-point instructions with no restrictions on instruction alignment. Data dependencies are detected and interlocked by the same register scoreboard that is used for all other instructions.

The graphics functionality of the MC88110 extends beyond support for incremental drawing and shading algorithms, which is provided by multipixel add and subtract instructions. The multipixel add and subtract instructions also have saturation arithmetic variations with the ability to specify either maximum (or minimum) field values or user-defined saturation limits. The multipixel add and subtract instructions are supplemented by pixel pack and unpack instructions, which facilitate efficient storing, retrieving, and manipulation of images stored in a packed pixel format such as a frame buffer.

In a typical interactive graphics system, displays are composed of preexisting background objects, objects being created on the screen, and objects held in memory (such as fonts) for rapid transfer to the screen. In a complex environment, any or all of these objects may require anti-aliasing and/or be partially transparent. Combining these objects into a single image is typically performed by a process called compositing, in which object images are blended together rather than being tiled or overlaid.

To perform compositing, every pixel of every image must be multiplied by a value representing its level of transparency. The MC88110 provides the capability to perform interactive compositing of images with the pixel multiply instruction, which can perform a parallel multiply of each individual red-green-blue (RGB) intensity component of a pixel in one instruction.

Table 5-1 summarizes the MC88110 graphics instructions and the options available for each instruction.

Table 5-1. Graphics Instructions

Instruction	Name	Operand Syntax	Operation
padd.t	Pixel Add	rD,rS1,rS2	rD:rD+1 ← rS1:rS1+1 + rS2: rS2+1 modulo 2^t add
padds.x.t	Pixel Add and Saturate	rD,rS1,rS2	rD:rD+1 ← rS1:rS1+1 + rS2: rS2+1 modulo 2^t add and saturate; **x** specifies signed, unsigned, or mixed arithmetic
pcmp	Z-Compare	rD,rS1,rS2	rD ←rS1:rS1+1 :: rS2: rS2+1
pmul	Pixel Multiply	rD,rS1,rS2	rD:rD+1 ← rS1 * rS2: rS2+1
ppack.r.t	Pixel Truncate, Insert, and Pack	rD,rS1,rS2	rD:rD+1 ← fields of size t from rS2: rS2+1 truncated to t*r/64, packed together, and concatenated with rS1:rS1+1 rotated left by r bits
prot	Pixel Rotate	rD,rS1,<O6> rD,rS1,rS2	rD:rD+1 ← rS1:rS1+1 rotated left by rS2 or O6 bits; rS2 or O6 should be an even multiple of 4
psub.t	Pixel Subtract	rD,rS1,rS2	rD:rD+1 ← rS1:rS1+1 – rS2: rS2+ modulo 2^t subtract
psubs.x.t	Pixel Subtract and Saturate	rD,rS1,rS2	rD:rD+1 ← rS1:rS1+1 – rS2: rS2+1 modulo 2^t subtract and saturate; **x** specifies signed, unsigned, or mixed arithmetic
punpk.t	Pixel Unpack	rD,rS1	rD:rD+1 ← fields of size t from rS1 are put in fields of size 2t with zero fill and placed in rD:rD+1

All of these instructions (with the exception of **pmul**) are executed by the pixel add unit or the pixel pack/unpack unit of the MC88110, each of which operates completely independently. Each of these execution units executes instructions in a single clock and can accept a new instruction on every clock. The **pmul** instruction is executed in the multiply execution unit of the MC88110. It is subject to the same issue restrictions and latency times as all other multiply instructions.

No control registers are associated with the GPU. The only exception generated as the result of executing a graphics instruction is the graphics unimplemented opcode exception. This exception is generated if SFU2 is disabled (bit 4 of PSR set; see **Section 2 Programming Model**) and execution of an SFU2 instruction is attempted. This exception also occurs if an odd register is specified for a double-word operand, or if execution of any undefined SFU2 instruction is attempted. Refer to **Section 7 Exceptions** for a more detailed description of exception processing.

5.2 GRAPHICS DATA TYPES

Operands for all graphics instructions are located in the general register file, providing the graphics instructions with the same register flexibility as all other instructions. All graphics instructions have the capability to operate on 64-bit values, which allows multiple pixels to be processed with a single instruction. Double-word operands used in graphics instructions must be aligned in even-numbered register pairs (e.g., **r4:r5** rather than **r5:r6**) with the first register (the even one) specifying the register pair in the instruction syntax.

Graphics data is represented by packed bit fields that are 4, 8, 16, or 32 bits wide to reduce storage and memory bandwidth requirements. The MC88110 graphics instructions operate in parallel on the individual pixel fields packed into a 64-bit double-word value. This parallel operation on packed pixels avoids the need to extract the individual fields from the data structures for performing many graphics operations.

The following paragraphs summarize the organization of data in general registers for the MC88110 and describe how the specific options of the graphics instructions use the general registers to manipulate some common graphics data types.

5.2.1 General Data Types

Figure 5-1 depicts the general data types for packed bit fields that are supported by the MC88110. It shows how the MC88110 interprets packed nibbles, packed bytes, packed half-words, and packed words. The width of the fields for pixel add/subtract or pack/unpack instructions is defined by the **t** value specified with the instruction syntax (**.n** for nibble, **.b** for byte, **.h** for half-words, and blank for word). These fields can be represented as signed or unsigned integers, fractional values, or, in the case of the **pcmp** instruction, floating-point values as an arbitrary convention chosen by the user to simplify the implementation of data structures.
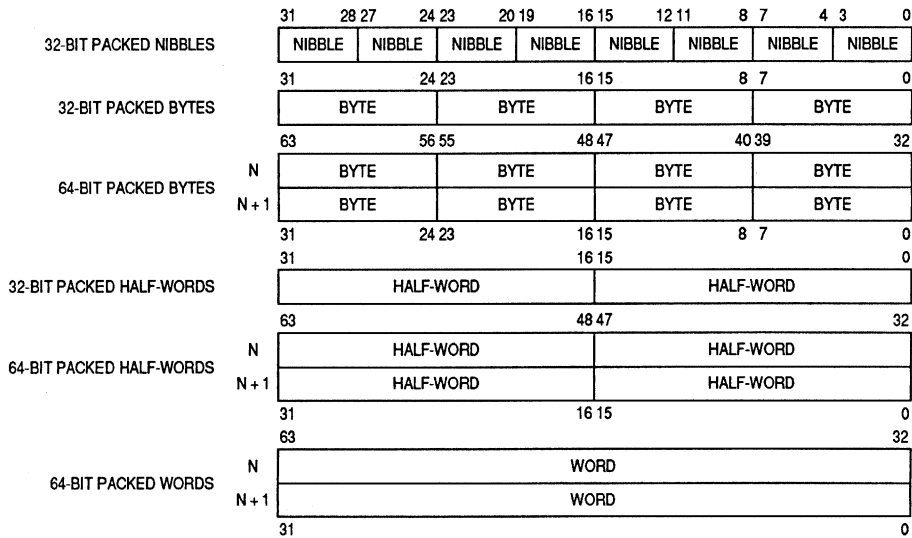
Figure 5-1. Packed Data Organization in General Registers

The pixel add/subtract instructions perform integer operations on fields of either 8, 16, or 32 bits contained in a register pair. Although 4-bit fields may not be explicitly specified by the pixel add/subtract instructions they can be first unpacked into 8-bit fields and then added/subtracted.

The pixel pack/unpack instructions operate on data fields that are 4, 8, or 16 bits wide within a register pair. Because fields of 32 bits are easily manipulated by other 88000 instructions, the graphics instructions require no additional support for pixel packing or unpacking operations.

5.2.2 Fixed-Point Data Type Definition

Graphics data is always treated as unsigned integers by the MC88110, but it is often convenient to assign it other forms, at least conceptually. A common practice is to assign a binary point to an arbitrary bit location, treating the value as a fixed-point number with both an integer and fractional part. For example, Figure 5-2 shows a 32-bit value specified as an 8.24 fixed-point number, consisting of an 8-bit integer part in bit locations 31–24 and a 24-bit fractional part in bit locations 23–0. Unsigned integer operations carried out on fixed-point numbers of this type always give correct results, regardless of the location of the conceptual binary point.
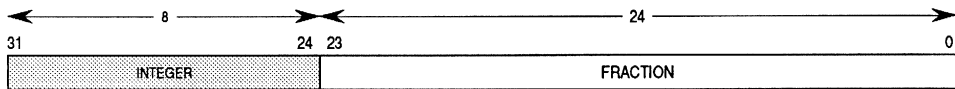
Figure 5-2. Example 32-Bit Fixed-Point Number (8.24)

Fixed-point representation is useful for maintaining higher precision intermediate values when interpolating between endpoints on a scan line. This prevents quantization errors from propagating and causing visible artifacts when the results are written out to the frame buffer.

When using the **pcmp** instruction, operand values may be represented as either 32-bit unsigned integers or positive single-precision floating-point values, depending upon the dynamic range required. Although the **pcmp** instruction performs a comparison using fixed-point unsigned arithmetic, the bit pattern maintains the same relative order when single-precision positive IEEE floating-point values are interpreted as unsigned integers in this context.

5.2.3 Other Common Data Types

The MC88110 allows a wide variety of formats to be used for graphics data structures. Data used by the GPU is generally interpreted as either of pixel type (visual image information represented) or of number type (numerical light intensity value represented). Figure 5-3 illustrates some of the more commonly used data types which can be implemented by the MC88110; a brief explanation of pixel and number types is presented in the paragraphs that follow.

5
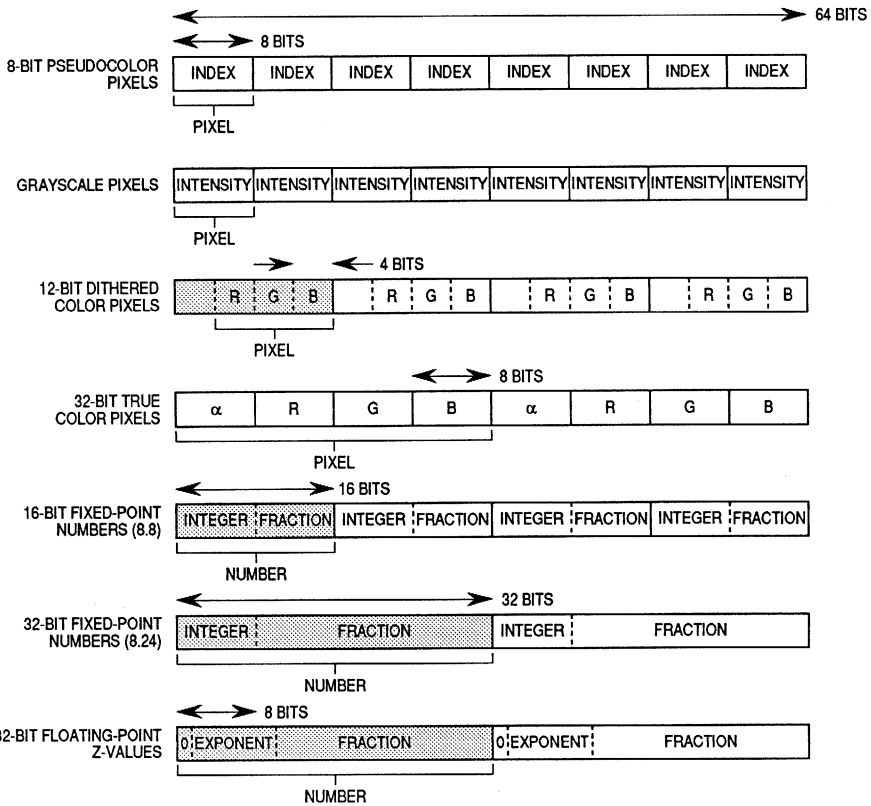
Figure 5-3. Common Graphics Data Types

5.2.3.1 PIXEL TYPES. Pseudocolor pixels are multibit values used to index a lookup table that maps them to an RGB color value. The 8-bit value shown in Figure 5-3 allows 256 different color values to be represented by each pixel. Grayscale values represent a fractional intensity value between 0 and 1. For example, 1 can be a white pixel and 0 can be a black pixel. An 8-bit grayscale value provides 256 shades of gray.

True color pixels provide a separate 8-bit field for the intensity of each of the RGB color components. These can be direct intensity values and are large enough to satisfactorily represent all colors in most situations. Some applications also include a fourth field, called alpha (α), which represents either the transparency or the relative coverage of an object over the pixel. If used to represent transparency, α is a fractional value between 0 and 1. (e.g., 0 can indicate totally transparent and 1 can indicate totally opaque). If used to represent relative coverage, 0 can indicate no contribution to the pixel and 1 can indicate that the pixel color is completely determined from the associated RGB value— i.e., the object completely covers the pixel. Intermediate values of α indicate that the RGB value must be combined with another RGB value to determine the final color of the pixel. See **5.4.3 Intensity Scaling** and **5.5.4 Compositing** for further explanation of the α field.

MC88110 USER'S MANUAL   MOTOROLA

For applications where system cost is more critical than color realism, dithered color pixels can substantially reduce memory size and bus bandwidth requirements relative to true color pixel representation. As can be seen in Figure 5-3, each color channel of the dithered color pixel is represented by a 4-bit value, rather than the 8-bit value associated with the true color data type.

5.2.3.2 NUMBER TYPES. Fixed-point numbers are a convenient representation when using integers in graphics operations, particularly when performing scaling and conversion operations with graphics instructions. Figure 5-3 illustrates two fixed-point formats, 8.8 and 8.24, which can be used as intermediate representations of 8-bit pixels requiring different degrees of precision. The decimal point may be placed anywhere in the bit field of fixed-point numbers since it is merely a data abstraction used for the convenience of the programmer.

Floating-point number representations can be used as Z-buffer values because integer operations on IEEE single-precision numbers yield correct results in this context (see **5.2.2 Fixed-Point Data Type Definition**).

5.3 GRAPHICS INSTRUCTIONS

The following paragraphs provide an overview of the functionality provided by the graphics instructions of the MC88110. Refer to **Section 10 Instruction Set** for a complete definition of the graphics instructions.

5.3.1 Pixel Add/Subtract Operations

The possibility exists that, when adding and subtracting fields, the final result will overflow (or underflow) the destination field size. For example, adding a 75% intensity value to a 50% intensity value would cause the most significant bits to be lost, resulting in a 25% intensity value. This could produce an unacceptable visual anomaly in the resultant image if the values were representing color intensity. Therefore, it is more appropriate for the addition operation to clamp or saturate at the maximum intensity representable by the field, resulting in a more visually acceptable 100% intensity for this example.

The mathematics used in many graphics algorithms automatically precludes the possibility of overflow and therefore does not require saturation arithmetic. An example is a shading interpolation between two known points, where no overflow is possible. Other algorithms depend on the wraparound nature of modulo arithmetic and also require nonsaturating arithmetic. Still other algorithms perform intermediate calculations that may overflow, but the final operation does not. Thus, some calculations need to be performed with modulo arithmetic and some with saturation arithmetic.

The pixel add/subtract instructions of the MC88110 are executed by a dual 32-bit adder with controllable carry chains on each 8-bit boundary, allowing multiple add operations to be performed in parallel. Arithmetic is executed using either modulo arithmetic using the **padd** and **psub** instructions or saturation arithmetic using the **padds** and **psubs**

instructions. Saturation arithmetic is performed by substituting the appropriate maximum or minimum value for any field that overflows or underflows. The pixel compare instruction, **pcmp**, is also executed in the pixel add unit.

Without saturation (using the **padd** and **psub** instructions), the arithmetic is performed modulo [destination bit-field size], and bits that overflow (i.e., carry or borrow out of) the destination field are lost.

The following paragraphs describe the three types of saturation arithmetic that are provided to handle various data representations: 1) unsigned ± unsigned = unsigned, 2) signed ± signed = signed, and 3) unsigned ± signed = unsigned. Overflow (underflow) detection and the maximum (minimum) field value is different in all three cases. Whether data is signed or unsigned is a data abstraction only and does not affect any graphics operation except for saturation. In addition, the setting of user-defined saturation limits is discussed.

5.3.1.1 TYPES OF SATURATION. Saturation arithmetic results are clamped at a given value, depending upon how the source operands are specified in the instruction syntax. The actual binary arithmetic performed in all saturation forms is identical to the arithmetic performed in the nonsaturation form. The differences between the various forms of saturation are defined by the method used to detect overflow or underflow in a field and the value substituted for the result when a field overflows or underflows as described below:

Unsigned ± unsigned = unsigned: saturation occurs if there is a carry (or borrow in the case of subtraction) out of the most significant bit (MSB) of the sum. The maximum field value is 2^t-1, where t is the value of field size, and is substituted if an addition carries out. The minimum field value is 0 and is substituted if a subtraction borrows out.

Unsigned ± signed = unsigned: saturation occurs if the MSBs of the two source fields are different in sign and if the MSB of the signed field is the same as the MSB of the sum; for rS1+rS2 = rD, saturation = ((rS1[MSB] ∧ rS2[MSB]) ∧ !(rS2[MSB] ∧ rD[MSB])) where rS2 contains the signed field. The maximum field value is 2^t-1, where t is the value of field size, and is substituted if addition of a positive number or subtraction of a negative number saturates. The minimum field value is 0 and is substituted if addition of a negative number or subtraction of a positive number saturates.

Signed ± signed = signed: saturation occurs if the carry into the MSB of the sum is different than the carry out of the MSB of the sum. The maximum field value is $2^{(t-1)}-1$, where t is the value of field size, and is substituted if the sum does not carry out. The minimum field value is $-2^{(t-1)}$ and is substituted if the sum does carry out.

Table 5-2 lists some of the permutations possible when performing saturation arithmetic on 8-bit fields (t = 8).

Table 5-2. 8-Bit Saturation Examples

s1	s2	padd.b	padds.u.b	padds.s.b	padds.us.b
00	00	00	00	00	00
00	55	55	55	55	55
00	7F	7F	7F	7F	7F
00	80	80	80	80	00
00	AA	AA	AA	AA	00
00	FF	FF	FF	FF	00
55	00	55	55	55	55
55	55	AA	AA	7F	AA
55	7F	D4	D4	7F	D4
55	80	D5	D5	D5	00
55	AA	FF	FF	FF	00
55	FF	54	FF	54	54
7F	00	7F	7F	7F	7F
7F	55	D4	D4	7F	D4
7F	7F	FE	FE	7F	FE
7F	80	FF	FF	FF	00
7F	AA	29	FF	29	29
7F	FF	7E	FF	7E	7E
80	00	80	80	80	80
80	55	D5	D5	D5	D5
80	7F	FF	FF	FF	FF
80	80	00	FF	80	00
80	AA	2A	FF	80	2A
80	FF	7F	FF	80	7F
AA	00	AA	AA	AA	AA
AA	55	FF	FF	FF	FF
AA	7F	29	FF	29	FF
AA	80	2A	FF	80	2A
AA	AA	54	FF	80	54
AA	FF	A9	FF	A9	A9
FF	00	FF	FF	FF	FF
FF	55	54	FF	54	FF
FF	7F	7E	FF	7E	FF
FF	80	7F	FF	80	7F
FF	AA	A9	FF	A9	A9
FF	FF	FE	FF	FE	FE

5.3.1.2 USER-DEFINED SATURATION LIMITS. User-defined saturation limits allow a result to be kept within a certain range smaller than the normal field range. The saturation forms of pixel addition and pixel subtraction provided by the GPU can be used to synthesize this functionality.

A value can be clamped below a user-defined saturation level using the method described below (see Figure 5-4). The **padds.u** instruction can be used to add the difference between the user-defined upper saturation limit and the maximum field value. Then the **psubs.u** instruction can be used to subtract that difference. If the value being clamped was already below the user-defined upper saturation limit, then this operation would be a NOP and the result would be unchanged, shown by point A in Figure 5-4. However, if the value being clamped was above the user-defined saturation limit, then the first add operation would have saturated at the maximum field value, and the subtract operation would have set the result to the user-defined saturation limit value, as shown by point B. An analogous operation can be performed to clamp the value above a certain user-defined lower saturation level.
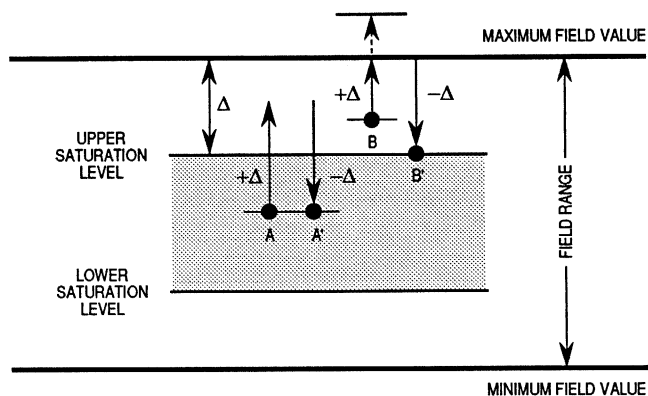
Figure 5-4. User-Defined Saturation Limits

5.3.2 Pixel Pack/Unpack Operations

The pixel pack/unpack instructions are executed by a specialized bit-field unit for packing, unpacking, and shifting pixel or fixed-point data. The pixel pack/unpack instructions operate in parallel on multiple bit fields within 64-bit operands. The pixel pack instruction (**ppack**) accumulates pixel data as it is computed by truncating multiple fixed-point values from the second source operand, packing the resulting bit fields together as specified, concatenating them with data from the first source operand, and rotating the result as specified. The **punpk** instruction performs the inverse operation by unpacking bit fields from a 32-bit operand and properly placing them into a 64-bit double word for subsequent arithmetic calculations.

Figures 5-5, 5-6, and 5-7 show representative examples of the **ppack, punpk,** and **prot** instructions, respectively. These operations are explained further in **5.4.2.1 Packing Pixels** and **5.4.2.2 Unpacking Pixels**.

The **ppack** instruction performs a format conversion from a high-precision intensity value to a low-precision pixel value. Intensity values can be 8, 16, or 32 bits and can be truncated to 4, 8, or 16 bits. The truncated values are concatenated to the least significant end of rD:rD+1, building a packed pixel value in raster order. Figure 5-5 shows 8/8/8/8 bit αRGB pixels being constructed from their high-precision, 32-bit, fixed-point values.
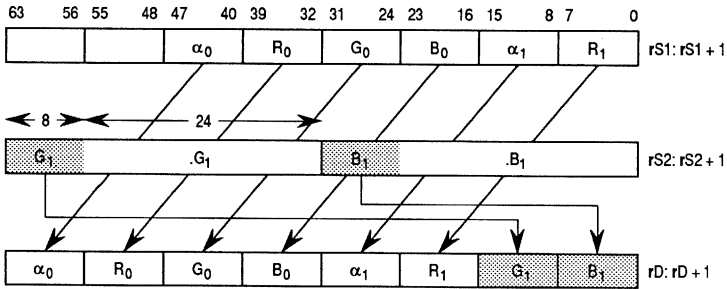
Figure 5-5. ppack.16 rD,rS1,rS2

The **punpk** instruction performs a format conversion from a low-precision packed pixel representation to a high-precision format, providing more dynamic range and allowing arithmetic calculations to be performed without overflow. The **punpk** instruction takes 8-, 16-, or 32-bit packed pixel fields and expands them into fields twice as large, right justified, and zero-filled. Figure 5-6 shows 8/8/8/8 bit αRGB packed pixels being unpacked into 16-bit fields.
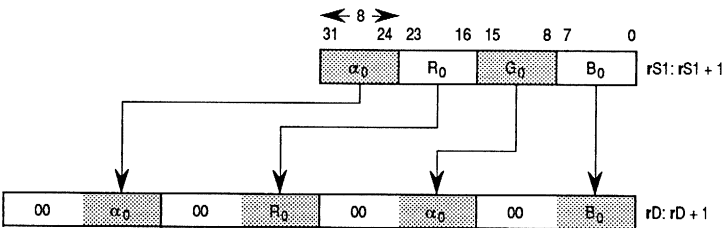
Figure 5-6. punpk.b rD,rS1

The **prot** instruction rotates a 64-bit value to any modulo-4 boundary between 0 and 60 bits. Any rotation count that is not a multiple of 4 is truncated to the next lower multiple of 4. Any count greater than 60 bits is truncated to be less than or equal to 60 in multiples of 4. Figure 5-7 illustrates eight 8-bit pixels being rotated by two pixels (16 bits).

The **prot** instruction is intended primarily for rotation of color pixels having a pixel depth of four bits or greater. Monochrome pixels or pixels represented by less than four bits can be rotated using the standard 88000 bit manipulation instructions, **mak, ext, extu,** and **rot**.
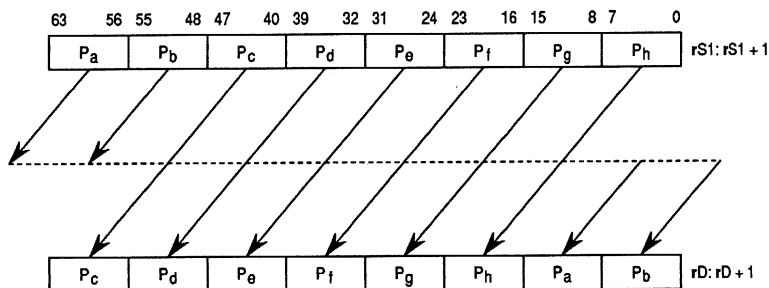
Figure 5-7. prot rD,rS1,<16>

5.3.3 Pixel Multiply Operation

The **pmul** instruction executes in the multiply execution unit. The multiply execution unit can accept one integer, one floating-point, or one pixel multiplication instruction every clock.

Unlike the pixel add unit, when carries occur out of each field in the multiplier, they are not prevented from affecting the next most significant field. The contents of register pair rS1:rS1+1 are multiplied by the contents of register rS2 as if they were full 64- and 32-bit numbers, respectively, as shown in Figure 5-8. Any bits lost as a result of truncating the product to 64 bits are discarded, with no indication of loss of significance. Note that Figure 5-8 shows an example with zeros in the upper bytes of each field. This is not a hardware requirement, but it is one way for the programmer to prevent the results of one multiply from overflowing and affecting the results of the next field. See **5.5.4 Compositing** for more information on how to use the **pmul** instruction.
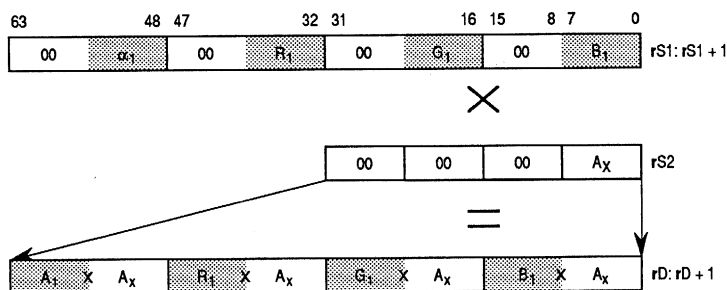
Figure 5-8. pmul rD,rS1,rS2

5.4 PRIMITIVE OPERATIONS

The GPU instructions are designed to be general enough to support a wide variety of graphical primitive operations, including rendering and shading operations. Since the graphics instructions were not intended to directly implement any particular set of algorithms, examples are provided to illustrate how to implement various graphics algorithms.

Graphics instructions perform four classes of operations: arithmetic (pixel add/subtract), format conversion (pixel pack/unpack), intensity scaling (pixel multiply), and coordinate comparison (pixel compare). The operation of the graphics instructions are explained in the following paragraphs in terms of the types of operations they perform in a graphics context. For a detailed description of each instruction's operation, see **Section 10 Instruction Set**.

5.4.1 Arithmetic Operations

Many graphics algorithms have been optimized for incremental modification, in which new values are generated by adding or subtracting an error term from a previous value. This type of optimization emphasizes addition and subtraction operations that generally execute quickly. The pixel add instructions can simultaneously perform several addition or several subtraction operations, further increasing the performance of these types of algorithms.

5.4.1.1 INTERPOLATION. Interpolation is a common operation used in incremental algorithms, such as Gouraud shading. The nonsaturation version of the **padd** instruction can be used for these types of algorithms. Shading interpolation can be performed by incrementally adding delta values to each of the RGB intensity components of sequential pixels, resulting in a linear change in color between a set of endpoints. Since the incremental delta values are computed from bounded endpoints, overflow cannot occur during the summing operation. A sample instruction sequence for Gouraud shading is described in section **5.5.1 Gouraud Shading**.

5.4.1.2 INTENSITY SUMMING. When calculating intensity at a point where there may be several light sources and their total contribution exceeds the largest value that can be represented, the saturation version of the **padd** instruction can be used to ensure that a visually acceptable result is obtained. In this case, **padds** can clamp the maximum value to either the maximum representable value or some other user-defined value as described in **5.3.1.2 User-Defined Saturation Limits**.

5.4.2 Format Conversion

Modeling or display list data is generally stored in a database with a high degree of precision. To ensure display fidelity, this computational precision must be maintained through the transformation and rendering operations until the image is ready to be displayed; then the pixel data is truncated to the depth of the frame buffer. The **ppack** instruction is used to truncate pixel data and pack it tightly into a format that matches the structure of the frame buffer.

Sometimes images are not generated from an internal data structure or display list, but are built from existing images already in the frame buffer or in memory in packed form. In this case, in order not to lose precision and generate objectionable artifacts in the final image, the source pixels must be expanded to a higher precision format. The **punpk** instruction can be used to unpack pixels from a frame buffer format and expand their precision.

5.4.2.1 PACKING PIXELS. As an image is rendered and pixels are generated, there is a precision conversion step in which the computed color value of the pixel is truncated to the depth of the frame buffer. The **ppack** instruction performs this precision conversion and combines the resulting value into a packed structure alongside adjacent pixels. The **ppack** instruction performs this operation in raster order, allowing the display image to be built up as it is computed. Data can then be written to the frame buffer after 64 bits of image are assembled, thus reducing bus traffic to the frame buffer.

Figure 5-9 illustrates the operation of the **ppack.32.h r2,r2,r1** instruction. Grayscale or pseudocolor pixels in 8.8 fixed-point format are shown being packed before being written to an 8-bit frame buffer. The packed pixels are being accumulated in **r2:r3**, with P_0–P_3 having been generated in a previous operation. P_4–P_7 have been rendered in 8.8 format, with **ppack** truncating their fractional parts and concatenating the integer portion to the previously generated pixels in raster order. After the **ppack** operation is complete, the eight pixels in **r2:r3** are ready to be written to the frame buffer.
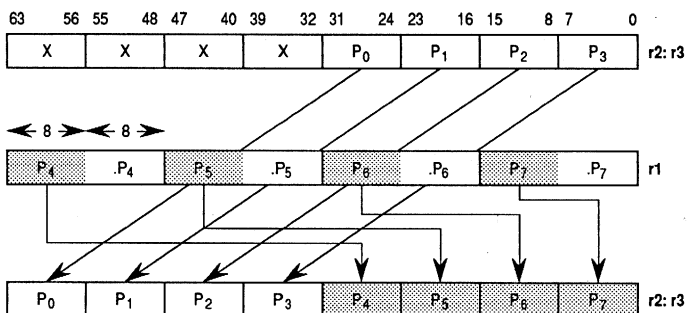
Figure 5-9. ppack.32.h r2,r2,r1

There are six possible variations of the **ppack** instruction. Figures 5-9–5-14 show the six possible combinations of the **ppack** instruction as well as some possible applications for each.

The **ppack.8** instruction can be used to build 16-bit 4/4/4/4 αRGB pixels from 4.28-bit fixed-point intensity values (see Figure 5-10).
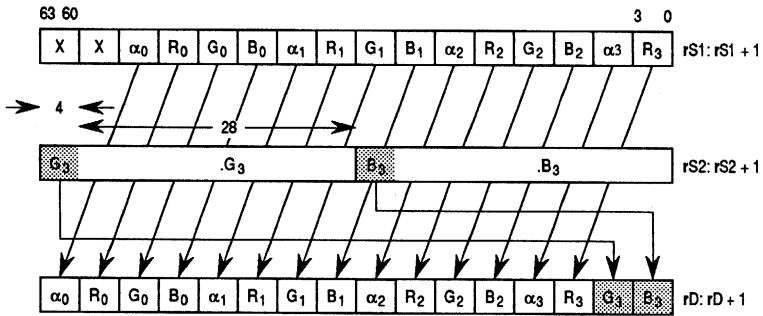
Figure 5-10. ppack.8

The **ppack.16** instruction can be used to build 32-bit 8/8/8/8 αRGB pixels from 8.24-bit fixed-point intensity values (see Figure 5-11).
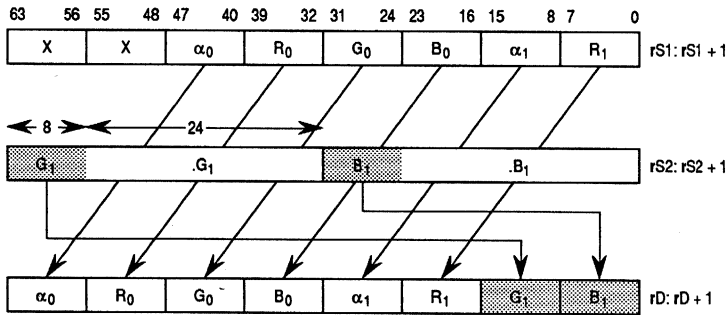
Figure 5-11. ppack.16

The **ppack.16.h** instruction can be used to build 16-bit 4/4/4/4 αRGB pixels from 4.12-bit fixed-point intensity values (see Figure 5-12).
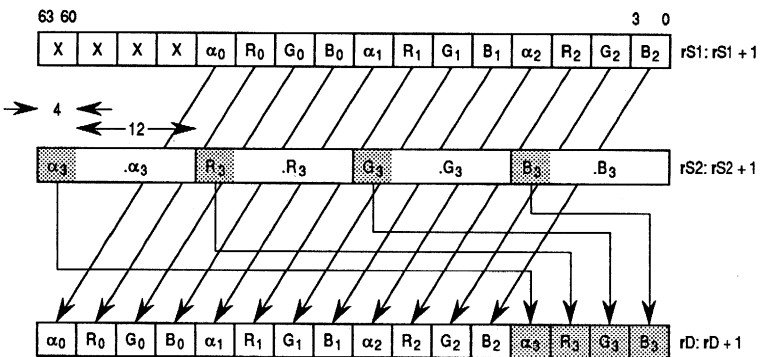
Figure 5-12. ppack.16.h

The **ppack.32** instruction can be used to build 64-bit 16/16/16/16 αRGB pixels from 16.16-bit fixed-point intensity values (see Figure 5-13).
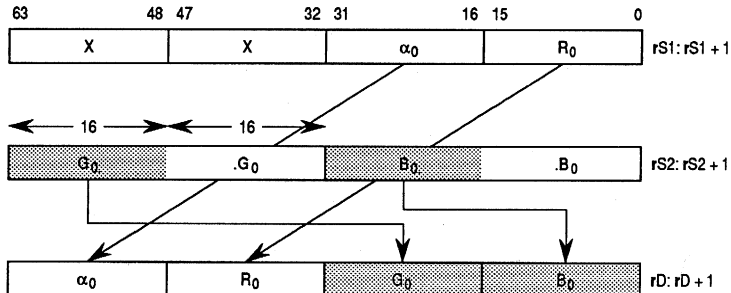
Figure 5-13. ppack.32

The **ppack.32.b** instruction can be used to build 16-bit 4/4/4/4 αRGB pixels from 4.4-bit fixed-point intensity values. This variation can also be used to convert 32-bit 8/8/8/8 αRGB pixels to 4/4/4/4 pixels (see Figure 5-14).
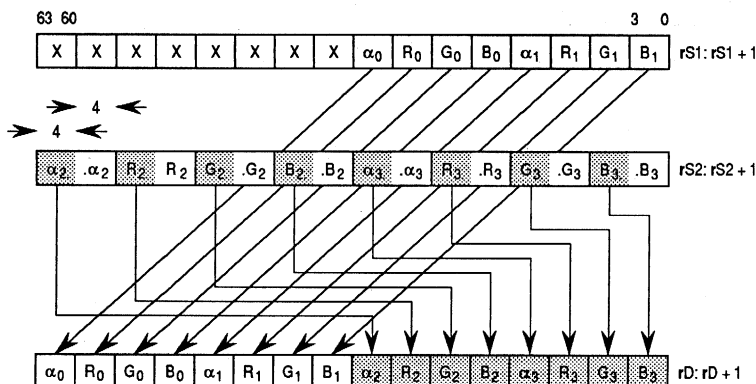
Figure 5-14. ppack.32.b

5.4.2.2 UNPACKING PIXELS. The **punpk** instruction takes a packed bit field and expands it into a bit field twice as large. The original value is right justified in the expanded field, and the remainder of the expanded field is zero-filled. The original value can also be placed in the upper half of the expanded field by following the **punpk** instruction with a **prot** instruction. This procedure is useful during incremental algorithms where a difference value must be added to the color value of a previous pixel. After performing the desired calculations, the integer result can then be obtained using the **ppack** instruction. The **punpk** instruction can also be used to prevent overflow in scaling operations by assuring that the destination field is large enough to hold the largest possible result. The **punpk** instruction can operate on source fields of 4, 8, or 16 bits in length where the field length is specified as the **t** value in the instruction syntax

Figures 5-15, 5-16, and 5-17 illustrate various permutations of the **punpk** instruction. Figure 5-18 demonstrates how the **prot** instruction is used to unpack pixels into the integer rather than the fractional portion of the destination bit field.
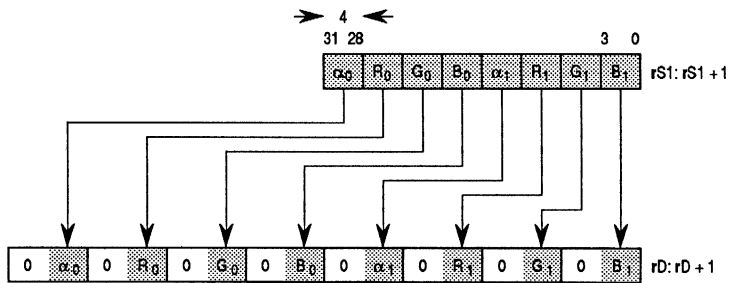
Figure 5-15. punpk.n
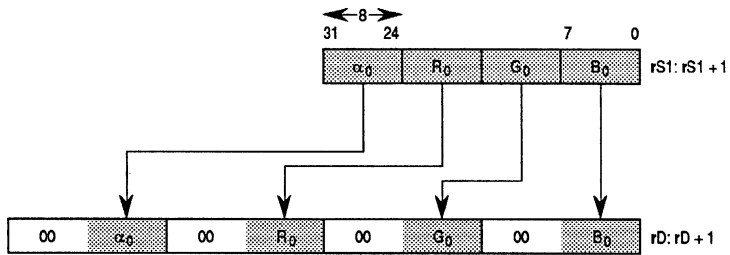
Figure 5-16. punpk.b
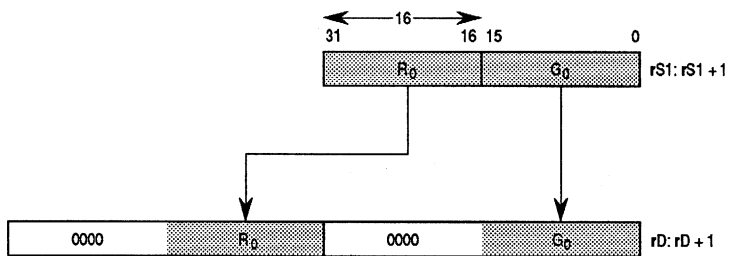
Figure 5-17. punpk.h
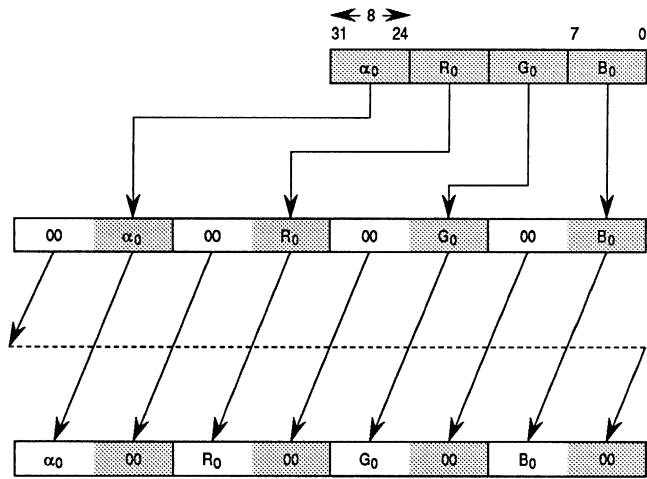
Figure 5-18. punpk.b followed by prot by 8

5.4.3 Intensity Scaling

Operations such as compositing require intensity values to be scaled by an α value represented by a fractional value between 0 and 1. Although the **pmul** instruction operates only on unsigned integers, these can be interpreted as fixed-point fractional values, as described in **5.2.2 Fixed-Point Data Type Definition**. By performing a **punpk** operation on the multiplicand, integer values can be converted to fractions. When multiplied by the alpha value, the proper result is obtained. There is no possibility of overflow during the multiplication because the **punpk** operation automatically provides the necessary resolution to hold the largest possible result. The integer portion of the result can then be extracted using the **ppack** instruction.

The example shown in Figure 5-19 illustrates an 8/8/8/8 packed αRGB pixel being unpacked, scaled by a fractional value, then packed and combined with a previously computed pixel.

```
punpk.b    r2,r1
pmul       r2,r2,r4
ppack.32.h        r6,r6,r2
```
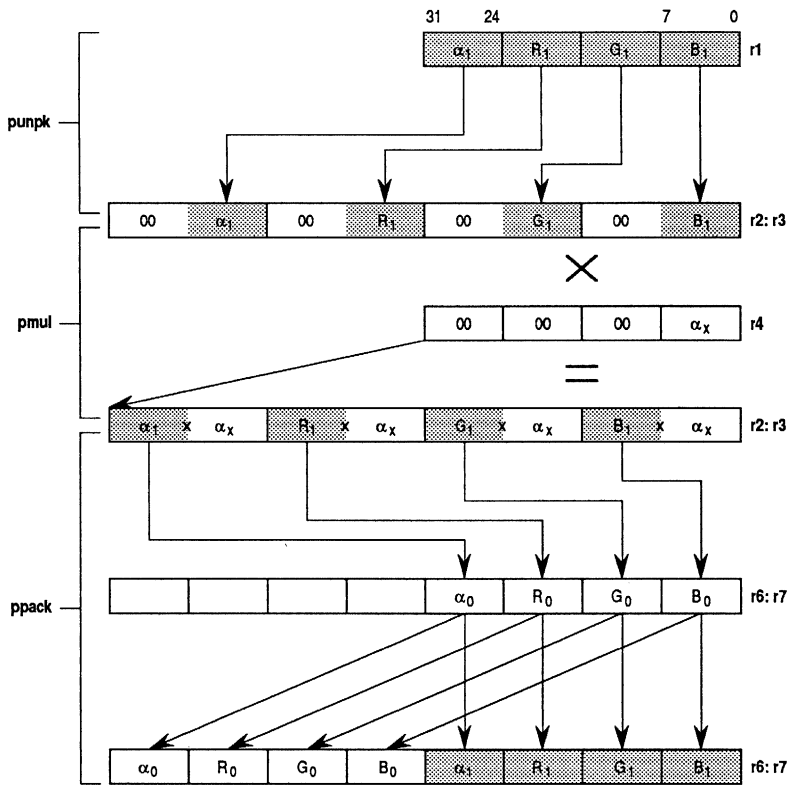

Figure 5-19. Intensity Scaling Example

5.4.4 Coordinate Comparison

When rendering 3D objects, their front-to-back hierarchy must be maintained in the 2D projection to the display. A common method of preserving this order is to store the Z-axis coordinate of each pixel. These coordinates can then be compared to determine which pixel is frontmost. The **pcmp** instruction simultaneously compares two pairs of 32-bit coordinates, which can be represented as either a 32-bit unsigned integer or a positive single-precision floating-point value. Each of the two comparisons may return either less-than ($<$) or greater-than-or-equal-to ($\geq$), resulting in four possible combinations.

The **pcmp** instruction returns an 8-bit result string; four bits indicate which of the four possible conditions was met, and four bits indicate the complement of those bits. Conditional branching on the results of a **pcmp** instruction can be implemented in two ways: with a sequence of conditional branches or with a jump table. Section **5.5.2 Hidden-Surface Removal** discusses some of the trade-offs with both methods. Table 5-3 lists the logical definitions of each bit in the result string for the **pcmp** instruction.

Table 5-3. pcmp Result String

Register Bit(s)	Condition
rD[3:0]:	0
rD[4]:	(rS1:rS1+1[63:32] ≥ rS2:rS2+1[63:32]) and (rS1:rS1+1[31:0] ≥ rS2:rS2+1[31:0])
rD[5]:	(rS1:rS1+1[63:32] < rS2:rS2+1[63:32]) and (rS1:rS1+1[31:0] < rS2:rS2+1[31:0])
rD[6]:	(rS1:rS1+1[63:32] ≥ rS2:rS2+1[63:32]) and (rS1:rS1+1[31:0] < rS2:rS2+1[31:0])
rD[7]:	(rS1:rS1+1[63:32] < rS2:rS2+1[63:32]) and (rS1:rS1+1[31:0] ≥ rS2:rS2+1[31:0])
rD[8]:	!rD[4]
rD[9]:	!rD[5]
rD[10]:	!rD[6]
rD[11]:	!rD[7]
rD[31:12]:	0

5.5 ACCELERATED FUNCTIONS

The graphics extensions of the MC88110 are targeted at improving performance for several of the most common 3D rendering and display maintenance operations. These operations include Gouraud and incremental Phong shading, Z-buffering, compositing, and pixel block transferring.

Several of these operations are discussed in the following paragraphs, and examples are given of how the graphics instructions may be used to implement some of these algorithms.

5.5.1 Gouraud Shading

When polygons are rendered, a color is computed for each vertex of the polygon. The interior of the polygon could be filled with a solid color, usually computed as the average color of the polygon vertex values. This technique is called flat shading, and, although it gives a rudimentary illusion of a solid object and is relatively simple to compute, it produces visible facets that detract from the desired realistic appearance.

Gouraud shading is a more visually accurate method of modeling solid objects. Gouraud shading first interpolates the colors along the edges of the polygon between the vertices, then interpolates across the face of the polygon between edges. This bilinear interpolation removes much of the objectionable faceting and discontinuities found in flat shaded images, creating a more realistic image.

In the Gouraud algorithm, once the edges of the polygon have been interpolated, the interior is filled by interpolating between edges along scan lines. Figure 5-20 shows both the pseudocode and graphical representation for this operation. This example is for an 8/8/8/8 αRGB frame-buffer format, with intermediate color values maintained in an 8.24 fixed-point format.

```
for each scanline do {
        compute ΔαRGB = (P_rt(αRGB) - P_lt(αRGB)) + (y_rt - y_lt)
        from left to right endpoints do {
                P_n = padd(P_n-1(αRGB),ΔαRGB)
                ppack(P_n)
                if (two pixels have been packed together)
                        write double word out to frame buffer
        }
}
```
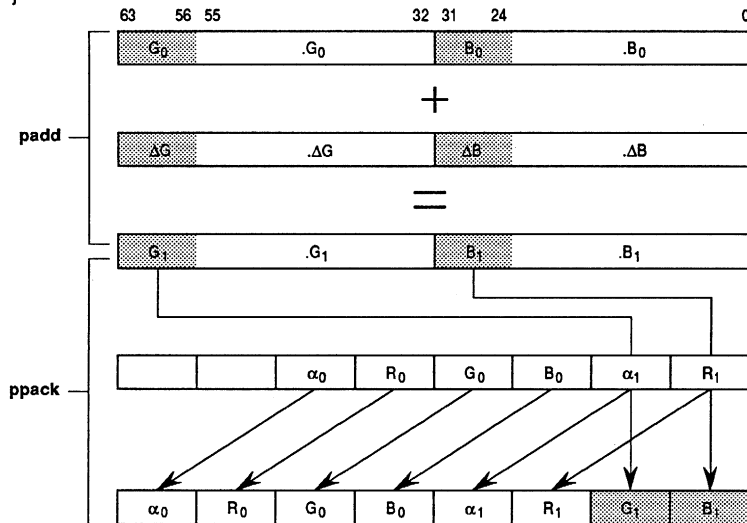

Figure 5-20. Interpolating and Building Pixels

The following code is an example implementation of the inner loop of the Gouraud shading algorithm shown in the pseudocode of Figure 5-19, unrolled to a depth of two. This loop computes two 32-bit αRGB pixels in each iteration. Each loop executes 12 instructions in six clocks generating a new pixel every three clocks (16.7 million pixels per second at 50 MHz assuming cache hits).

```
clock:      0    1    2    3    4    5
g_loop     |-|                          padd        AR,AR,DAR
001C       |-|  |-|                     st.d        P0P1,R0,PPTR
0020            |-|                      ppack.16    P0P1,P0P1,AR
0024            |-|                      add         PPTR,PPTR,8
0028                 |-|                 padd        GB,GB,DGB
002C                 |-|                 sub         N,N,2
0030                      |-|            ppack.16    P0P1,P0P1,GB
0034                      |-|            padd        AR,AR,DAR
0038                           |-|       ppack.16    P0P1,P0P1,AR
003C                           |-|       padd        GB,GB,DGB
0040                                |-|  ppack.16    P0P1,P0P1,GB
0044                                |-|  bcnd        ne0,N,g_loop
```

5.5.2 Hidden-Surface Removal

One of the simplest algorithms for solving the hidden-surface problem is the Z-buffer. For each pixel displayed, a depth value (Z-axis distance from the eye) is maintained along with the color value. When a new polygon is rendered, the Z-value for each pixel is computed and compared to the existing value in the Z-buffer. If the Z-value indicates that the new pixel is closer to the viewer than the pixel currently in the Z-buffer, the existing Z-value and color value is replaced. Figure 5-21 describes a Z-buffer compare algorithm. Note that this algorithm is often combined with the Gouraud shading algorithm, but it is described separately here for clarity.

```
for each polygon P in the polygon list do {
    for each pixel in the polygon P(x,y) do {
        compute Z-depth of P(x,y)
        if Z-depth < Z-buffer(x,y) then {
            compute color(x,y) = color of P(x,y)
            Z-buffer(x,y) = Z-depth
        }
    }
}
```

Figure 5-21. Example Z-Buffer Algorithm

Using the **pcmp** instruction, two 32-bit Z-values can be compared in parallel. The comparison is performed using unsigned arithmetic, allowing Z-values to be represented either as unsigned integers or as positive single-precision floating-point numbers. See **5.2.2 Fixed-Point Data Type Definition** for a description of possible Z-buffer coordinate data types. Each pair may return either < or ≥, resulting in four possible compare combinations.

The following test method explicitly tests for the four possible combinations and branches to the appropriate routine. Due to the continuity of objects being rendered, adjacent pixels are generally either both visible or both obscured. Therefore, in the majority of cases, the sequential test takes either the first or second branch, resulting in an average of 1.5 clocks of execution time.

```
seq_test
pcmp    CCR,newZ,Zbufr      ;compare new point to Z-buffer value
 bb1    4,CCR,repl_both     ;both pixels <, copy both to frame buffer
 bb1    5,CCR,repl_none     ;both pixels ≥, move to next pair
 bb1    6,CCR,repl_first    ;first pixel <, copy to frame buffer
 bb1    7,CCR,repl_secnd    ;second pixel <, copy to frame buffer

repl_both
xxx     rD,rS1,rS2
 .
 .
 .
```

A jump table method uses the result string to directly index into the four routines. The **mak** instruction masks off the complement bits and shifts the result string as required to allow enough room for the largest target routine. Using a jump table as shown in the following code segment takes a longer but more consistent amount of time (4 to 5 clocks)

to execute (depending upon instruction alignment and the state of the target instruction cache), as compared to the 1–6 clocks of the previous sequential test method.

```
jmp_table
  bsr.n    @next                    ;put PC of next instruction in r1
  pcmp     CCR,newZ,Zbufr           ;compare new point to Z-buffer value
@next
  subu     r1,r1,(1<<SCALE)-16      ;adjust PC to point to first routine
  mak      CCR,CCR,8<SCALE>         ;SCALE = LOG2(size of largest routine)
  addu     r1,r1,CCR                ;add offset to target routine
  jmp      r1

repl_none
xxx        rD,rS1,rS2
  •
  •
  •
```

5.5.3 Pixel Block Transfer (PixBlt)

In a display system, it is convenient to move rectangular blocks of pixels to and from the frame buffer—an operation called a pixel block transfer (PixBlt). The PixBlt operation is not used for rendering primitives directly, but is used rather to make portions of off-screen bit maps visible and to save and restore pieces of the screen for window management, menu handling, scrolling, and other display maintenance functions.

A PixBlt function may include some or all of the following operations: reading the source and destination pixel maps, masking, rotating, and logically combining them, and writing the result to the frame buffer. The GPU instruction **prot**, the user-mode cache control features, and the base 88000 architecture bit field and logical instructions provide a flexible and efficient mechanism for performing PixBlt operations on pixels of any color depth.

5.5.4 Compositing

Images that have been rendered independently often need to be combined into one image. For simple opaque rectangular images that can be overlaid on pixel boundaries (such as windowing operations), a PixBlt operation is usually sufficient. For more complex objects of arbitrary shape, which may also be partially transparent, another method of combination should be used to avoid objectionable aliasing artifacts and to create a realistic image.

For objects of arbitrary shape, compositing using an α channel to represent the transparency of a pixel is an effective method of combining images. In addition to having a color value, each pixel is also assigned an α value, which is stored with that pixel. A pixel near the edge of a polygon will have an α value less than one if the polygon does not cover the entire pixel. Images can be overlaid, and a composite color can be computed by summing the relative contribution of the foreground and background pixels. Figure 5-22 shows some α values that might be assigned to a simple solid polygon.

**Figure 5-22. Example Polygon
α Value Assignment**

The GPU efficiently supports this type of compositing operation with the **punpk, pmul,** and **ppack** instructions. First, foreground pixels are loaded and unpacked to a format appropriate for the **pmul** operation. Then they are multiplied by α. Background pixels are loaded, unpacked, and multiplied by $(1 - \alpha)$. The foreground and background are then summed, packed, and stored to the frame buffer. Saturation arithmetic is not used during the summing operation because the result cannot overflow. Figure 5-23 illustrates both the pseudocode and graphical representation for the compositing operation.

A similar compositing operation can be performed on objects that are not totally opaque. By assigning solid pixels α values of less than 1.0, an object will appear partially transparent when composited over a background image.

```
for each foreground pixel Pf(x,y) {
    punpk (Pf(x,y))
    Pf' = pmul (α,Pf(x,y))
    fetch background pixel Pb(x,y)
    punpk (Pb(x,y))
    Pb' = pmul ((1-a),Pb(x,y))
    Pc(x,y) = padd (Pf',Pb')
    ppack (Pc(x,y))
    store composite result Pc(x,y) in frame buffer
}
```
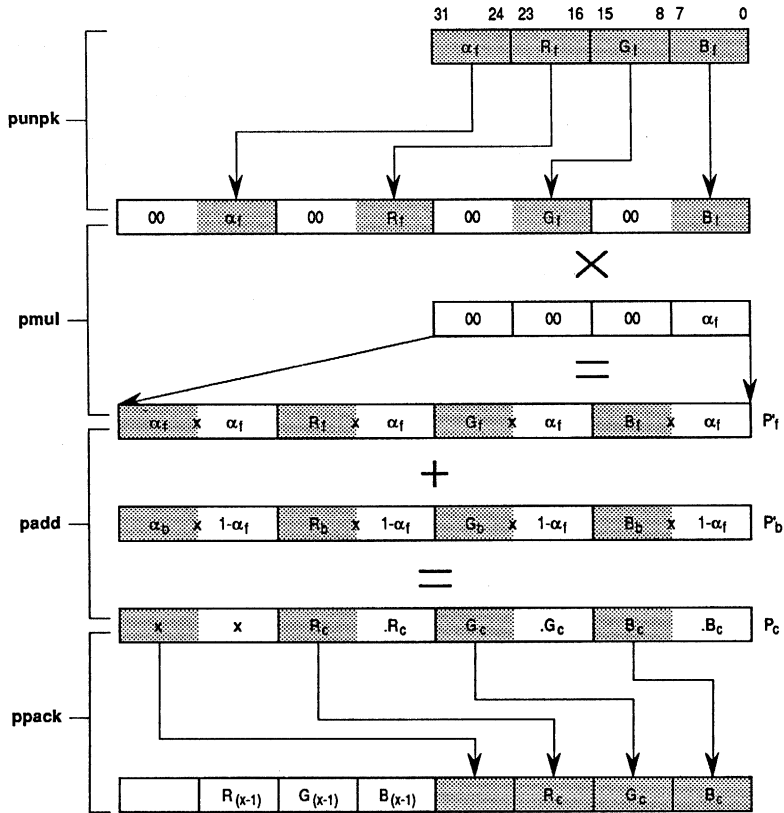

Figure 5-23. Compositing Operation Example

5

SECTION 6
INSTRUCTION AND DATA CACHES

The data and instruction caches of the MC88110 each provide 8K-bytes of high-speed storage. These caches are two-way set associative and physically addressed. Cache management facilities provide both the instruction and data caches with a cache freezing capability. In addition, the data cache has three software-selectable memory update policies and hardware to support cache coherency.

This section describes the cache organization, cache coherency support, memory update policies, cache accesses, data cache decoupling, and cache control and maintenance for the MC88110. This section also describes the target instruction cache (TIC), which is used by the instruction unit for branch acceleration. Refer to **Section 11 System Hardware Design** for more information on hardware cache coherency support and secondary cache support.

Although this section describes the action of the data cache during memory accesses, it does not include information on the data unit and the run-time reordering of loads and stores. This information resides in **Section 9 Instruction Timing and Code Scheduling Considerations**. Also, note that the timing for the external signals in this section is only accurate to within a half-clock cycle and is included for reference only.

6.1 CACHE ORGANIZATION

The instruction and data caches are each configured as 128 sets with two lines per set: line zero and line one (see Figure 6-1). Each line contains eight words of data, an address tag, and status bits. Note that in Figure 6-1, line zero of each set is shaded. The entire shaded area (all of the line zeros as a unit) is bank zero. Likewise, the entire nonshaded area (all of the line ones as a unit) is bank one.
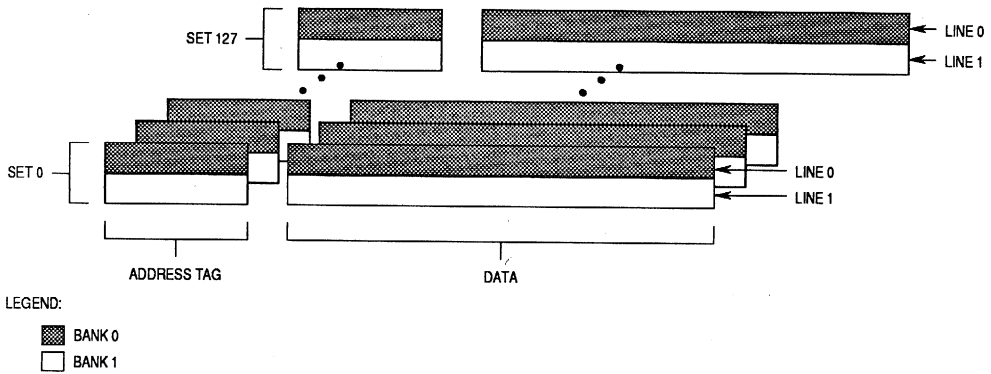
Figure 6-1. MC88110 Cache Terminology

The following paragraphs describe the organization of the data cache, instruction cache, and TIC.

6.1.1 Data Cache

Each line of the data cache contains eight 32-bit words, an address tag, and three status bits. The three status bits indicate whether the cache line is valid or invalid, modified or unmodified, and shared or exclusive. A block diagram of the data cache organization is shown in the Figure 6-2.

Each data cache line contains eight contiguous words from memory which are loaded from an 8-word boundary (i.e., bits A4–A0 of the logical addresses are zero); thus, a cache line will never cross a page boundary. All bus operations that load data into or out of the cache from memory are performed on a line basis (i.e., an entire line is filled). New lines are allocated into empty cache lines if possible. A pseudorandom replacement algorithm is used to select a cache line when no invalid lines are available.
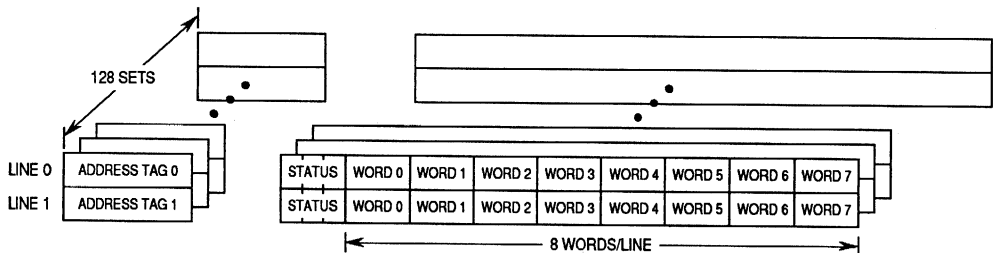
Figure 6-2. Data Cache Organization

Bus transactions to load data into the data cache always begin with the address of the evenly aligned double word containing the missed data. For example, if a half-word load from the address $700016 is requested but misses the cache, then the double word at address $700010 is loaded into the cache first, followed by the double words at

$700018, $700000, and $700008 (see Figure 6-3). The missed data is forwarded to the data unit as soon as it is received from the bus so that it can be used as soon as possible.
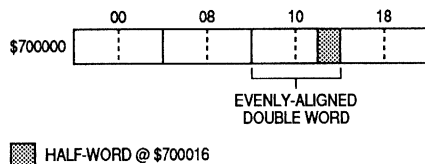
Figure 6-3. Double-Word Alignment

The data cache supports write-through, write-back, and cache inhibited memory update policies which are selectable on a page-by-page or block-by-block basis. These memory update policies are described in **6.4 Memory Update Policies**.

The data cache uses physical address tags, so the data cache does not need to be flushed on a context switch. Cache coherency is automatically maintained by hardware bus snooping. To prevent snooping traffic on the bus from interfering with processor operation and degrading performance, the state bits associated with each line in the cache are dual ported and a duplicate set of cache tags is maintained.

6.1.2 Instruction Cache

Each line of the instruction cache contains eight 32-bit words, an address tag, and a valid bit. A block diagram of the instruction cache organization is shown in Figure 6-4.

Each instruction cache line contains eight contiguous words from memory which are loaded from a 8-word boundary (i.e., bits A4–A0 of the logical addresses are zero); thus, a cache line will never cross a page boundary. All bus operations that load instructions into the cache from memory are performed on a line basis (i.e., an entire line is filled). New lines are allocated into empty cache lines if possible. A pseudorandom replacement algorithm is used to select a cache line when no empty lines are available.

Bus transactions to load instructions into the cache always begin with the address of the evenly aligned double word containing the missed word. The missed word(s) is forwarded to the instruction unit as it is received from the bus so that instruction issue and execution can be resumed as quickly as possible.

The instruction cache uses physical address tags, so the instruction cache does not need to be flushed on a context switch. Instruction cache coherency must be maintained by software and is supported by a fast hardware invalidation capability. For a detailed description of the invalidate feature, refer to **6.9.3 The Invalidate Command**.
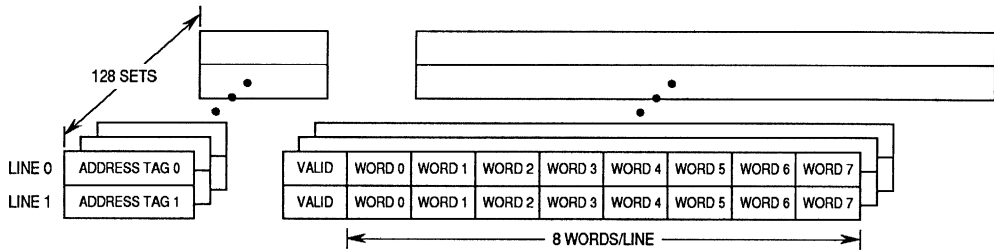
Figure 6-4. Instruction Cache Organization

6.1.3 Target Instruction Cache (TIC)

The MC88110 has a TIC, which is a 32-entry, fully associative, logically addressed cache. Each entry in the TIC contains the first two instructions of a branch target instruction stream, a 31-bit logical address tag, and a valid bit (see Figure 6-5). The 31-bit logical address tag holds a supervisor/user bit and the upper 30 bits of the address of the branch instruction. Because the TIC is logically addressed, it must be invalidated on a context switch. The operation of the TIC is discussed in detail in **Section 9 Instruction Timing and Code Scheduling Considerations**.
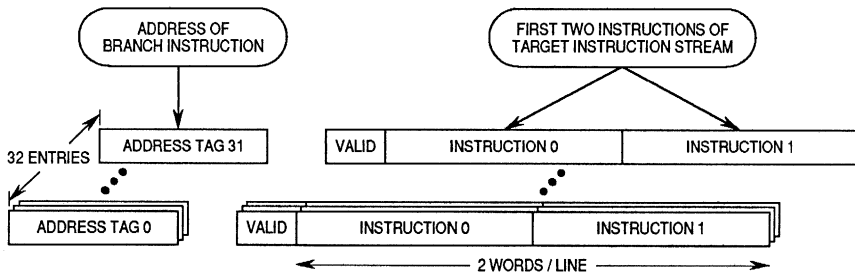
Figure 6-5. Target Instruction Cache (TIC)

6.2 CACHE COHERENCY

The instruction cache is physically addressed and is therefore coherent across multiple process contexts. However, no hardware support is provided to maintain coherency between multiple instruction caches or between the instruction cache and main memory. Software must force coherency in any situation which could cause the instruction cache to have invalid data (e.g., virtual memory page replacement).

Hardware support is provided to maintain coherency between the data cache and memory. To maintain this coherency, the MC88110 incorporates bus snooping. For a complete description of data cache coherency and bus snooping refer to **Section 11 System Hardware Design**.

One aspect of data cache coherency that must be considered by the software programmer is whether to mark a page global or local. In order for other processors to snoop a transaction, the page containing the data must be marked global. Therefore, if a page is accessed by more than one processor, and changes made by one processor are relevant to the others, the page should be marked as global. If a page is being accessed by more than one processor, but the changes are only relevant to one processor, the page should be marked as local.

Note that marking a page as global can cause a decrease in performance. In a no-wait state system with snooping enabled, one clock cycle must be added to the beginning of each bus transaction to give the snooping processor time to assert a retry. When a processor receives a retry, it is delayed while the snooping processor arbitrates for the bus and copies the data back to memory. This process of flushing a dirty (modified) cache line to update memory is called a copyback. Then, the initiating processor must retry the original transaction from the logical address cache lookup and re-arbitrate for the bus. The snooping processor's data cache is unavailable for five clocks while it copies back the modified data to memory. Because of this potential performance degradation, care should be taken to group local data together on a page without global data so that transactions accessing local data are not snooped.

6.3 ADDRESS TRANSLATION OVERVIEW

6

When an instruction or data access is initiated, the instruction unit or data unit provides the appropriate cache and memory management unit (MMU) with the logical address of the desired information. The instruction or data MMU translates the logical address to the physical address and provides the cache with information about the type of cache access to be performed. For detailed information on the MMUs, see **Section 8 Memory Management Units**.

The instruction and data MMUs each contain two address translation caches (ATCs) that provide address translation with no time penalty: a page address translation cache (PATC) and a block address translation cache (BATC). The PATCs are 32-entry, fully associative caches that contain address translations for 4K-byte memory pages and are automatically maintained by MC88110 hardware. The BATCs are 8-entry, fully associative caches containing address translations for block sizes ranging from 512K-byte to 64M-byte. The BATC descriptors are managed by system software. Each BATC and PATC descriptor contains a logical address field, a physical address field, and attribute bits that define the characteristics of the corresponding block or page (e.g., supervisor/user memory space, global/local space, etc.)

When address translation is enabled, the MMU compares the logical address of the access with the upper bits of the logical address fields of each descriptor in the BATC and PATC. If there is a match with an descriptor in either ATC, then the contents of the physical address field of that descriptor replace the most significant bits of the logical address, and the resulting bit string forms the physical address for the access (see Figure 6-6). The physical address and the attribute bits are then used by the cache. If there is no logical address match in either ATC, the MMU retrieves a new descriptor by performing a table search operation (if table searching is enabled).
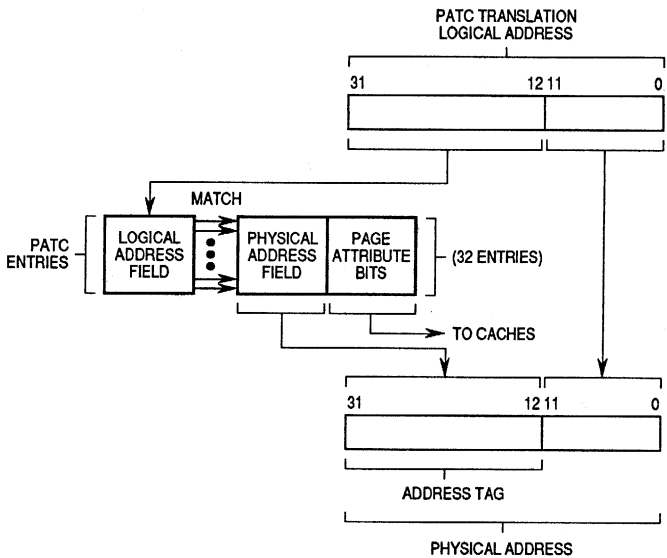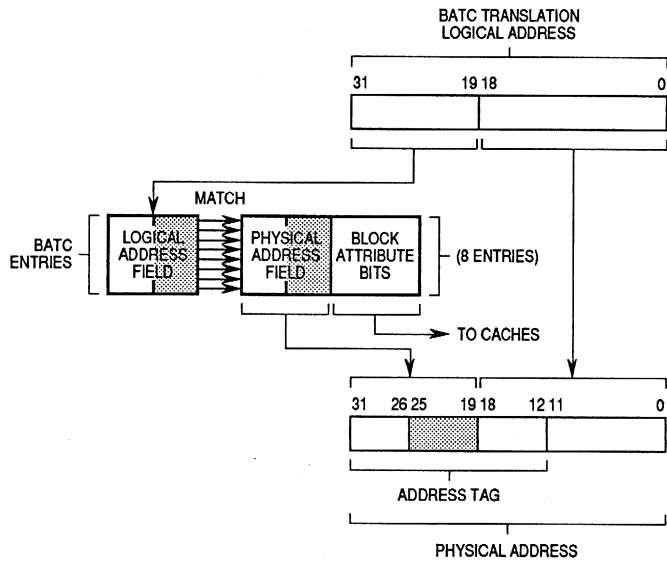
LEGEND:

▨ BITS WHICH MAY BE TRANSLATED BY BATC OR REMAIN UNCHANGED
FROM THE LOGICAL ADDRESS DEPENDING ON THE BLOCK SIZE

Figure 6-6. Physical Address Generation Using ATCs (ATC Hit)

When address translation is disabled, the logical address is the same as the physical address, and the attribute bits are contained in the instruction supervisor area pointer (ISAP) register, instruction user area pointer (IUAP) register, data supervisor area pointer (DSAP) register, or data user area pointer (DUAP) register for each corresponding memory area.

6.3.1 BATC Descriptors

Each BATC contains eight 34-bit descriptors that provide address translation, control, and protection information for logical-to-physical block address translation. The format of the BATC descriptors for both the instruction BATC and the data BATC is shown in Figure 6-7.

33	32	31	26	25	19	18	12	11	6	5	4	3	2	1	0
U1	U0		LBA				PBA			S	WT	G	CI	WP	V

▓ IGNORED DEPENDING ON BATC SIZE. SIZE MASK WRITTEN VIA THE INSTRUCTION MMU/CACHE CONTROL REGISTER OR DATA MMU/CACHE CONTROL REGISTER
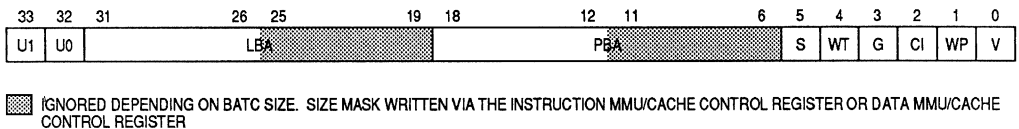
Figure 6-7. BATC Descriptor Format

U0,U1—User Page Attributes 0,1
These bits are not used by the MMU but are user definable from software. They are driven on external signals during bus transactions mapped by this descriptor. They are loaded into the BATC descriptor via the instruction or data index register (xIR) at the time of a BATC write.

LBA—Logical Block Address
This field contains the logical address (tag) that is matched against the logical address of a memory access. If an address match is found (including the supervisor mode bit S) and the valid (V) bit is set, then the logical address is mapped to the physical address as specified in the PBA field.

PBA—Physical Block Address
If the logical address hits with this descriptor, the 6–13 (depending on block size) most significant bits of the logical address are replaced by the bits in this field to form the translated physical address.

S/U—Supervisor/User Mode
This bit is compared to the level of privilege for the memory access being translated. If this bit matches the level of privilege, and the LBA field matches the logical address, and this descriptor is valid (V=1), then this descriptor is used to map the logical address to the physical address specified in the PBA field.

6

WT—Write-Through

If this bit is set, then cache memory updates to memory mapped by this descriptor are performed using a write-through policy. If this bit is clear then cache memory updates are performed using a write-back policy. Note: if the CI bit is set, WT has no effect.

G—Global

If this bit is set, then the memory space mapped by this descriptor is designated global memory. The state of this bit is reflected on an external signal during the bus transaction for the access and can be used by other devices on the bus to enable or disable snooping for this address.

CI—Cache Inhibit

If this bit is set, then data in the block mapped by this descriptor is not cached. All accesses to this block will go through to memory and no data is read from, written to, or allocated in, the cache. If this bit is clear then data mapped by this descriptor is cached normally.

WP—Write Protect

If this bit is set, then memory mapped by this descriptor is write protected. Any write access to this page causes a data access exception. If this bit is clear then memory mapped by this descriptor can be written.

V—Valid

If this bit is set, then this descriptor is currently valid and if the logical address matches the LBA, this descriptor is used to translate the address.

6.3.2 PATC Descriptors

The PATC contains thirty-two 64-bit descriptors that provide address translation, control, and protection information for logical-to-physical page translation. The format of the PATC descriptor for both the instruction PATC and the data PATC is shown in Figure 6-8.
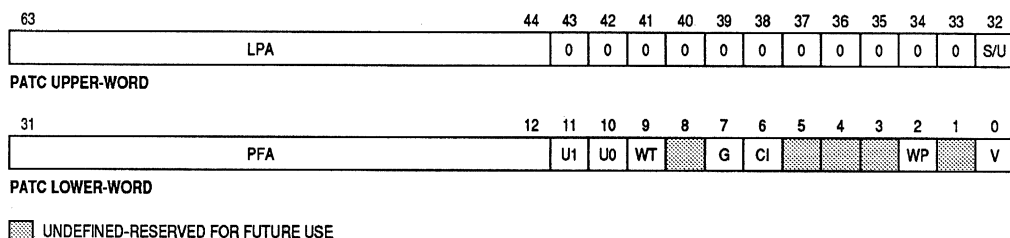
Figure 6-8. PATC Descriptor Format

LPA—Logical Page Address

This field contains the logical address that is to be mapped to a physical address by this ATC descriptor. The LPA is used as a tag which is matched against the logical address of subsequent memory references. If an address match is found (hit) then this descriptor is used to translate the logical address to a physical address as specified in the PFA field.

S/U—Supervisor/User Bit

This bit is compared to the level of privilege for the memory access being translated. If this bit matches the level of privilege, and the LPA matches the logical address, and this descriptor is valid (V=1), then this descriptor is used to map the logical address to the physical address specified in the PFA field.

PFA—Page Frame Address

This field contains the upper 20 bits of the physical address to which the logical address is being mapped. If the upper 20 bits of the logical address of the access match the LPA and access privileges are not violated, then the 20 most significant bits of the logical address are replaced by the PFA to create the physical address.

U0,U1—User Page Attribute 0,1

These bits are not used by the MMU but are user definable via the page descriptors or by a PATC control register write (using xIR). They are driven on external signals during the bus transaction.

WT—Write-Through

If this bit is set, then cache memory updates are performed using a write-through policy. If this bit is clear then cache memory updates are performed using a write-back policy. Note: if the CI bit is set, this bit has no effect.

G—Global

If this bit is set, then the memory space mapped by this descriptor is global memory. The state of this bit is reflected on an external signal during the bus transaction for the access and can be used by other devices on the bus to enable or disable snooping on this address.

CI—Cache Inhibit

If this bit is set, then data in the page mapped by this descriptor is not cached. All accesses to this page will go through to memory and no data is read from, written to, or allocated in, the cache. If this bit is clear then data mapped by this descriptor is cached normally.

WP—Write Protect

If this bit is set, then memory mapped by this descriptor is write protected. Any write access to this page causes a data access exception. If this bit is clear then memory mapped by this descriptor can be written.

6

V—Valid

>If this bit is set, then this descriptor is currently valid and if the logical address matches the LPA, this descriptor it is used to translate the address.

6.4 MEMORY UPDATE POLICY

The MC88110 provides hardware support for three memory update modes: write-back, write-through, and cache inhibit. Each page or block of memory is specified to be in one of these modes, with write-back mode being the default upon reset. The MC88110 also has a store-through option which allows individual accesses to be performed in write-through mode, even if the page being written to is in write-back mode.

In write-back mode, external memory is not updated each time a corresponding cache line is modified. In write-through mode, writes update external memory every time the cache line is modified. For cache inhibited accesses, reads and writes access main memory, but data is never stored in the data cache. All three of these modes of operation have specific advantages; therefore, the choice of which mode to use depends on the system environment and the application.

When address translation is enabled, the cache memory update policy is determined on a page-by-page (or block-by-block) basis by the WT and CI bits in the address translation cache descriptor that was used to translate the corresponding logical address. When address translation is disabled, the WT and CI bits in the ISAP, IUAP, DSAP, or DUAP register control the memory update policy for each corresponding memory area.

Two logical pages that map to the same physical page can have different memory update policies. This can be useful as a user mode cache control feature to flush and invalidate a line in the cache. To do this, one of the two logical pages should be marked cache inhibited and the other should be marked either write-back or write-through. Any hit on the cache-inhibited page will flush and invalidate the line. Then, for normal accesses, an address on the write-back or write-through page can be specified. To flush and invalidate a line, an address on the cache-inhibited page is specified.

6.4.1 Write-Back Mode

When storing to memory in write-back mode, store operations for cacheable data do not necessarily cause an external bus cycle to update memory. Instead, memory updates only occur when another bus master attempts to access a specific address for which the corresponding cache descriptor has been modified (i.e., the cache descriptor is "dirty") or when the cache performs a replacement copyback due to a read or write miss. Memory can also be updated if the location is declared to be cache inhibited or during the write portion of an **xmem** transfer. For this reason, write-back mode may be preferred when external bus bandwidth is a potential bottleneck—e.g., in a multiprocessor environment without a secondary cache. Write-back mode is also well suited for high-use data that is closely coupled to a processor, such as stacks and local variables. Reads to memory in

write-back mode that hit the on-chip data cache can complete in two clock cycles. Writes that hit in the on-chip caches can complete in three clock cycles. A read or write can be initiated on every clock.

In general, addresses at which data is to be used by only one processor and with no other bus master should be mapped as local (G = 0) and write-back (WT = 0) for maximum performance. The G bit is located in the same place that the corresponding WT bit is located in (i.e., in the same area pointer or BATC or PATC descriptor).

The MC88110 implements snooping hardware to prevent other devices from accessing invalid data. If more than one processor uses data stored in a page or block which is in write-back mode, snooping must be enabled to allow write-back operations and cache invalidations of modified data. When snooping is enabled, the page should be marked as global (G = 1) and write-back (WT = 0). When bus snooping is enabled, the MC88110 monitors the transactions of the other devices. For example, if another device accesses a memory location that is cached and modified in an MC88110 and the global (G) bit corresponding to that page is set, the MC88110 preempts the bus transaction, and updates memory with the cached data. See **Section 11 System Hardware Design** for complete information on bus snooping.

6.4.2 Write-Through Mode

Write-through mode is used when external memory and internal cache images must agree (e.g., video memory) or when there is shared (global) data that may be used frequently. In write-through mode, store operations which hit the cache update external memory as well as the data cache and do not change the state of the line. Store operations in write-through mode which miss the cache update external memory only. In write-through mode, global transactions cause snoop logic to invalidate other processors' cached images of updated memory. This mode of operation is normally selected for systems employing an external secondary cache.

In write-back mode, the cache may contain data which is modified with respect to memory. Therefore, a page marked as write-back may contain data in an exclusive state. However, in write-through mode, reads and writes that hit the cache do not change the status of the line. If a write-back page is changed to write-through mode, the data will not change state even if a write hits the cache and the data is no longer exclusive.

One type of access, the store-through access, is determined on an access-by-access basis. Store-through may be optionally specified for any store instruction using the triadic register addressing mode. A store-through access operates in precisely the same manner as an operation in write-through mode even if write-back mode is specified for the page being accessed; however, if the page is specified as cache inhibited, the store-through option has no effect. For more information on selecting the store-through option, refer to **6.9.1 User-Mode Cache Control Features.**

If the force write-through (FWT) bit is set in the data MMU/data cache control register (DCTL), all stores are forced to write-through the data cache regardless of the page or block status.

6.4.3 Cache Inhibit

A memory location is cache inhibited if the CI bit in the corresponding ATC descriptor is set. If a memory location is declared to be cache inhibited, data from this location will never be stored in the data cache.

Certain transactions are cache inhibited regardless of the memory update policy. The data cache is bypassed whenever the MC88110 executes a hardware table search; the segment and page descriptors are not cached in the data cache. In addition, **xmem** operations are always performed as if cache inhibition is in effect regardless of the memory update mode for the location being accessed.

A transaction that is translated as a cache inhibited access and hits a modified line in the data cache causes the corresponding line to be copied back to memory and invalidated.

6.5 CACHE LOOKUP OPERATION

Each time the processor performs a memory access, the MC88110 initiates a cache lookup operation in which, simultaneously, the cache selects the correct cache set, and the memory management unit (MMU) performs the address translation (see Figure 6-9). To achieve this concurrency, the MC88110 uses the fact that the low-order 12 bits of an address are the same for both the logical and physical address. Thus, the high-order bits of an address can be used by the MMU for the address translation while the low-order bits are being used by the cache for the set selection. Figure 6-10 shows how the fields of the logical address are used by the caches and MMUs.

6
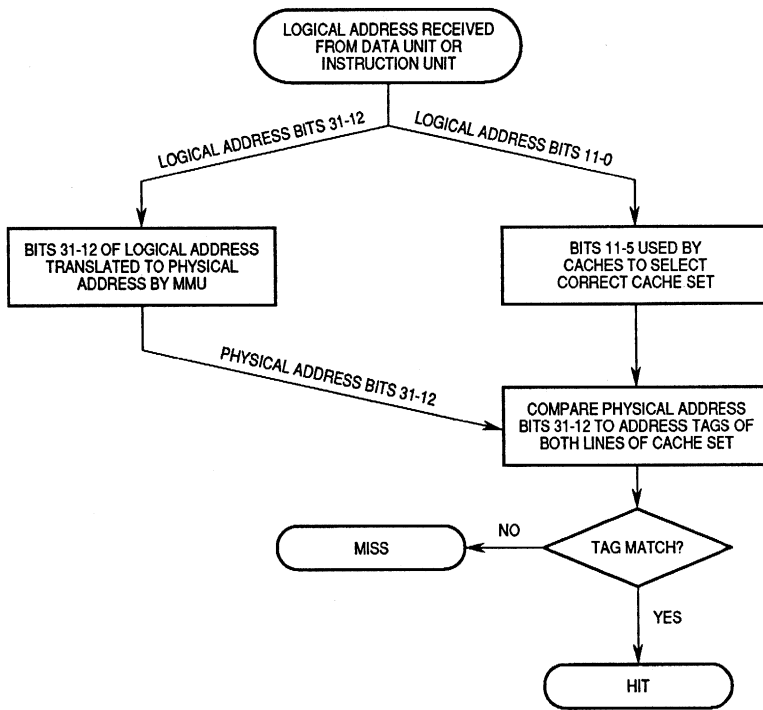
Figure 6-9. Cache Lookup Operation

When the data or instruction cache receives a logical address, the appropriate cache set is selected based on bits 11–5 of the address. While the set is being selected, the 20 most significant bits (bits 31–12) of the logical address are translated to a physical address by the MMU. The address tags from the fetched cache lines are then compared against the translated physical address bits from the MMU. If the address tag from either line matches the translated physical address bits, then a cache hit has occurred. If neither of the address tags from the fetched set matches the translated physical address bits, a cache miss has occurred.

As shown in Figure 6-10(a), for a data cache hit, the appropriate double word in the cache line is accessed according to bits 4–3 of the address and forwarded to the appropriate execution unit. As shown in Figure 6-10(b), for an instruction cache hit, bits 4–2 are used. The extra bit is needed because the instruction cache address does not necessarily fall on an evenly aligned double word boundary.

LOGICAL ADDRESS

```
31                                    12 11        5 4 3 2   0
┌──────────────────────────────────┬──────────┬─────┬─┬─┬─┐
│                                  │          │     │ │ │ │
└──────────────────────────────────┴──────────┴─────┴─┴─┴─┘
```

LOGICAL ADDRESS BITS TO
BE TRANSLATED BY MMU

BITS USED TO SELECT
APPROPRIATE CACHE SET

BITS USED TO SELECT APPROPRIATE
DOUBLE WORD IN CACHE LINE

BITS WHICH SPECIFY THE ALIGNMENT
OF DATA WITHIN THE DOUBLE WORD

(a) Data Cache

LOGICAL ADDRESS

```
31                                    12 11        5 4   2 1 0
┌──────────────────────────────────┬──────────┬─────┬───┬──┐
│                                  │          │     │   │  │
└──────────────────────────────────┴──────────┴─────┴───┴──┘
```

LOGICAL ADDRESS BITS TO
BE TRANSLATED BY MMU

BITS USED TO SELECT
APPROPRIATE CACHE SET

BITS USED TO SELECT APPROPRIATE
DOUBLE WORD IN CACHE LINE

UNUSED

(b) Instruction Cache

Figure 6-10. Logical Address Fields

For a cache miss, a line in the selected set must be chosen to hold the new data which will be fetched from memory. If one of the lines is invalid, then it is chosen to receive the data. If both lines are invalid, then line 0 is chosen. If both lines are valid then a pseudorandom selection algorithm is used to select one of the two lines for replacement. This replacement algorithm employs a one-bit counter which toggles the selection bias from one bank to the other upon the successful completion of a burst read or allocate. The counter is cleared to zero by a reset operation or a cache invalidate operation.

6.6 INSTRUCTION CACHE ACCESSES

The following paragraphs describe the actions taken by the MC88110 due to an instruction cache read (see Figure 6-11). It is assumed in Figure 6-11 that a physical address has already been generated by the instruction MMU. Note that the timing for the external signals in this section is only accurate to within a half-clock cycle and is included for reference only.
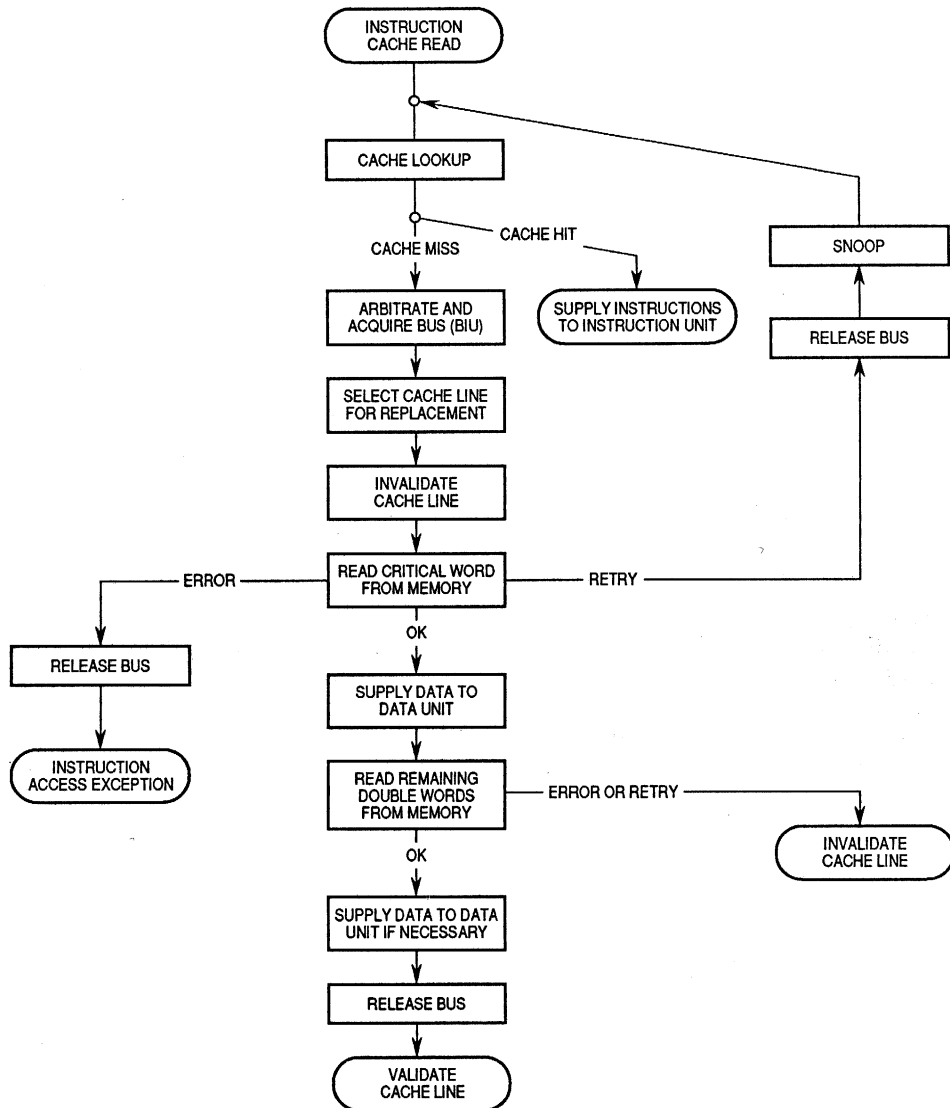
Figure 6-11. Instruction Cache Read Flowchart

6.6.1 Instruction Cache Hit

If an instruction cache read results in a cache hit, the needed double word (indicated by bits 4–2 of the address) is accessed from the cache line. There is no even or odd alignment restriction on double words in the cache line. The instruction(s) from the accessed word(s) are immediately transferred to the instruction unit, allowing two instructions per clock cycle to be delivered to the instruction unit. If an access is to the last word in a cache line, then only a single-word is retrieved. Figure 6-12 shows the timing for instruction cache hits. Note that since instruction 2 is the last instruction in a cache line, it is issued alone in clock cycle 2.
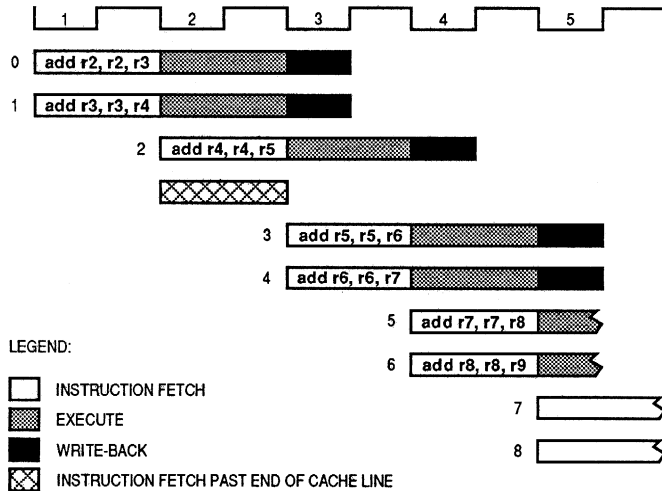
Figure 6-12. Instruction Cache Hit Timing

6.6.2 Instruction Cache Miss

On an instruction cache miss, the physical address of the missed instruction is sent to the bus interface unit (BIU) with a request to retrieve the cache line from memory, and a cache line is selected to receive the data which will be coming from the bus. If there is a simultaneous data cache miss, the BIU gives priority to the instruction cache request unless the data cache must perform a snoop copyback or an **xmem** transaction, or the data cache requests the bus after being retried and forced off the bus.

The instruction cache line fill always begins with the evenly aligned double word containing the missed instruction (i.e., critical word first), followed by the subsequent double word(s) in the line, if any. If the double word containing the missed instruction was not the first double word in the line, the fill wraps around and fills the double word(s) at the beginning of the line. As soon as the missed instruction is forwarded to the instruction unit, instruction execution is allowed to resume and proceeds in parallel with the remaining line fill. If there is no change in program flow, and the instruction unit can issue the following instructions, subsequent instructions are forwarded to the instruction

unit as they are received. This is referred to as streaming. If there is a change in program flow during a line fill, instruction issue is stalled until the line fill is completed. Figure 6-13 shows the timing for an instruction cache miss, assuming an ideal memory system. Note that, in this case, an instruction cache miss causes a three-cycle delay.
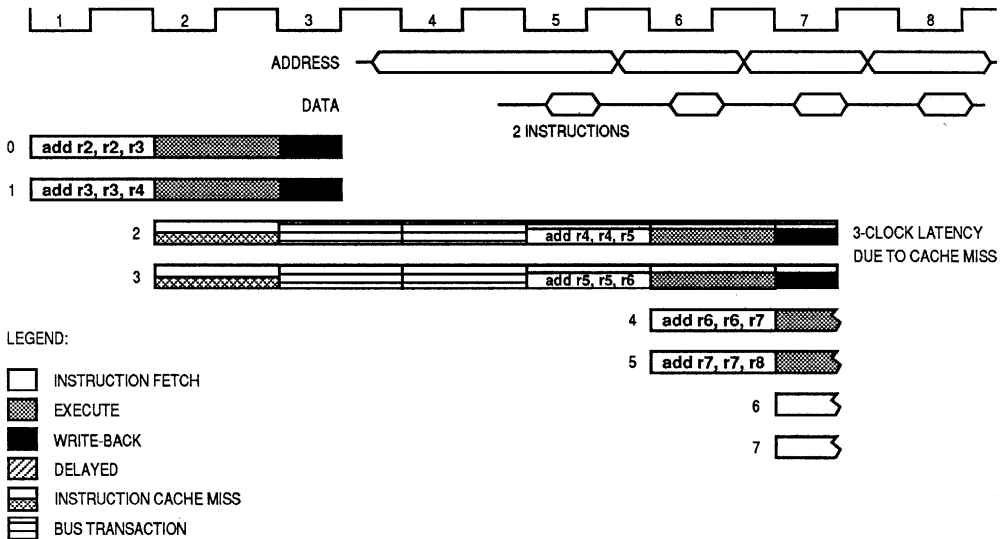
Figure 6-13. Instruction Cache Miss Timing

If a bus error is encountered on a memory access for a missed instruction, then an instruction access exception is taken (see **Section 7 Exceptions**). If a bus error occurs on any other word in the transfer, then the fetched line is marked invalid. If no bus error is encountered during the line fill, the line is marked valid.

6.7 DATA CACHE DECOUPLING

The data cache provides a decoupling feature to improve cache performance. When the decoupling feature is enabled, the data unit can continue making cache accesses while the data cache is waiting to receive data from the bus. These cache accesses are called decoupled cache accesses. If a decoupled cache access hits the cache and does not require an external bus transaction, the access is allowed to complete. If a decoupled cache access requires an external bus transaction, no further decoupled accesses are allowed, and the cache access is restarted when the cache is available. Decoupling is enabled by setting the decoupled cache access enable (DEN) bit in the DCTL. Refer to **Section 11 System Hardware Design** for more information on decoupled cache accesses.

When the data cache determines that a line fill or single-beat read is necessary, there will be at least one clock cycle during which the cache is idle while waiting for data. The

decoupling feature allows the data unit to access the cache during this time. The data cache is decoupled from the first clock cycle of the memory access until the pretransfer acknowledge (PTA) signalis asserted on the external bus (see **Section 11 System Hardware Design**). If a line must be copied back to memory before the line fill, then the cache will be decoupled for one clock cycle between the copyback and the line fill (assuming ideal memory). If there is no copyback, the cache will be decoupled for two clock cycles before the line fill begins.

Not all cache accesses can bypass other cache accesses during decoupled cycles. For example, loads can bypass stores during decoupled cycles, but stores cannot pass loads or other stores during decoupled cycles. Both loads and stores can bypass a touch load (see **6.9.1.2 Touch Load**). For example timings of decoupled cache accesses, refer to Figures 6-20, 6-22 and 6-23.

If both the data cache and the instruction cache need the bus at the same time, the instruction cache has priority unless the data cache must perform a snoop copyback or an **xmem** transaction, or the data cache requests the bus after being retried and forced off the bus. If the data cache is waiting to perform a line fill and decoupling is enabled, the cache is available. If the cache needs to copyback a line to memory, the cache is unavailable.

6.8 DATA CACHE ACCESSES

During a data cache access, the actions taken by the cache depend on the state of the line. Each data cache line can be in one of four states. These states reflect the state of the line with respect to memory, and whether or not the processor has exclusive ownership of the cached data. The state of each data cache line is indicated by the three state bits in each line. The following are the four possible data cache line states:

1. Invalid—The data in this line is no longer the most recent copy of the data and should not be used. Refer to **6.9 Cache Control and Maintenance** for more information on invalidating the cache.

2. Shared Unmodified—The data in this line is shared among processors, so other caches may have a copy of this line. However, this line is unmodified with respect to memory.

3. Exclusive Modified—Only one processor has a copy of the data from this line in its cache, and the line has been modified with respect to memory (dirty). Note that if any word in the line is dirty, then the entire line is dirty.

4. Exclusive Unmodified—Only one processor has a copy of the data from this line in its internal cache, and the line is unmodified with respect to memory.

During a data cache access, the line that contains the data being read or written by the processor may change state. The state of the cache line after the access depends on the type of access and whether the access resulted in a hit or a miss. For a complete explanation of these states and how transitions between states occur, refer to **Section 11 System Hardware Design.**

When an exclusive modified line in the data cache is to be replaced because of a cache miss, the line is flushed to memory before the access is completed. This process of flushing a dirty cache line to update memory is called copyback.

Figure 6-14 shows the data cache actions due to a read. It is assumed in Figure 6-14 that a physical address has already been generated by address translation. Note that the timing for the external signals in this section is only accurate to within a half-clock cycle and is included for reference only.

The following paragraphs describe the actions taken by the MC88110 due to a data cache read hit, read miss, write hit, and write miss.
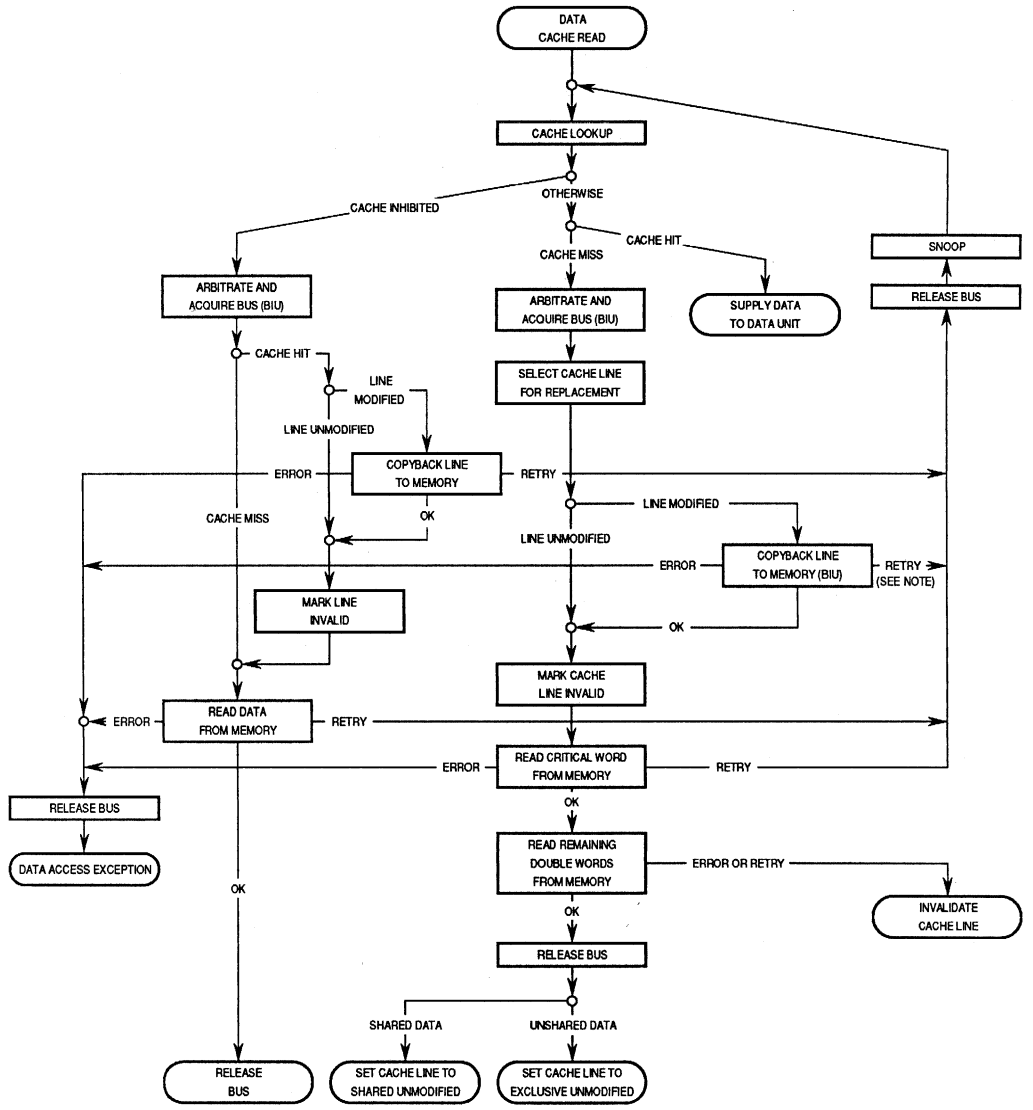
Figure 6-14. Data Cache Read Flowchart

NOTE: This retry will never be generated by an MC88110, since two MC88110s will never contain modified data at the same physical address in their caches. However, another device may generate a retry at this point.

6

6.8.1 Data Cache Read Hit

If a read operation (not cache inhibited) results in a data cache hit, the appropriate evenly aligned double word (indicated by bits 4–3 of the address) is accessed from the cache line and no state transition occurs. The double word is transferred to the data unit. Access time for the data cache on a read hit is one clock cycle. Figure 6-15 shows an example of the timing for a read hit.

Figure 6-15. Data Cache Read Hit Timing

On a cache inhibited read hit, the cache line is invalidated and the new data is read from memory but not placed in the cache. If the cache inhibited read hit is to an exclusive modified line, the line is copied back to memory before being invalidated, and the new data is read from memory but not placed in the cache.

6.8.2 Data Cache Read Miss

On a data cache read miss, a line in the cache is selected to hold the data which will be fetched from memory. If the selected line is marked exclusive modified, the line is sent to the BIU to be copied back to memory. When the copyback is complete, or if the selected line has not been modified, then the line is marked invalid and the physical address of the aligned double word containing the missed data is sent to the BIU along with a request to retrieve the missed cache line. The BIU arbitrates for the bus and initiates an 8-word burst transfer read request to fill the line. If decoupling is enabled, cache accesses can be performed while the cache is waiting for the line fill to begin.

The data cache line fill always begins with the evenly aligned double word containing the missed data (i.e., critical word first), followed by the subsequent double word(s) in the line, if any. If the double word containing the missed data is not the first double word in the line, the fill wraps around and fills the double word(s) at the beginning of the line. When the missed word is received from the bus, it is simultaneously written to the cache

and forwarded to the data unit, which stores the word in the register file. If needed, subsequent data from the line fill is forwarded (streamed) to the data unit as it is received. The data cache remains busy and inaccessible to the processor until the line fill is complete.

Figure 6-16 shows an example of the timing for a data cache read miss. In Figure 6-16, the address of the read ($700010) is determined by adding the offset found in **r16** to the base address in **r4**. The double word containing the missed data is not the first double word on the line boundary. Therefore, the line fill wraps around and fill the double words at the beginning of the line. Note that there are two clock cycles in which the data cache is decoupled. Since a write cannot run decoupled with a read, the data cache waits for the read to write-back before it can perform the write access.

LEGEND:

☐ INSTRUCTION FETCH
▨ EXECUTE
▼ FEED FORWARD
■ WRITE-BACK
▨ DELAYED
▨ DELAYED IN LOAD BUFFER OR STORE RESERVATION STATION
▨ CACHE ACCESS
▨ CACHE BUSY AND NOT AVAILABLE
▨ CACHE READ FROM BIU
▨ CACHE DECOUPLED (BUSY BUT AVAILABLE)
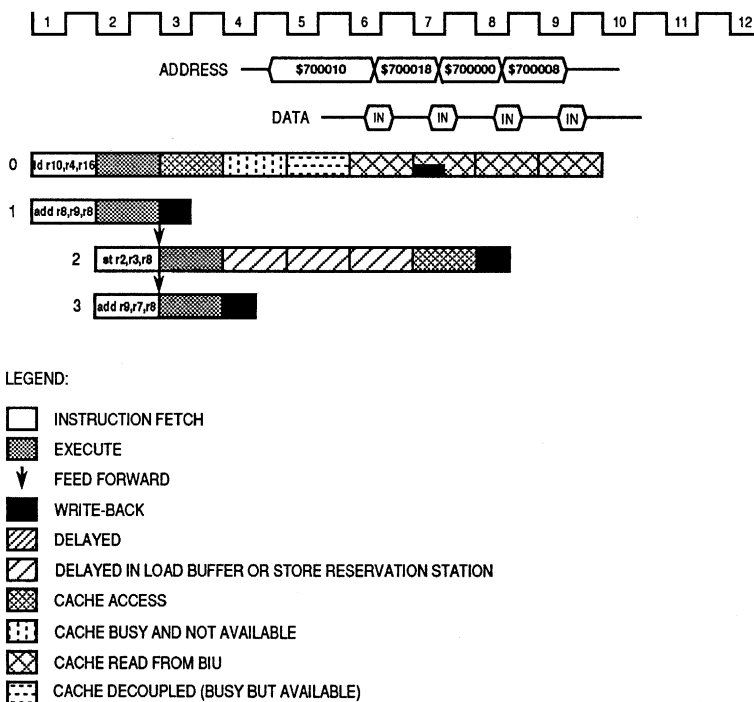
Figure 6-16. Data Cache Read Miss—No Copyback Timing

Figure 6-17 shows an example of the timing for a read miss when the line chosen for replacement is marked exclusive modified; therefore, the line must be sent to the BIU to be copied back to memory. There are six cycles of dirty line copyback and then the cache is decoupled for one cycle. Since the following instruction is also a read, which cannot run decoupled with the first read, the second instruction waits until the cache performs the read from the BIU before it can execute the second read and write-back.
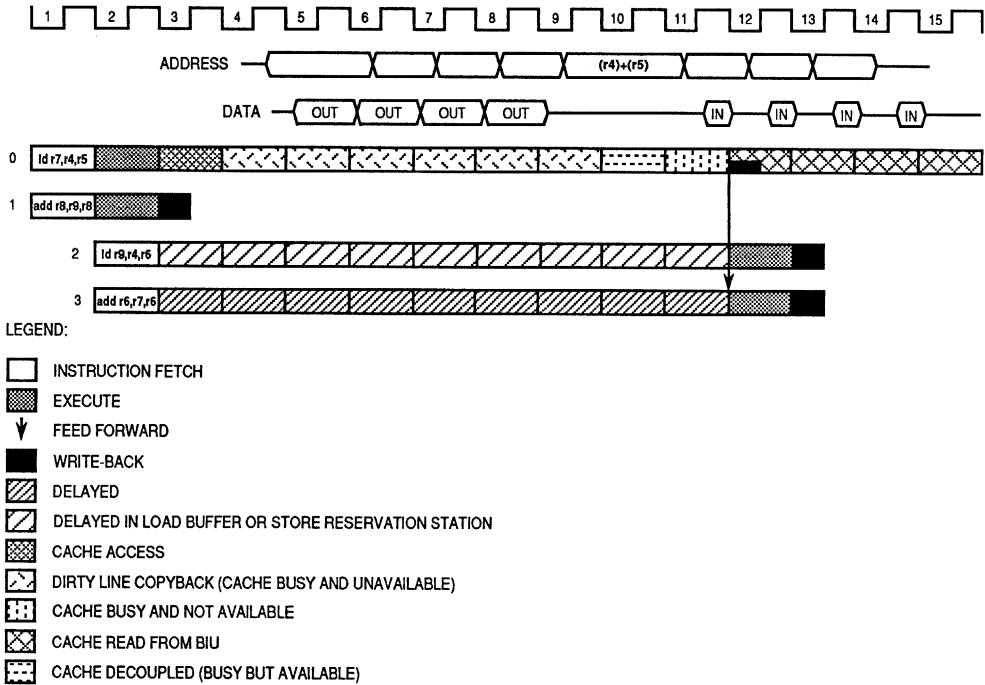
Figure 6-17. Data Cache Read Miss with Copyback Timing

If, at the beginning of a line fill, a snooping processor on the bus recognizes the address of the missed word as global and has a modified copy of the data in its cache, it will assert the retry signal, $\overline{\text{ARTRY}}$. Upon receipt of the retry signal the BIU will abort the line fill transaction and relinquish the bus. The snooping processor will acquire the bus and update memory with its modified copy of the line. The initiating processor will then start the read transaction over again from the logical address and cache lookup.

If a bus error is encountered during a copyback, then a data access exception is taken. A data access exception is also taken if a bus error is encountered on the bus access to the missed word. If the bus error occurs on any word other than the missed word in the line transfer then the fetched line is simply marked invalid. If no bus error is encountered, then the line is marked either shared unmodified (if the shared input signal $\overline{\text{(SHD)}}$ is asserted for this access) or exclusive unmodified (if the $\overline{\text{SHD}}$ input signal is negated for this access).

On a cache inhibited read access which misses the cache, data is read directly from memory but not placed in the cache.

6.8.3 Data Cache Write Hit

Writes that hit the data cache are handled according to the memory update mode of the data being accessed. Figure 6-18 shows the data cache actions due to a write hit. It is assumed in Figure 6-18 that a physical address has already been generated as a result of the address translation.
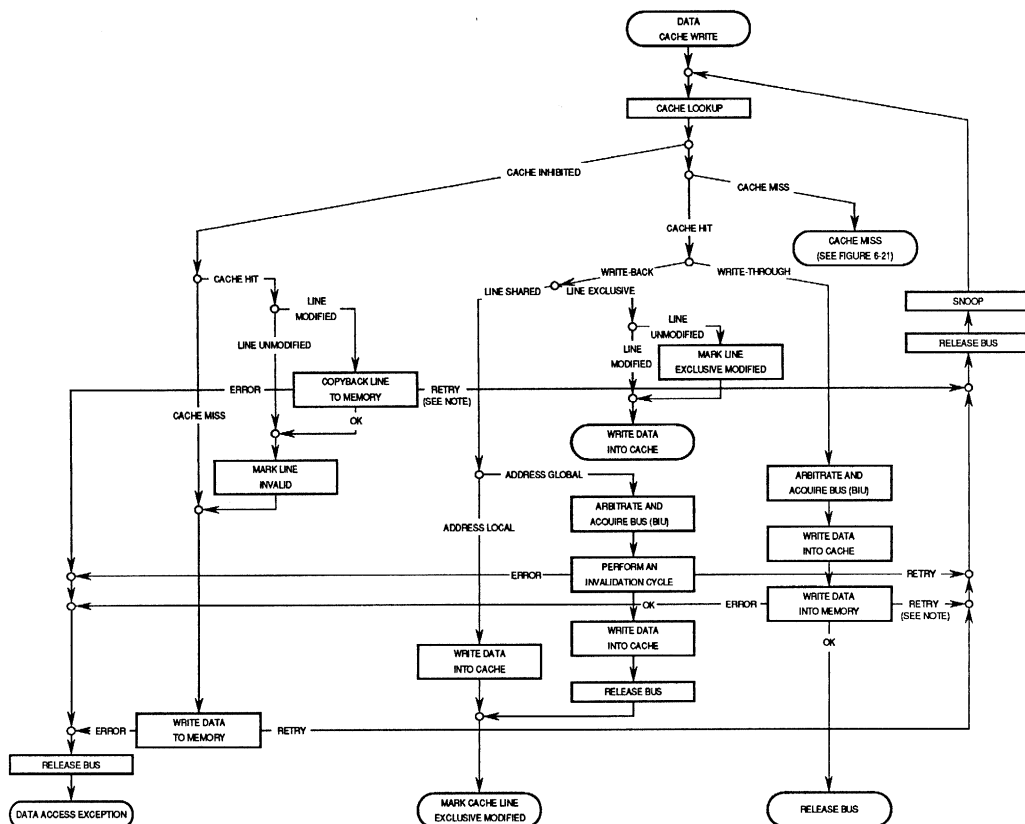
NOTE: This retry will never be generated by an MC88110, since two MC88110s will never contain modified data at the same physical address in their caches. However, another device may generate a retry at this point.

Figure 6-18. Data Cache Write Hit Flowchart

On a write hit to an exclusive line in write-back mode, data is simply written to the cache. If the line is unmodified, then the state of the line is changed to exclusive modified. Instruction 0 in Figure 6-19 is an example of a write hit to an exclusive modified or unmodified line. After the execute cycle, there are two cycles for data alignment. Actually only one cycle is used for data alignment. The other cycle is needed to guarantee a precise exception model. For a detailed explanation of store instruction timing, refer to **Section 9 Instruction Timing and Code Scheduling Considerations**. In this example, the cache is only busy for one clock cycle.

MC88110 USER'S MANUAL MOTOROLA

If the address is local on a write hit to a shared line in write-back mode, then the data is written into the cache and the line is marked exclusive modified. If the access is to a global page, then an invalidation bus transaction is performed first. The invalidation transaction notifies other caches on the bus that any local copy they may currently have of the cache line is no longer valid. The invalidation cycle is similar to a write cycle but the normal write cycle bus latency is avoided because data is not actually written. Instruction 2 in Figure 6-19 is a best-case example of a write hit to an unmodified line in write-back mode. In this case, the cache is unavailable for three clock cycles.
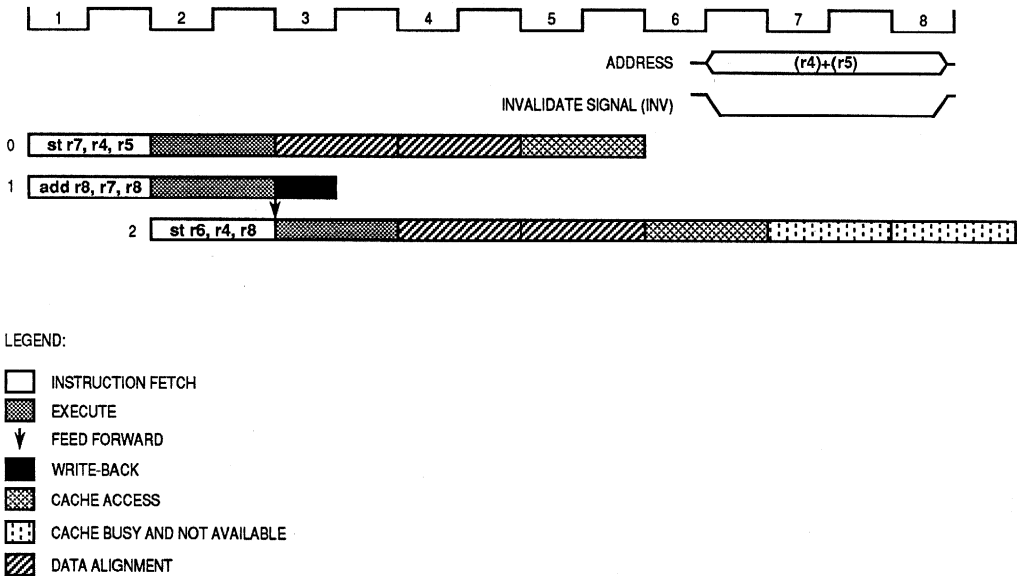
Figure 6-19. Data Cache Write Hit in Write-Back Mode Timing

On a write hit in write-through mode or on a store-through access, data is written to both the cache and to memory, and no cache state transition occurs. The invalidate signal (INV) is asserted during the write cycle so that any other cache on the bus which has a copy of the line containing the data will invalidate its copy of the line. If the write cycle experiences a bus error, the cache is still updated.

On a cache inhibited write hit, the line is flushed if it is modified, and the data is written to memory but not written into the cache. Figure 6-20 shows examples of the timing for write hits in write-through and cache inhibited mode with the cache decoupling feature enabled. Instruction 0 in this example shows a typical write hit to a line in write-through or cache inhibited mode. In this case, the cache is busy for four clock cycles; however, during two of the cycles, the cache is available for other accesses since cache decoupling is enabled. Thus, instruction 4 is allowed to access the cache during clock 6. Instruction 6 illustrates the special case in which a cache inhibited access hits an exclusive modified line in the cache. In this case, the cache line is copied back to

memory before the write access occurs, and the cache is busy for ten clock cycles. Again, since cache decoupling is enabled, the cache is busy but available during clock cycle 15.

LEGEND:

☐ INSTRUCTION FETCH

▨ EXECUTE

↓ FEED FORWARD

■ WRITE-BACK

▨ DELAYED IN LOAD BUFFER OR STORE RESERVATION STATION

▨ CACHE ACCESS

▨ DIRTY LINE COPYBACK (CACHE BUSY AND UNAVAILABLE)

▨ CACHE WRITE TO BIU

▨ CACHE DECOUPLED (BUSY BUT AVAILABLE)

▨ CACHE BUSY AND NOT AVAILABLE

▨ DATA ALIGNMENT
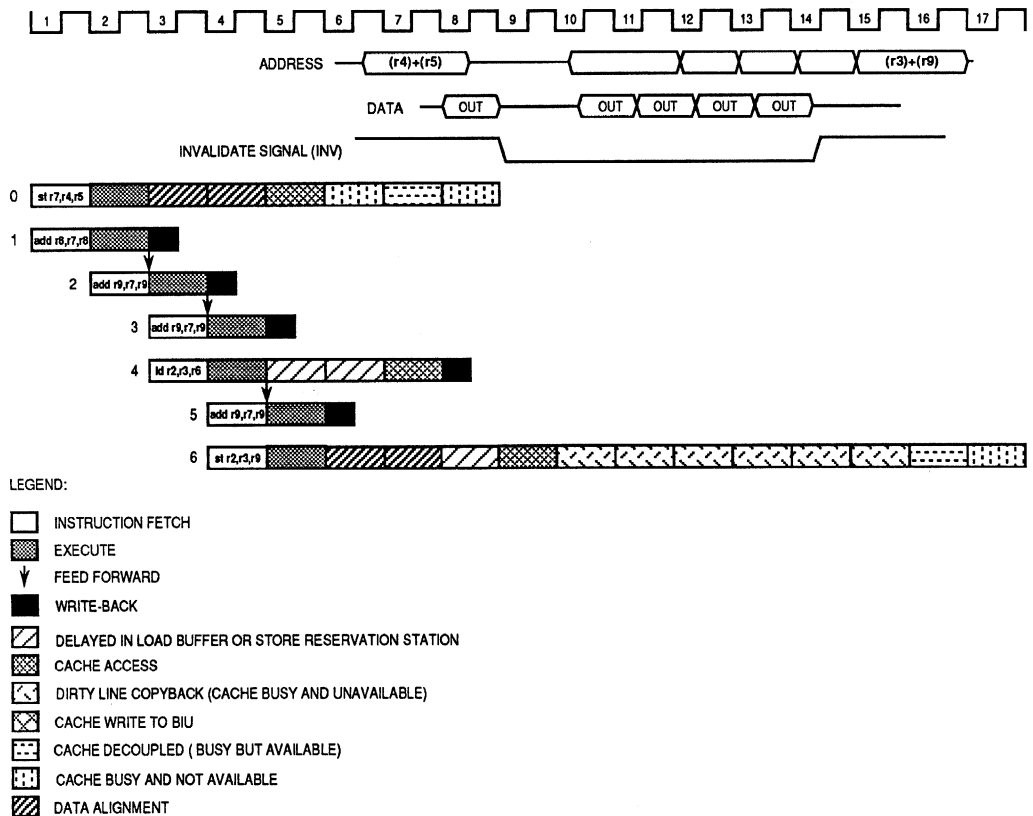
Figure 6-20. Write Hit in Write-Through or Cache Inhibited Mode Timing

6.8.4 Data Cache Write Miss

Writes that miss the data cache are handled according to the memory update mode of the data being accessed. Figure 6-21 shows the data cache actions due to a write miss. It is assumed in Figure 6-21 that a physical address has already been generated by address translation.
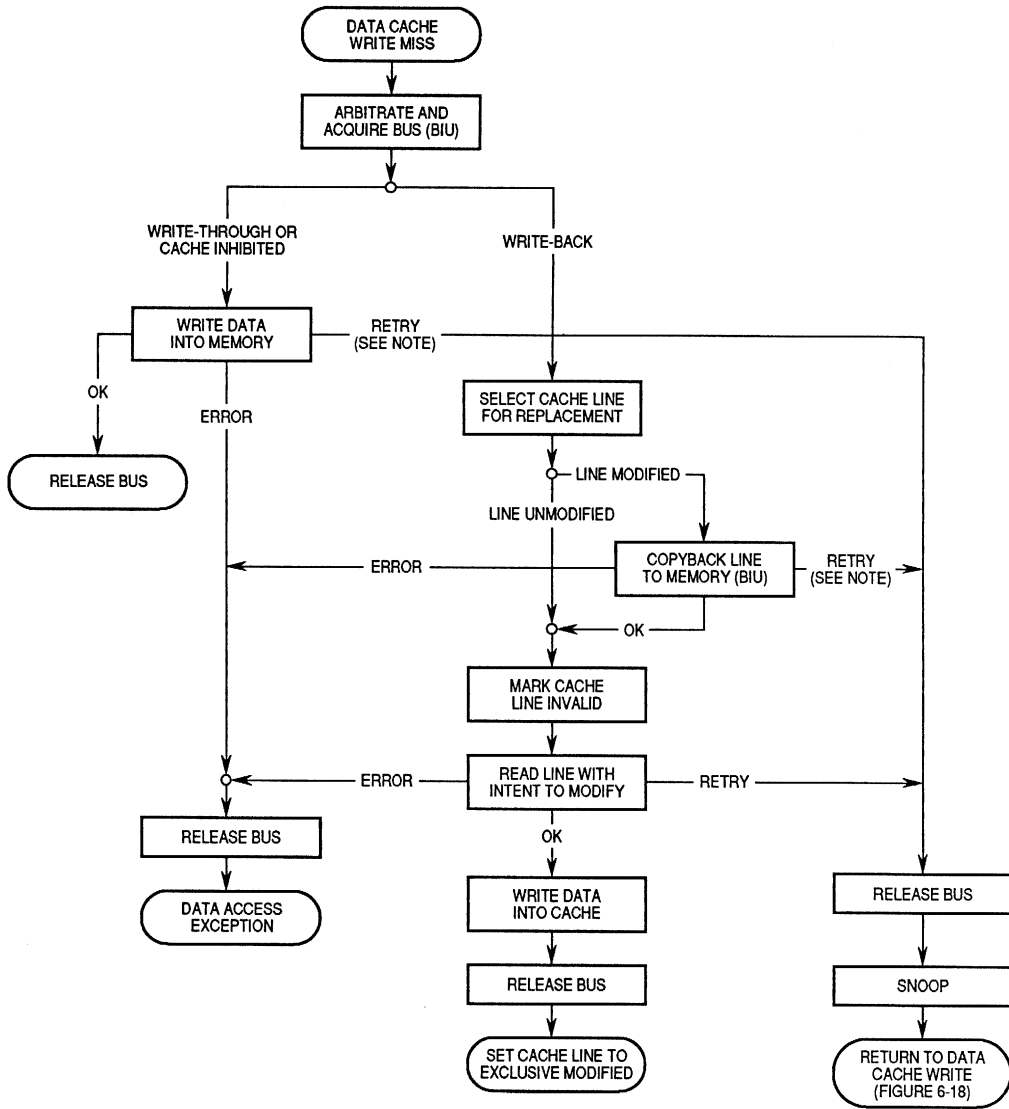
Figure 6-21. Data Cache Write Miss Flowchart

NOTE: This retry will never be generated by an MC88110, since two MC88110s will never contain modified data at the same physical address in their caches. However, another device may generate a retry at this point.

On a write miss in write-back mode, a cache line is first selected to receive data from memory. If the selected line is marked exclusive modified, the line is sent to the BIU to be copied back to memory. If the selected line is not modified or when the copyback is complete, the physical address of the missed data is sent to the BIU along with a request to retrieve the missed cache line. The BIU arbitrates for the bus and initiates an 8-word read-with-intent-to-modify burst transfer cycle. The data cache line fill always begins with the evenly aligned double word containing the missed data (i.e., critical word first) and is followed by the subsequent double word(s) in the line. The write transaction occurs simultaneously with the line fill, merging the write data with the current data read from external memory. If the double word containing the missed data is not the first double word in the line, the fill wraps around and fills the double word(s) at the beginning of the line. If a bus error is encountered on the dirty line flush or the line fill operation, a data access exception is taken.

The special read-with-intent-to-modify cycle is like a normal read cycle but has the side effect of broadcasting to other processors on the bus that the line being fetched will be modified; thus, the other processors should invalidate any local copy of the line they may have. If another processor on the bus recognizes the address as global and has an exclusive modified copy of the data in its cache then it will assert the retry signal (ARTRY). Upon receipt of the retry signal, the BIU will abort the line fill transaction and relinquish the bus. The snooping processor will acquire the bus and update memory with its copy of the line. The initiating processor will then start the write transaction over again beginning with a cache lookup from the logical address. The write transaction will occur simultaneously with the line fill, merging the write data with the current data read from external memory. Once the merged write transaction and line fill are complete, the line is marked exclusive-modified.

Instruction 0 in Figure 6-22 is an example of a write miss to an exclusive modified line in the cache with decoupling enabled. Since the line to be replaced is modified, the line is copied back to memory and then a read-with-intent-to-modify transfer cycle is performed to fill the cache line. Since decoupling is enabled, instruction 4 is allowed to access the cache during clock 12; however, instruction 6 is forced wait until instruction 0 has completed before being allowed access to the cache, since only one instruction can access the cache during clock 12.

LEGEND:

☐	INSTRUCTION FETCH
▨	EXECUTE
↓	FEED FORWARD
■	WRITE-BACK
▨	DELAYED IN LOAD BUFFER OR STORE RESERVATION STATION
▨	CACHE ACCESS
⧄	DIRTY LINE COPYBACK (CACHE BUSY AND UNAVAILABLE)
⊠	CACHE READ FROM BIU
⋯	CACHE DECOUPLED (BUSY BUT AVAILABLE)
▨	DATA ALIGNMENT
⊞	BUSY AND NOT AVAILABLE
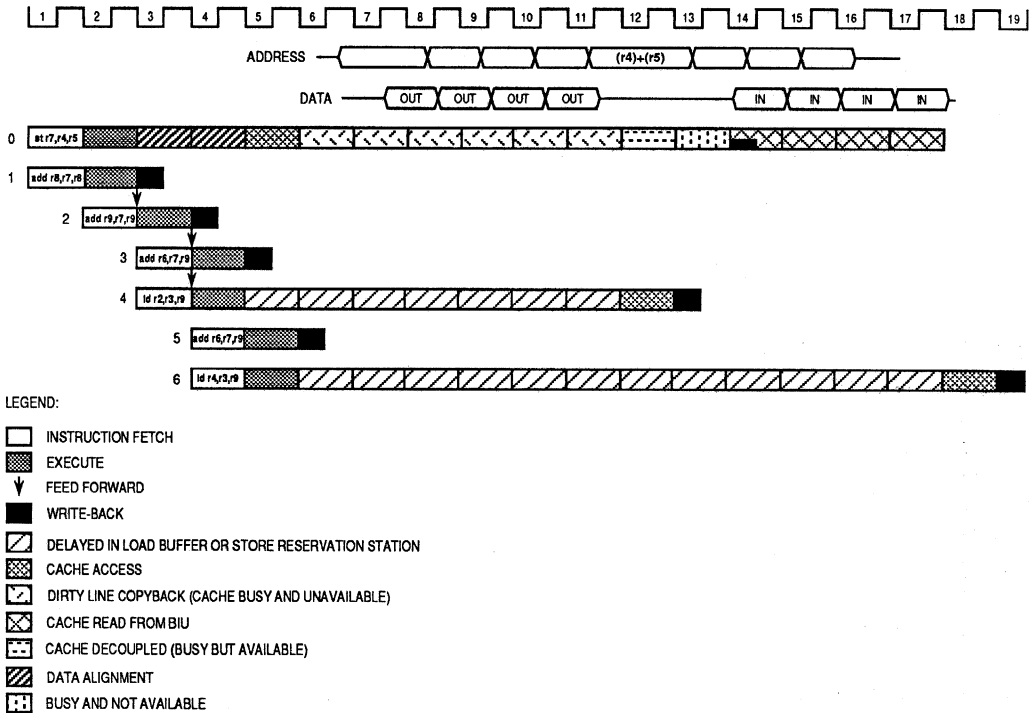
Figure 6-22. Write Miss with Copyback Timing

Instruction 0 in Figure 6-23 is an example of a write miss when the line to be replaced is exclusive or shared unmodified and decoupling is enabled. Since the line is unmodified, no copyback is required, so the cache is busy for only 8 clock cycles. Also, because decoupling is enabled, instruction 4 can access the cache during clock 7. Since the cache cannot be accessed during the line fill, instruction 6 must wait for instruction 0 to complete before being allowed access to the cache.

LEGEND:

☐ INSTRUCTION FETCH
■ WRITE-BACK
▨ DELAYED IN LOAD BUFFER OR STORE RESERVATION STATION
▨ CACHE ACCESS
▧ CACHE READ FROM BIU
▨ CACHE DECOUPLED (BUSY BUT AVAILABLE)
▨ DATA ALIGNMENT
▨ CACHE BUSY AND NOT AVAILABLE
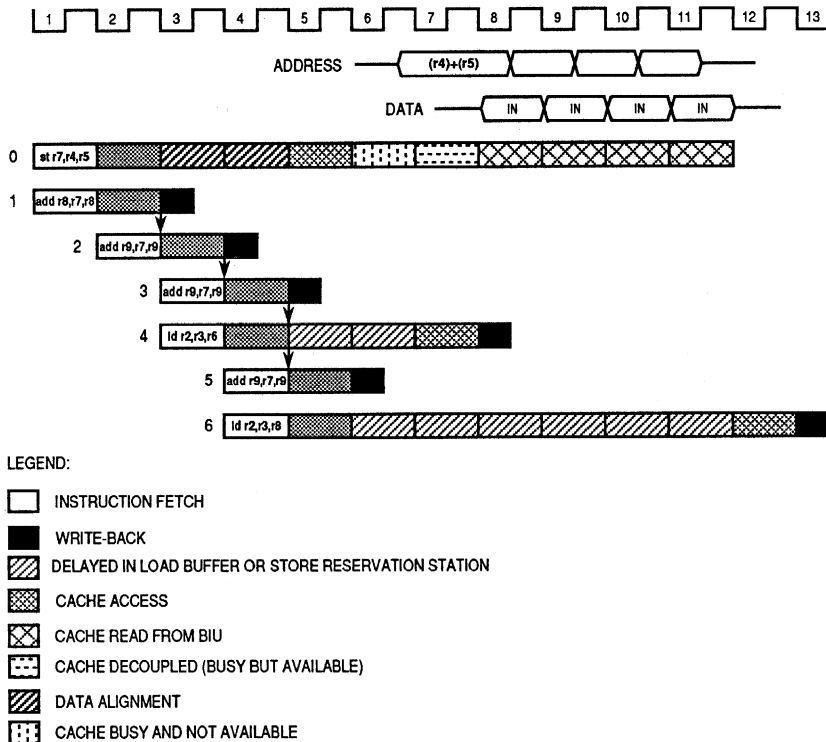
Figure 6-23. Write Miss—No Copyback Timing

On a write miss in write-through mode (or on a store-through access), data is written to memory only—no line is allocated in the cache and no data is written to the cache. The invalidate signal ($\overline{INV}$) is asserted during the write cycle so that any other cache on the bus which has a copy of the line containing the data will invalidate its copy.

On a cache-inhibited write access which misses the cache, data is written to memory but no data is placed in the cache. No state transition occurs.

The timing for write misses in write-through or cache inhibited mode is the same as the timing for write hits in write-through or cache inhibited mode, as long as the cache inhibited write does not cause a copyback. Therefore, instruction 0 in Figure 6-20 shows an example of a write miss to a line in write-through or cache inhibited mode.

6.8.5 Data Cache xmem Accesses

The MC88110 supports an exchange memory (**xmem**) instruction that is a combination of a load and store instruction. The **xmem** instruction is normally a read access followed by a write access. However, the **xmem** instruction can also function as a write access followed by a read access if the XMEM bit is set in the DCTL. The **xmem** accesses are cache inhibited, so if a cache hit occurs to a modified line, the line is first copied back to memory and invalidated. Figure 6-24 shows a flowchart of how **xmem** operates. **Section 11 System Hardware Design** provides functional timing for the **xmem** instruction.

6.9 CACHE CONTROL AND MAINTENANCE

Software can either issue cache control instructions or set or clear bits in the cache control registers to control and maintain the instruction and data caches of the MC88110. This section describes the following cache control and maintenance topics: user-mode cache control features, cache control registers, the invalidate command, the flush command, and cache freezing.

6.9.1 User-Mode Cache Control Features

Four features are implemented in the MC88110 which provide explicit control over caching behavior. These features allow performance to be improved in cases where the programmer has some specific knowledge about how or when data will be used. These new features include:

- Cache Bypassing on Stores (Store-Through)
- Cache Preloading (Touch Load)
- Forced Dirty Line Flush (Flush Load)
- Line Allocation Without Line Fill (Allocate Load)

Three of the cache control features are specified by performing signed loads of various sizes with **r0** as the destination register: touch load, flush load, and allocate load. The touch, flush, and allocate load accesses are visible on the external bus through the transfer code (TC3–TC0) pins. These pins will read 0010 if the processor is in user mode during a cache control access, and they will read 0110 if the processor is in supervisor mode during a cache control access.

Past and future implementations which do not support these three cache control features will still be compatible with code employing these features because they do not affect the functionality of the user program. Whether or not the memory references specified by these features are actually performed is irrelevant to the program; however, performance may be affected.
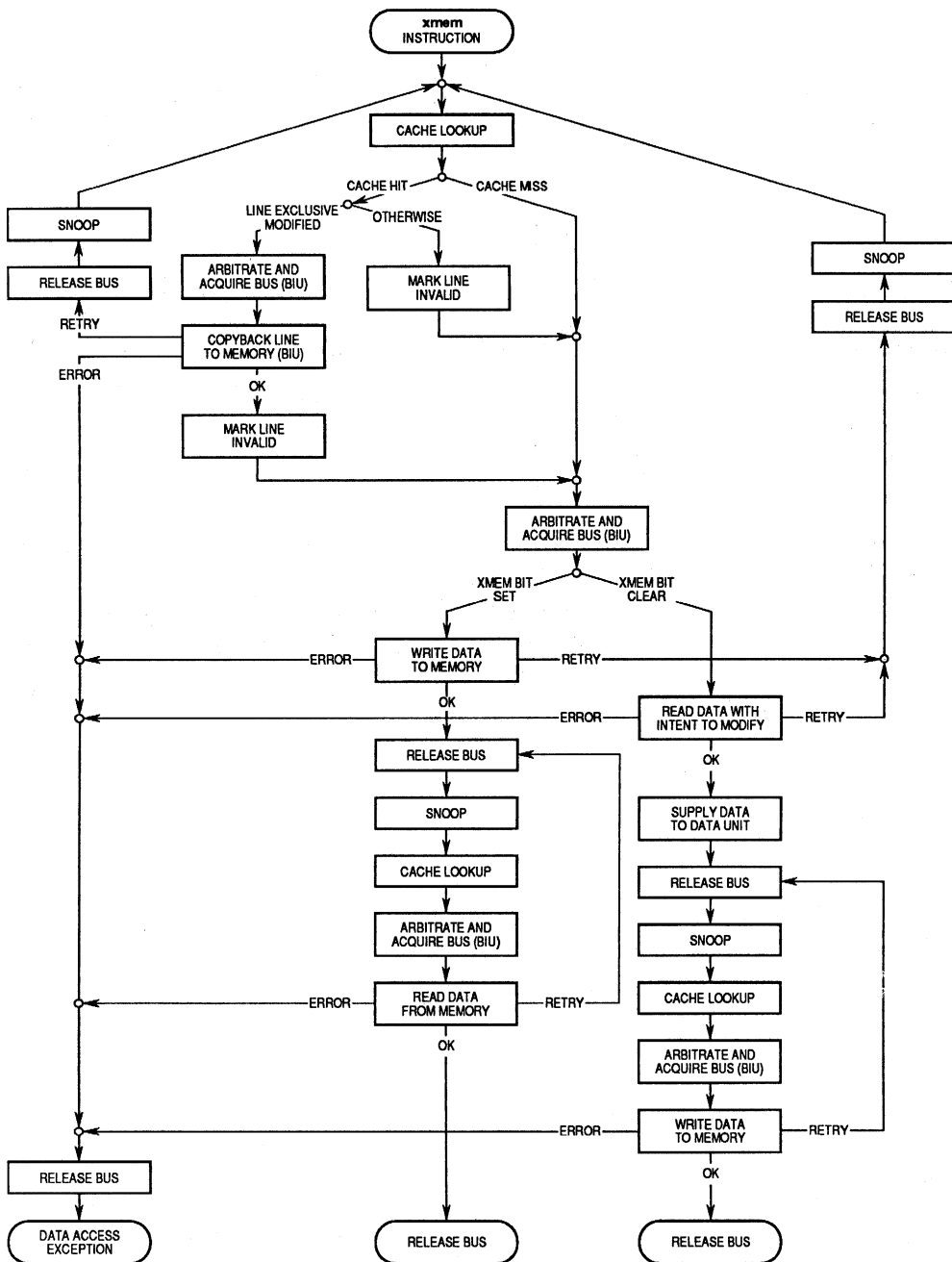
Figure 6-24. xmem Flowchart

6.9.1.1 STORE-THROUGH. When specified, the store-through option unconditionally causes the **st** operation to write through the cache directly into memory. If a store-through access hits in the cache, the data is written both to memory and the cache, but the state of the cache line is not changed. When the store-through misses in the cache, no line is allocated in the cache, and the access simply writes directly to memory, bypassing the cache completely. Note that this operation is identical to the cache accesses in write-through mode.

The store-through option serves two purposes. First, it provides a mechanism to force a particular piece of data to write through the cache and into memory even if the access is to a write-back page. Second, it provides a way to prevent data that is being stored that might miss the cache and that the program knows will not be reused soon from replacing a potentially more useful line in the cache. This not only avoids the wasted time of moving a line out of the cache and another back in, but also improves the hit rate of subsequent **ld** operations to the cache lines which might have been replaced. Store-through is specified by a **.wt** (for write-through) extension on any triadic register addressing form of the **st** instruction.

6.9.1.2 TOUCH LOAD. The touch load option allows data to be loaded into the cache under user program control. Normally, data is brought into the cache only when it is needed. This can lead to instruction execution stalls due to dependencies on data which must be read from main memory. In many cases, however, the need for data can be predicted. By requesting data to be read into the cache ahead of its actual use, the latency of the memory system can be overlapped with useful work, and stalls due to long latency cache misses can be minimized.

A touch load is specified by a *byte* load to **r0** as shown in the following example **ld** instructions:

> **ld.b**   r0,rS1,rS2
>
> **ld.b**   r0,rS1[rS2]
>
> **ld.b**   r0,rS1,IMM16

If the data specified by the effective address of the touch load operation is not already in the cache, then it is brought into the cache and replaces an existing line if necessary (just as a normal load miss would). A touch load to a cache inhibited line is treated as a normal cache inhibited **ld** operation.

A touch load is similar in most respects to normal loads except for two important distinctions. First, a touch load never generates an exception, and, therefore, the machine never needs to recover from one. This means that a touch load can be retired from the history buffer as soon as it enters the data unit rather than waiting until the load completes execution. Second, although a touch load operation may bring data into the cache, it does not write a result into the register files. Thus, load operations executing during a touch load do not need to run in program sequence with the touch load and can be allowed access to the cache while waiting for the touch load operation's line fill to begin.

6

6.9.1.3 FLUSH LOAD. The flush load option forces a dirty cache line out to memory. Normally, dirty cache lines are copied back to memory only as a side effect of needing to allocate a new cache line. However, it is sometimes convenient to be able to flush data in the cache to immediately update the memory image. For example, the user may store several data words to memory which get filtered by the cache and never actually update memory. In this case, the flush load option could be used to flush the data words from the cache out to memory. Note that no actual load operation is performed, but the operation is encoded as a load instruction.

A programmer may perform multiple store operations to the cache in write-back mode, and then use the flush load option to write the data to memory in a single burst transaction, all from user-mode code; thus, the flush load option provides performance advantages over other methods of keeping memory coherent with the cache. Placing a memory page in write-through mode or using the store-through option may have an undesirable performance impact because of the multiple individual bus transactions which would occur. Also, the time required to flush a line from supervisor mode may be prohibitive.

A flush load is specified by a *word* load to **r0** as shown in the following example **ld** instructions:

    ld   **r0,rS1,rS2**

    ld   **r0,rS1[rS2]**

    ld   **r0,rS1,IMM16**

When a flush load operation hits a dirty line in the data cache, the line is flushed out to memory and the modified bit for the line is cleared. On a cache miss, the flush load is treated as a NOP.

6.9.1.4 ALLOCATE LOAD. It is sometimes known in advance that an entire cache line is going to be overwritten. In these cases, performance can be improved if the overhead of fetching a new line from memory that is going to be overwritten can be avoided. The allocate load option provides this capability. Allocate load allows the user to allocate a line in the cache for a subsequent store operation while avoiding the normal line fill memory transaction. Instead, this option allocates a line in the cache, as any normal load does on a cache miss, but performs only a single-beat invalidation transaction on the bus rather than a full line fill bus transaction.

The allocate load option should be used with this caution: if the sequence of stores which is overwriting the allocated line is interrupted, it is possible that the partially valid line could be pushed out to memory. However, upon returning from the interrupt, the remaining stores in the sequence will be completed and the memory state will be corrected. Thus the invalid version of the line in memory will only have been a transient phenomenon.

An allocate load is specified by a *half-word* load to **r0** as shown in the following example **ld** instructions:

> **ld.h r0,rS1,rS2**
>
> **ld.h r0,rS1[rS2]**
>
> **ld.h r0,rS1,IMM16**

When allocate load is used on a cache inhibited access, no line is allocated but the single-beat bus transaction is still performed. On a data cache hit, allocate load is simply a NOP. An allocate load is also a NOP in the case of an exception.

6.9.2 Cache Control Registers

Many of the cache control features supported by the MC88110 are initiated by writing information to the cache control registers. The cache control registers can be accessed via the **stcr** and **ldcr** instructions, which are supervisor instructions. Note that, unlike other control registers, the **xcr** instruction is not valid for the cache control registers.

The MC88110 has the following cache control registers: the instruction MMU/cache/TIC command register (ICMD), the instruction MMU/cache/TIC control register (ICTL), the instruction system address register (ISAR), the data MMU/cache command register (DCMD), the data MMU/cache control register (DCTL), and the data system address register (DSAR). These registers are described in detail in the following paragraphs.

6.9.2.1 INSTRUCTION MMU/CACHE/TIC COMMAND REGISTER (ICMD).

The ICMD (see Figure 6-25) controls instruction cache flushing, instruction cache and ATC invalidation, and instruction MMU probing. The desired action is initiated by writing the appropriate command code to the ICMD using the **stcr** instruction. The command code is contained in the four least significant bits (3–0) of the data written to the ICMD; the other 28 bits should be zero. Reading the ICMD will return all zeros. Table 6-1 lists the command codes defined for the ICMD.

```
31                                                              4  3        0
┌──────────────────────────────────────────────────────────┬──────────────┐
│                                                            │ COMMAND CODE │
└──────────────────────────────────────────────────────────┴──────────────┘
```

▒ UNDEFINED-RESERVED FOR FUTURE USE
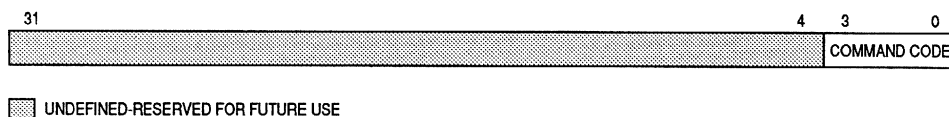
Figure 6-25. Instruction MMU/Cache Command Register

Table 6-1. ICMD Command Codes

Code	Command
0000	Reserved
0001	Invalidate Instruction Cache and TIC
0010	Invalidate TIC
0011	Reserved
0100	Reserved
0101	Invalidate Instruction Cache Line (see Note)
0110	Reserved
0111	Reserved
1000	MMU Probe Supervisor
1001	MMU Probe User
1010	Invalidate All Supervisor ATC Descriptors
1011	Invalidate All User ATC Descriptors
11xx	Reserved

Note: The cache line affected by this command is specified
in the instruction system address register (ISAR).

6.9.2.2 INSTRUCTION MMU/CACHE CONTROL REGISTER (ICTL). The ICTL (see Figure 6-26) controls the operating modes for the instruction cache, TIC, and the instruction MMU. The ICTL includes mask bits for specifying the BATC block size (512K-bytes–64M-bytes).
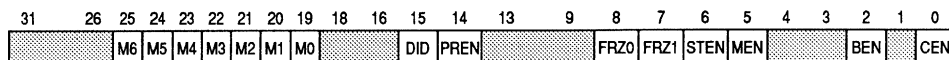
UNDEFINED-RESERVED FOR FUTURE USE

Figure 6-26. Instruction MMU/Cache Control Register

M6–M0—Instruction MMU BATC Block Size Selection Bits

The block sizes mapped by the BATC can be programmed by setting bits M6–M0 in the ICTL according to Table 6-2.

Table 6-2. Instruction MMU BATC
Block Size Selection Settings

Instruction BATC Size Mask Bits							Block Size
M6	M5	M4	M3	M2	M1	M0	
1	1	1	1	1	1	1	64M-Byte
0	1	1	1	1	1	1	32M-Byte
0	0	1	1	1	1	1	16M-Byte
0	0	0	1	1	1	1	8M-Byte
0	0	0	0	1	1	1	4M-Byte
0	0	0	0	0	1	1	2M-Byte
0	0	0	0	0	0	1	1M-Byte
0	0	0	0	0	0	0	512K-Byte
Any Other Combination							Undefined

DID—Double Instruction Issue Enable/Disable

When double instruction issue is enabled, the instruction unit will attempt to issue two instructions each clock cycle. When double instruction issue is disabled, the instruction unit will attempt to issue only one instruction per clock. On reset, double instruction issue is enabled.

PREN—Branch Prediction Enable/Disable

When branch prediction is disabled, the branch reservation station is disabled. In this case, if a branch instruction with a data dependency is encountered, instruction issue will stall. When branch prediction is enabled, branches with data dependencies issue to the branch reservation station, and conditional instruction issue occurs in the predicted direction. On reset, branch prediction is disabled.

    0—Branch Prediction Disabled
    1—Branch Prediction Enabled

FRZ0—Instruction Cache Freeze Bank 0 Enable/Disable

When instruction cache freeze bank 0 is enabled, the first line (line 0) in each set in the instruction cache is frozen. On reset, instruction cache freeze bank 0 is disabled.

    0—Instruction Cache Freeze Bank 0 Disabled
    1—Instruction Cache Freeze Bank 0 Enabled

FRZ1—Instruction Cache Freeze Bank 1 Enable/Disable

When instruction cache freeze bank 1 is enabled, the first line (line 1) in each set in the instruction cache is frozen. On reset, instruction cache freeze bank 1 is disabled.

    0—Instruction Cache Freeze Bank 1 Disabled
    1—Instruction Cache Freeze Bank 1 Enabled

6

STEN—Instruction MMU Software Table Search Enable/Disable

When software table searches are disabled, a hardware table search is performed when a ATC miss occurs. When software table searches are enabled, a trap to the instruction MMU ATC miss vector occurs on an ATC miss, and no hardware table search occurs. On reset, instruction MMU software table searches are disabled.

    0—Instruction MMU Software Table Search Disabled
    1—Instruction MMU Software Table Search Enabled

MEN—Instruction MMU Enable/Disable

When the instruction MMU is enabled, address translations can occur via the BATC/PATC or table searches. If the instruction MMU is disabled, then the logical address for each memory location is the same as the physical address (identity translation), and the access/protection information (e.g., memory update mode, global/local page designations, etc.) is taken from the ISAP or IUAP. On reset the instruction MMU is disabled.

    0—Instruction MMU Disabled
    1—Instruction MMU Enabled

BEN—TIC Cache Enable/Disable

When the TIC is disabled, no instructions are fetched from the TIC cache and the TIC is not accessed or updated. On reset the TIC is disabled.

    0—TIC Disabled
    1—TIC Enabled

CEN—Instruction Cache Enable/Disable

When the instruction cache is disabled, instruction fetches pass directly to the bus interface unit and the instruction cache is not accessed or updated. On reset the instruction cache is disabled.

    0—Instruction Cache Disabled
    1—Instruction Cache Enabled

6.9.2.3 INSTRUCTION SYSTEM ADDRESS REGISTER (ISAR). The ISAR (see Figure 6-27) indicates the logical address for an instruction ATC probe or the physical address of an instruction cache line to be invalidated during a line invalidate operation.

31   0
| LOGICAL ADDRESS FOR ATC PROBE OR PHYSICAL ADDRESS OF CACHE LINE |
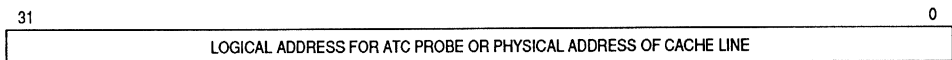
Figure 6-27. Instruction System Address Register

6.9.2.4 DATA MMU/CACHE COMMAND REGISTER (DCMD). The DCMD (see Figure 6-28) controls data cache flushing, data cache and ATC invalidation, and data MMU probing. The desired action is initiated by writing the appropriate command code to the DCMD using the **stcr** instruction. The command code is contained in the four least significant bits (3–0) of the data written to the DCMD; the other 28 bits should be zeros. Reading the DCMD will return all zeros. Table 6-3 lists the command codes defined for the DCMD.

31	4	3	0
		COMMAND CODE	

▒ UNDEFINED-RESERVED FOR FUTURE USE
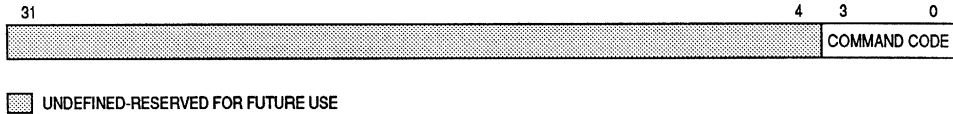
Figure 6-28. Data MMU/Cache Command Register

Table 6-3. DCMD Command Codes

Code	Command
0000	Flush Data Cache Page (copyback) (see Note)
0001	Invalidate Data Cache All
0010	Flush Data Cache All (copyback)
0011	Flush Data Cache All (copyback & invalidate)
0100	Flush Data Cache Page (copyback & invalidate) (see Note)
0101	Invalidate Data Cache Line (see Note)
0110	Flush Data Cache Line (copyback) (see Note)
0111	Flush Data Cache Line (copyback & invalidate) (see Note)
1000	MMU Probe Supervisor (see Note)
1001	MMU Probe User (see Note)
1010	Invalidate All Supervisor ATC Descriptors
1011	Invalidate All User ATC Descriptors
11xx	Reserved

Note: The cache line or page affected by this command is specified in the data system address register (DSAR).

6.9.2.5 DATA MMU/CACHE CONTROL REGISTER (DCTL).
The DCTL (see Figure 6-29) controls the operating modes for the data cache and the data MMU. The DCTL includes the mask bits for specifying the BATC block size (512K-bytes–64M-bytes).

UNDEFINED-RESERVED FOR FUTURE USE
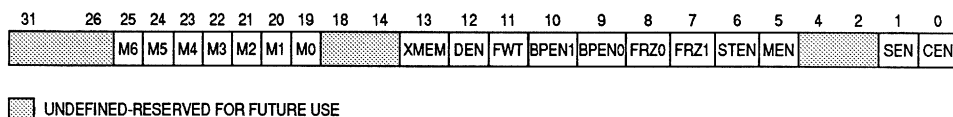
Figure 6-29. Data MMU/Cache Control Register

M6–M0—Data MMU BATC Block Size Selection Bits
The block sizes mapped by the BATC can be programmed by setting bits M6–M0 according to Table 6-4.

Table 6-4. Data MMU BATC Block Size Selection Settings

Data BATC Size Mask Bits							Block Size
M6	M5	M4	M3	M2	M1	M0	
1	1	1	1	1	1	1	64M-Byte
0	1	1	1	1	1	1	32M-Byte
0	0	1	1	1	1	1	16M-Byte
0	0	0	1	1	1	1	8M-Byte
0	0	0	0	1	1	1	4M-Byte
0	0	0	0	0	1	1	2M-Byte
0	0	0	0	0	0	1	1M-Byte
0	0	0	0	0	0	0	512K-Byte
Any Other Combination							Undefined

XMEM—**xmem** Instruction Control bit
When this bit is cleared, the **xmem** instruction performs a locked bus sequence consisting of a load followed by a store. When this bit is set, the **xmem** instruction performs a locked bus sequence consisting of a store followed by a load. On reset, the XMEM bit is cleared.

MC88110 USER'S MANUAL   MOTOROLA

DEN—Decoupled Cache Access Enable/Disable

When this bit is clear, decoupled accesses to the data cache are disabled regardless of the type of bus transaction or the status of the $\overline{\text{PTA}}$ input signal. When this bit is set, decoupled accesses are allowed under the control of the $\overline{\text{PTA}}$ signal (see **6.7 Data Cache Decoupling**). On reset, decoupled cache accesses are disabled.

    0—Decoupled Cache Accesses Disabled
    1—Decoupled Cache Accesses Enabled

FWT—Force Write-Through

When this bit is set, all stores are forced to write-through the data cache regardless of the page or block status or store-through instruction option; however, the FWT bit does not affect the normal operation of the $\overline{\text{WT}}$ pin.

BPEN1—Breakpoint Enable/Disable 1

When breakpoint 1 is disabled, the breakpoint registers do not cause a data access exception when a matching logical address is detected. When breakpoint 1 is enabled, the breakpoint registers cause a data access exception upon detecting a matching logical address. On reset, breakpoints are disabled. See **Section 8 Memory Management Unit** for detailed information on breakpoints.

    0—Breakpoints Disabled
    1—Breakpoints Enabled

BPEN0—Breakpoint Enable/Disable 0

When breakpoint 0 is disabled, the breakpoint registers do not cause a data access exception when a matching logical address is detected. When breakpoint 0 is enabled, the breakpoint registers cause a data access exception upon detecting a matching logical address. On reset, breakpoints are disabled. See **Section 8 Memory Management Unit** for detailed information on breakpoints.

    0—Breakpoints Disabled
    1—Breakpoints Enabled

FRZ0—Data Cache Freeze Bank 0 Enable/Disable

When data cache freeze bank 0 is enabled, the first line (line 0) in each set in the data cache is frozen. On reset, data cache freeze bank 0 is disabled.

    0—Data Cache Freeze Bank 0 Disabled
    1—Data Cache Freeze Bank 0 Enabled

FRZ1—Data Cache Freeze Bank 1 Enable/Disable

When data cache freeze bank 1 is enabled, the first line (line 1) in each set in the data cache is frozen. On reset, data cache freeze bank 1 is disabled.

    0—Data Cache Freeze Bank 1 Disabled
    1—Data Cache Freeze Bank 1 Enabled

STEN—Data MMU Software Table Search Enable/Disable

When software table searches are disabled, a hardware table search is performed when a ATC miss occurs. When software table searches are enabled, a trap to the data MMU ATC miss vector occurs on an ATC miss, and no hardware table search occurs. On reset, data MMU software table searches are disabled.

　0—Data MMU Software Table Search Disabled
　1—Data MMU Software Table Search Enabled

MEN—Data MMU Enable/Disable

When the data MMU is enabled, address translations can occur via the BATC/PATC or table searches. If the data MMU is disabled, then the logical address for each memory location is the same as the physical address (identity translation), and the access/protection information (e.g., memory update mode, global/local variable designations, etc.) is taken from the DSAP or DUAP. On reset, the data MMU is disabled.

　0—Data MMU Disabled
　1—Data MMU Enabled

SEN—Data Cache Snooping Enable/Disable

Snooping is disabled on reset.

　0—Data Cache Snooping Disabled
　1—Data Cache Snooping Enabled

CEN—Data Cache Enable/Disable

When the data cache is disabled, loads and stores go directly to the BIU and the data cache is not accessed or updated. On reset, the data cache is disabled. The timing for memory accesses with the cache disabled is identical to cache inhibited accesses.

　0—Data Cache Disabled
　1—Data Cache Enabled

6.9.2.6 DATA SYSTEM ADDRESS REGISTER (DSAR). The DSAR (see Figure 6-30) indicates the logical address for a data ATC probe or the physical address of a data cache line or page to be invalidated or flushed during a line invalidate operation.

31　　0

LOGICAL ADDRESS FOR ATC PROBE OR PHYSICAL ADDRESS OF CACHE LINE
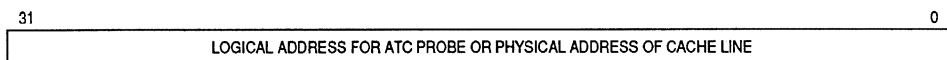
Figure 6-30. Data System Address Register

6.9.3 The Invalidate Command

The invalidation operation marks the state of each line in the desired cache (data cache, TIC, or instruction cache/TIC) as invalid without copying back any dirty lines to memory. Supervisor code initiates this operation by writing the appropriate command code to the ICMD or DCMD. The invalidation operation requires three clock cycles plus the time required to serialize the machine. The **stcr** which was used to write the invalidate command to the ICMD or DCMD causes the machine to serialize.

A more selective mechanism is also provided which allows any line in the instruction or data cache to be invalidated. This mechanism is invoked by first writing the physical address of the line to be invalidated into the DSAR or ISAR and then writing either an invalidate data cache line command to the DCMD or an invalidate instruction cache line command to the ICMD. Line invalidation requires two clocks plus serialization time. Refer to Table 6-5 for the number of clock cycles required to invalidate the entire cache or a single line in the cache.

6.9.4 The Flush Command

The flush operation causes all dirty lines in the data cache to be transferred out to memory and marks the transferred lines as unmodified. The flush operation can also be specified as a flush with invalidate command which causes all dirty lines in the data cache to be transferred out to memory and marks the transferred lines invalid. Supervisor code initiates these operations by writing the appropriate command code to the DCMD. No further instruction processing occurs during the flush (or flush and invalidate) operation, but the cache will snoop global bus transactions.

More selective flush and invalidate mechanisms are also provided which allow any line or page in the data cache to be flushed or flushed and invalidated. These mechanisms are invoked by first writing the physical address of the line or page to be flushed into the DSAR and then writing the appropriate command code to the DCMD. Table 6-5 lists the number of clock cycles necessary to flush or flush and invalidate a line, a page, or the entire cache.

6

Table 6-5. Clock Cycles for Data Cache Flush/Invalidate Commands

Command	Clock Cycles
Invalidate Data or Instruction Cache	
Line	5 + SER
All	5 + SER
Flush Data Cache	
Line	8 + SER + n(SCB) Where n = 0,1
Page	136 + SER + n(SCB+2) Where n = 0–128
All	136 + SER + n(SCB+2) Where n = 0–256
Flush and Invalidate Data Cache	
Line	8 + SER + n(SCB) Where n = 0,1
Page	136 + SER + n(SCB+2) Where n = 0–128
All	136 + SER + n(SCB+2) Where n = 0–256

NOTE : SER = The time required to serialize the machine
SCB (simple copyback) = The time required to copyback a dirty line to memory
n = The number of lines marked exclusive modified

6.9.5 Cache Freezing

Supervisor code has the ability to freeze one or both banks of the instruction and/or data caches. Setting the FRZ1 bit in the ICTL or DCTL freezes bank 1 in the instruction cache or data cache, respectively, and setting the FRZ0 bit in either register freezes bank 0 in the corresponding cache. When a bank is frozen, the data contained in that bank can be read and modified, but not replaced. On reset, the FRZ0 and FRZ1 bits are cleared. The flush and invalidate commands have priority over the cache freeze option; therefore, if a flush or invalidate is specified for a frozen cache bank, the MC88110 will ignore the fact that the bank is frozen and perform the specified function.

The cache freeze feature can be used to improve performance by allowing instructions or data to be kept in the cache for quick access. For the instruction cache, the freeze feature is useful for holding sections of code in the cache which need to be rapidly executed many times (e.g., an inner loop of a routine or an interrupt handler). For the data cache, the freeze feature is useful for keeping available frequently used data which could possibly be overwritten during normal operation.

To use the cache freeze feature, first freeze one bank of the cache, then load the other bank with the critical instructions or data, then unfreeze the first bank and freeze the other. To freeze a bank in the instruction cache, set the appropriate FRZx bit in the ICTL. To freeze a bank in the data cache, set the appropriate FRZx bit in the DCTL. To load the critical code into the instruction cache, the code must be executed so that it will be read into the free bank of the cache. To load data into the data cache, issue ld instructions. Recall that only one word must be loaded for a cache line to cause the entire line to be filled.

Data or instructions can be frozen into a bank 0 by using the invalidate command. First invalidate the entire cache by writing the appropriate command code to the ICMD or DCMD. Then, any code or data read into either cache will be read into bank 0. After the code or data has been read in, freeze bank 0. This method is not as flexible as the first because only bank 0 can be used and the entire cache is now invalidated.

It is possible to freeze both banks of the cache if there are two sections of code or data that need to be executed rapidly. Since the cache locking feature is implemented as bank selectable, when one bank is frozen (reducing the cache to a direct mapped cache), the cache can still be mapped to access the entire address space.

6

SECTION 7
EXCEPTIONS

This section details the MC88110 exceptions and the steps which are taken to recognize, process, and handle exceptions. This section also describes the exception model implemented in the MC88110 and the execution latencies associated with exceptions and interrupts.

7.1 EXCEPTION OVERVIEW

Exceptions are conditions that cause the processor to suspend execution of the current instruction stream and perform exception processing. Exception processing provides an efficient context switch so that system software can handle the exception condition while maintaining the integrity of the hardware and other software. Exception conditions include the following:

- External interrupts, signaled by the $\overline{\text{INT}}$ or $\overline{\text{NMI}}$ signals
- Memory access conditions, such as page faults and bus errors
- Internally recognized errors, such as divide-by-zero and arithmetic overflow
- Trap instructions
- Illegal instructions and privilege violations

When an exception is recognized by the processor, the execution context is saved into exception-time registers, and the machine is placed in supervisor mode. Control is then passed to a designated exception handler routine. The exception handler routine is user provided software which processes the condition that caused the exception. The handler routine performs specific functions (e.g., fixing internal errors, aborting operations, or servicing interrupts) based on the type of exception that has occurred. Once the exception handler routine has finished, it returns control to the normal instruction stream.

The MC88110 implements a precise exception model. This means that the precise address of the faulting instruction is provided to the exception handler and that neither the faulting instruction nor any instructions logically following it in the code stream will appear to have been issued. Because of the precise exception model, it is not necessary for the internal pipeline states of the processor to be visible to the software handlers.

7.2 THE EXCEPTION MODEL

The following paragraphs give detailed descriptions of the history buffer, the vector table, and the registers that are used in the exception model. The history buffer can be visualized as a first-in-first-out (FIFO) queue which maintains the instruction issue order and information about the machine state at the time each instruction was issued. The vector table consists of 512 exception vectors in a 4K-byte memory page which is pointed to by the vector base address in the vector base register (VBR).

7.2.1 The History Buffer

Instructions in the MC88110 execute in parallel and possibly out of order internally. This out-of-order execution makes it possible for an instruction to cause an exception after logically subsequent instructions have already been issued and completed.

To maintain a precise exception model, the MC88110 ensures that this out-of-order execution is not apparent to the exception handler. When an exception is recognized, all of the instructions issued before the excepting instruction finish execution, and then the MC88110 restores the machine state to a point where neither the excepting instruction nor any instructions logically following it in the instruction stream appear to have executed. This machine state is the same state which would be present if all the instructions had been executed in the order that they were fetched from memory; thus, the exception handler knows exactly which instructions have completed execution, which have not yet been issued, and which instruction caused the fault.

The information about the machine state and the order in which instructions were issued is maintained in a history buffer. This buffer can be visualized as a FIFO queue which records the relevant machine state at the time each instruction is issued. At the time of issue, each instruction and the associated machine state is placed at the tail of the queue. The instructions move through the FIFO until they reach the head of the queue. An instruction reaches the head when all of the instructions in front of it have finished execution; however, since instructions can be executed out of order, it is possible for an instruction to have finished execution, but still be in the middle of the queue. An instruction is removed from the queue when it reaches the head and has finished execution. The information placed in the history buffer is sourced through the same output ports from the register file as stores. This prevents writes to the history buffer from contending with other traffic on the source and destination busses.

Figure 7-1 shows an example of the history buffer where an **fmul.ddd** instruction was the first instruction issued by the MC88110, and a **subu** instruction was the last instruction issued. If the **fmul.ddd** has stalled and has not yet finished execution, then even if the **and** has finished execution, it cannot be retired from the buffer until the **fmul.ddd** has finished.

HEAD OF HISTORY BUFFER → fmul.ddd r8, r6, r4

and r15, r16, r17

or r2, r11, r13

addu r3, r11, r12

TAIL OF HISTORY BUFFER → subu r10, r11, r10
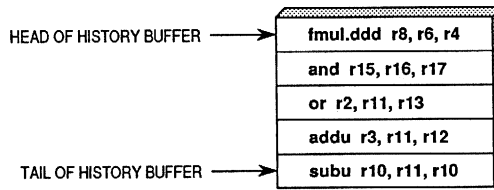
Figure 7-1. History Buffer Example

7.2.2 Exception Vectors and Vector Base Register (VBR)

The MC88110 uses exception vectors to transfer control to user exception handler routines during exception processing. The MC88110 maintains a vector table consisting of 512 exception vectors in a 4K-byte memory page which is pointed to by the vector table base address in the VBR.

The VBR is loaded by supervisor software, normally as part of the system initialization procedure. The VBR is control register 7 (**cr7**). The VBR has read/write access in supervisor mode and may be modified by supervisor software to dynamically specify which page in memory contains the vector table; however, it is recommended that this register only be modified when exceptions are disabled (i.e., EFRZ, SFD1, and IND in the processor status register (PSR) are set) so that an exception does not occur while the contents of the VBR are changing. The lower twelve bits of the VBR are unused, and the VBR is initialized to zero on reset.

Each exception has a 9-bit exception number which either is generated by hardware when that exception occurs or is specified as a 9-bit field in trap instructions. This number is used to form the address of the corresponding exception vector in the vector table. Exception vector addresses are formed by concatenating the 20 most significant bits of the VBR with the 9-bit exception vector number and then appending three zeros to this value. Figure 7-2 shows how exception vector addresses are formed.
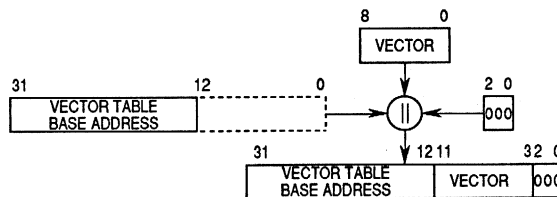
Figure 7-2. Exception Vector Address Formation

The lower 128 vectors (numbers 0–127) in the vector table are reserved for hardware and supervisor use. They are not accessible from user trap instructions. Attempting to specify any of these vectors from a trap instruction from user mode will cause a privilege

violation exception to occur. The upper 384 vectors (numbers 128–511) are allocated for software traps. Table 7-1 lists exception conditions and their respective exception numbers.

Table 7-1. Exception Vectors

Number	Vector Base Address Offset	Exception
0	$00	Reset
1	$08	Maskable Interrupt
2	$10	Instruction Access
3	$18	Data Access
4	$20	Misaligned Address
5	$28	Unimplemented Opcode
6	$30	Privilege Violation
7	$38	Bounds Check Violation
8	$40	Integer Divide-by-Zero
9	$48	Integer Overflow
10	$50	Unrecoverable Error
11	$58	Nonmaskable Interrupt
12	$60	Data MMU Read Miss
13	$68	Data MMU Write Miss
14	$70	Instruction MMU ATC Miss
15-113	—	Reserved
114	$390	SFU1—Floating-Point Exception
115	$398	Reserved
116	$3A0	SFU2—Graphics Exception
117	$3A8	Reserved
118	$3B0	SFU3—Unimplemented Opcode
119	$3B8	Reserved
120	$3C0	SFU4—Unimplemented Opcode
121	$3C8	Reserved
122	$3D0	SFU5—Unimplemented Opcode
123	$3D8	Reserved
124	$3E0	SFU6—Unimplemented Opcode
125	$3E8	Reserved
126	$3F0	SFU7—Unimplemented Opcode
127	$3F8	Reserved
128-511	—	Reserved—User Trap Vectors

7

Each exception vector contains two instructions: typically, one instruction is a branch instruction whose target address is the exception handler, and the other instruction is the first instruction of the corresponding exception handler routine.

7.3 EXCEPTION RECOGNITION, PROCESSING, HANDLING AND RETURN FROM EXCEPTIONS

The MC88110 treats all exceptions the same except the reset and error exceptions. When an exception occurs, the following four interrelated phases are performed:

1. Exception recognition—the processor restores the machine state associated with the faulting instruction.

2. Exception processing—the processor saves the execution context in exception-time registers, and changes program flow to the exception handler routine.

3. Exception handling—the exception handling software corrects the exception condition or performs the function initiated by a trap instruction.

4. Return from exception—the processor restores the execution context which was in effect before the exception occurred and resumes normal execution of program instructions.

The following paragraphs describe these four phases in more detail.

7.3.1 Exception Recognition

This section describes in detail when and how the MC88110 internally recognizes exceptions and interrupts, including the operation and states of the history buffer. Also discussed are the priorities associated with exceptions and interrupts.

7.3.1.1 INTERNAL OR BUS GENERATED EXCEPTIONS. When an instruction generates an exception during execution, the history buffer entry containing the associated instruction is marked as having a pending exception. The exception is not recognized until that entry reaches the head of the history queue. At this point, any instructions that are currently pending in execution unit pipelines or in data unit buffers that have not yet written back are discarded. In other words, all instructions which were issued after the excepting instruction are discarded. Any load instructions which have been granted access to the cache or bus are allowed to complete, but the write-back of their results is waived.

Figure 7-3 shows an example of instructions in the history buffer. In Figure 7-3(a), the **faddu** instruction has caused an overflow condition. Figure 7-3(b) shows that the exception will be recognized when the **faddu** instruction has reached the head of the queue after the **mul** instruction has finished.

HEAD OF HISTORY BUFFER → mul r15, r16, r17
or r2, r11, r13
FAULTING INSTRUCTION → faddu x3, x11, x12
(EXAMPLE: FLOATING-POINT OVERFLOW)
•
•
•
TAIL OF HISTORY BUFFER → subu r10, r11, r10

(a) Exception Occurs

EXCEPTION RECOGNIZED → faddu x3, x11, x12
•
•
•
subu r10, r11, r10
cmp r5, r10, r0
TAIL OF HISTORY BUFFER → and r5, r5, r6
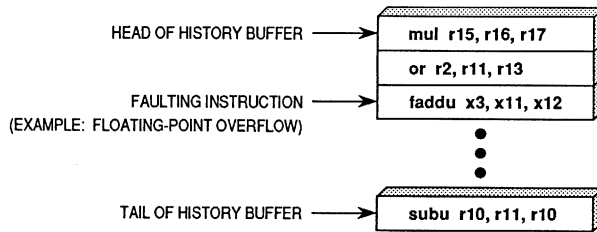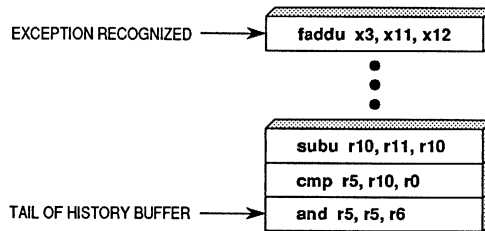
(b) Exception Recognized

Figure 7-3. Exception Recognition in the History Buffer

Next, the processor back tracks through the instructions and machine states stored in the history buffer, and the current machine state is restored at a rate of two instructions per clock cycle to its value at the time the excepting instruction was issued. The machine states stored in the history buffer include the contents of any destination registers, so as the processor restores the machine state, the destination registers for the instructions are updated to their original values.

Memory never has to be restored during the machine state restoration process. Store instructions are placed in the history buffer when they are issued to the load/store unit but are not allowed to update memory until they reach the head of the queue. This means that stores always complete in program order and never modify memory until all previous instructions have completed.

7.3.1.2 EXTERNALLY GENERATED INTERRUPTS. An externally generated interrupt will be taken when an instruction marked as causing an exception reaches the head of the history buffer. To make sure that the interrupt will be handled as soon as possible, all instructions issued after the interrupt is detected are issued as unimplemented instructions. While a load or store that has already accessed the bus will be allowed to complete, any additional instructions which write-back are marked as causing an exception. As soon as one of these instructions, or the first unimplemented instruction reaches the head of the history buffer, then the interrupt vector will be taken.

7.3.1.3 PRIORITIES. There is no priority associated with internally generated exceptions. If two or more instructions in execution at the same time cause exceptions, the exception caused by the instruction that was issued first will be recognized when it reaches the head of the queue. When the machine state is recovered, all other instructions in the queue will be discarded. After the exception has finished, these instructions will be reissued, and the next one in the stream causing an exception will be recognized when it reaches the head of the queue.

Externally generated interrupts have priority over all internally generated exceptions, except for data access exceptions. Once an interrupt has been detected, the next instruction to reach the head of the queue marked as having an exception will cause the interrupt vector to be taken, unless the exception is a data access exception.

7.3.2 Exception Processing

Once an exception has been recognized, the following actions are taken by the processor (see Figure 7-4):

1. The machine state is recovered.

2. The logical address of the excepting instruction is saved in the exception-time executing instruction pointer register (EXIP) along with a bit indicating whether the instruction was in the delay slot of a branch.

3. If the excepting instruction was in the delay slot of a branch, then the address of the next instruction to execute is saved in the exception-time next instruction pointer register (ENIP).

4. The PSR is saved in the exception-time processor status register (EPSR).

5. The IND bit in the PSR is set in order to disable maskable external interrupts (see **7.5.1 Interrupts**)

6. The EFRZ bit in the PSR is set in order to freeze the exception-time registers. If another exception occurs while this bit is set, it will be directed to the error exception vector.

7. The Mode bit in the PSR is set in order to place the machine into supervisor mode.

8. The exception vector address is computed using the VBR and the vector number as shown in Figure 7-2

9. Execution is resumed with the two instructions found at the exception vector address, causing program flow to be diverted to the exception handler.

7

EXCEPTION IS
RECOGNIZED

MACHINE STATE
IS RESTORED

LOGICAL ADDRESS OF
FAULTING INSTRUCTION
PLACED IN EXIP

WAS THE INSTRUCTION IN
A DELAY SLOT OF A BRANCH?

? — YES → SAVE THE ADDRESS OF
THE NEXT INSTRUCTION
TO EXECUTE IN THE ENIP

NO

PSR SAVED IS IN THE EPSR

THE IND BIT IN THE PSR IS
SET TO DISABLE INTERRUPTS
AND EXCEPTIONS

PROCESSOR IS PLACED IN
SUPERVISOR MODE

THE VECTOR ADDRESS
IS COMPUTED

EXECUTION IS RESUMED
WITH INSTRUCTIONS AT
THE VECTOR ADDRESS

Figure 7-4. Exception Processing Flow Chart

The EPSR has exactly the same format as the PSR (see **Section 2 Programming Model**). The format for the EXIP and ENIP are shown in Figures 7-5 and 7-6.

31	2	1	0
EXECUTING INSTRUCTION POINTER			D

LEGEND:
▒ UNDEFINED-RESERVED FOR FUTURE USE
D—DELAY SLOT BIT
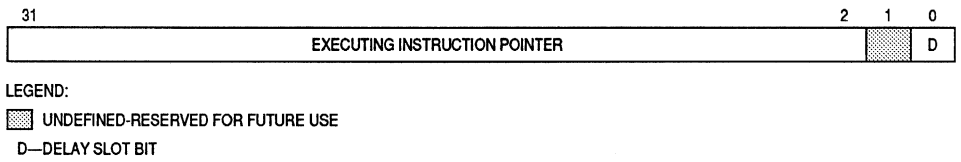
Figure 7-5. Exception-Time Executing Instruction Pointer (EXIP)

```
31                                                              2   1   0
┌──────────────────────────────────────────────────────┬──────────┐
│                NEXT INSTRUCTION POINTER                │░░░░░░░░░░│
└──────────────────────────────────────────────────────┴──────────┘
```

▒ UNDEFINED-RESERVED FOR FUTURE USE
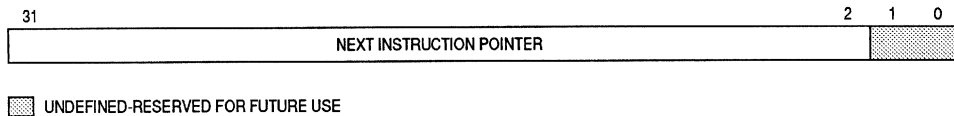
Figure 7-6. Exception Time Next Instruction Pointer (ENIP)

7.3.3 Exception Handling

Typically, exception handlers first save the state of the processor, including the exception-time registers (EPSR, EXIP, and ENIP), the floating-point status register (FPSR), the floating-point control register (FPCR), as well as any of the general registers or extended registers that may be used by the handler. Once the machine state has been stored in memory, exception handlers may reenable exceptions and interrupts by clearing the EFRZ and IND bits in the PSR. Doing so allows another exception (a nested exception) to occur while the first exception is being handled. If exceptions and interrupts are enabled before the state is saved and another interrupt occurs, the machine state information would be lost when the second exception writes to the exception processing registers.

If the required exception handling is simple and the system can tolerate execution of the handler with interrupts and exceptions disabled, then the handler can avoid the overhead of saving and restoring the processor state to memory. In this case, the handler routine must guarantee that it will not generate any additional exceptions. If any exceptions are generated, they will be referred to the error vector.

To simplify and speed up handling of exceptions, five control registers (**cr16–cr20**), which are accessible only in supervisor mode, are provided. These registers can be used to store the supervisor stack pointer or other operating system specific data. At exception time, general registers can be exchanged (using the exchange control register (**xcr**) instruction) with these registers to minimize the amount of memory traffic needed by fast trap handlers. These registers may also be used as scratch storage by fast exception handlers to avoid saving general registers to memory.

7

7.3.4 Return from Exceptions

If the machine state was saved at the start of the exception handler, then it must be restored when the handler is finished. A return from exception (**rte**) instruction should be the last instruction in the handler routine. The **rte** instruction is a privileged instruction and is the mechanism provided by MC88110 for exiting exception handling routines. When the **rte** instruction is executed, the following sequence of events is performed (see Figure 7-7):

1. The machine is serialized to guarantee that all exception handler instructions complete before control is returned to the program. Serialization means that instruction issue is halted until all currently executing instructions have finished, at which point all of the pipeline stages are empty and all outstanding memory transactions have been completed.

2. The PSR is restored from the EPSR.

3. The machine is placed in the mode (user or supervisor) it was in at the time of the exception (as indicated by the EPSR).

4. The instruction at the address indicated by the EXIP is fetched.

5. If the excepting instruction was in the delay slot of a branch, as indicated by the D bit in the EXIP (D=1), then the instruction at the address in the ENIP is fetched and will be executed as the second instruction. Software changes to the ENIP have no effect if **rte** is executed with D=0.

6. Normal execution resumes with the instruction at the address in the EXIP. If the D bit in the EXIP was set, then the instruction fetched in step 5 is the next instruction which executes; otherwise, execution continues sequentially.

Since the address stored in the EXIP by the processor was the address of the faulting instruction, the handler should determine if execution is to resume with that instruction or the next instruction. If the handler does not want the faulting instruction to be reissued, then it should increment the value in the EXIP so that it points to the next instruction in the stream.
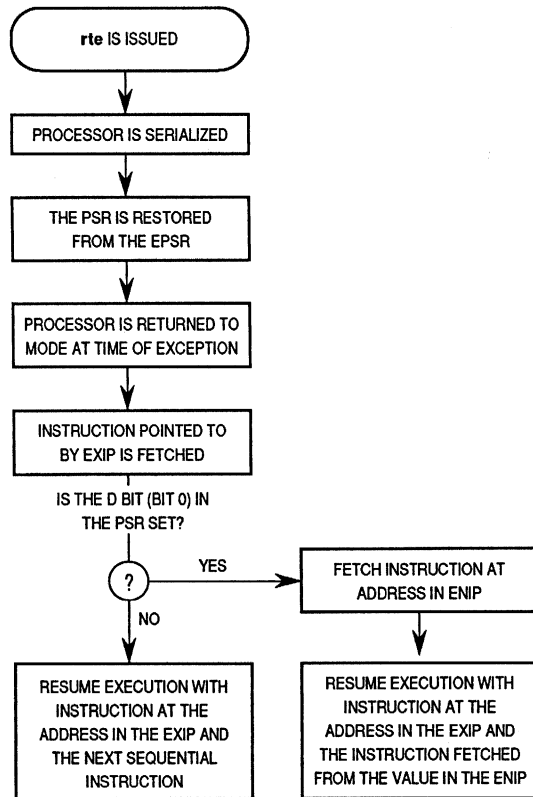
Figure 7-7. Return From Exceptions Flow Chart

7.4 EXCEPTION TIMING

The following paragraphs describe the latencies associated with exceptions and interrupts.

7.4.1 Latency for Internal or Bus Generated Exceptions

Figure 7-8 illustrates the latencies associated with exceptions other than interrupts.

A—At time A, an instruction which is destined to generate an exception is issued.

B—At time B, the instruction has reached the head of the history buffer implying that all instructions preceding it in the code stream have finished execution without generating any exceptions.

C—By time C, the instruction has caused an exception while being executed, and the exception has been recognized. At this time, exception processing begins. Note that if the instruction had not generated an exception by this time, it would have been retired from the buffer.

D—By time D, the state of the machine has been restored to the machine state at the time the excepting instruction was issued.

E—At time E, the PSR and instruction pointer of the currently executing process have been saved, and control has been transferred to the exception handler routine.
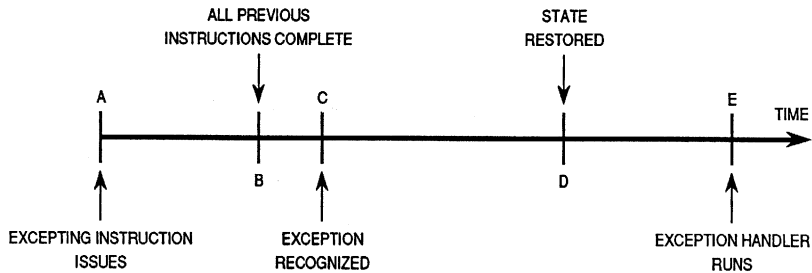
NOTE: At time A, an instruction which is destined to generate an exception is issued

Figure 7-8. Exception Latency Time Line

At time A, the excepting instruction is issued and begins execution. The faulting condition occurs sometime during the interval from A to C.

At time B, the excepting instruction has reached the head of the history buffer. The interval B–C is the time required for the machine to complete execution of the excepting instruction, and any load which is in progress on the external bus. Frequently, B–C will be zero since most instructions finish execution before reaching the head of the queue.

At time C, the exception is recognized, and during C–D the machine state is restored to the machine state at the time the excepting instruction was issued. The length of C–D depends on the number of instructions issued after the excepting instruction was issued. Since a maximum of 11 additional instructions could have been issued and the machine state is restored at a rate of 2 instructions per clock, the interval C–D can be a maximum of 6 clocks.

At time D, the machine state has been completely restored. During the interval D–E the MC88110 performs all of the actions associated with transferring control to the exception handler (exception processing). The interval D–E requires 2 clocks plus the time required to fetch the target handler instructions (3 clocks for ideal memory).

7.4.2 Latency for Externally Generated Interrupts

The interrupt latency differs from the latency for other exceptions only in the time up to when the exception is recognized. The latency after the exception is recognized (shown as the interval from C to E in **7.4.1 Latency for Internal or Bus Generated Exceptions**) is the same for all exceptions and external interrupts.

As discussed in **7.3.1.2 Externally Generated Interrupts**, the interrupts are recognized when the first instruction which causes an exception reaches the head of the history buffer after the interrupt signal is asserted. Therefore, the latency can vary depending on what instructions were issued just prior to the interrupt.

7.5 TYPES OF EXCEPTIONS

The following paragraphs describe the types and causes of interrupts and exceptions in the MC88110.

7.5.1 Interrupts

The MC88110 provides two external interrupt signals—one maskable and the other nonmaskable. Each interrupt has a unique vector in the exception vector table.

7.5.1.1 MASKABLE INTERRUPT (INT). INT is level sensitive (active low), not edge triggered. The interrupting device should keep the INT signal asserted until it receives explicit recognition. This recognition is normally generated by the interrupt handler. INT is software maskable by the IND bit in the PSR. Upon recognition of any exception, IND is automatically set by hardware to disable maskable interrupts. The maskable interrupt is of lower priority than the nonmaskable interrupt, but higher than internal exceptions.

7.5.1.2 NON-MASKABLE INTERRUPT (NMI). The nonmaskable interrupt is not masked by the IND interrupt disable bit in the PSR and is therefore useful as a high priority system interrupt. The nonmaskable interrupt can, however, be masked by the EFRZ bit in the PSR. Since the EFRZ bit is set during exception recognition, all exception handlers are guaranteed to have a chance to save the previous machine state before a nonmaskable interrupt is taken. A nonmaskable interrupt can be taken once the EFRZ bit is cleared. This assures recoverability from a nonmaskable interrupt which is nested within another exception or interrupt.

NMI is transition sensitive (falling edge) and should be held asserted until it is acknowledged by the interrupt handler. Once it is recognized by the MC88110, NMI must transition to a negated level and be reasserted before another nonmaskable interrupt will be taken. This requirement is illustrated in Figure 7-9.

REQUIRED TRANSITION

NMI

FIRST NMI VECTOR TAKEN   SECOND NMI VECTOR TAKEN
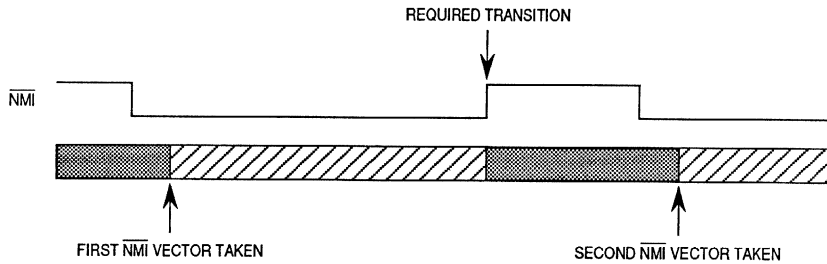
Figure 7-9. NMI Signal Timing

7.5.2 Instruction Unit Exceptions

There are five types of instruction unit exceptions: misaligned access exceptions, unimplemented opcode exceptions, privilege violation exceptions, trap instruction exceptions, and integer overflow exceptions. Recall that the value stored in the EXIP at the time any of these exceptions are processed is the address of the instruction which caused the exception.

7.5.2.1 MISALIGNED ACCESS EXCEPTION (VECTOR OFFSET $20). A misaligned access exception occurs when a load, store, or exchange is attempted to a memory address that is not consistent with the size of the access. For example, this exception occurs when a half-word access is attempted to an odd byte address. This exception also occurs when a double-word access is attempted to an address that is an odd-word boundary.

A misaligned access is detected before the memory access is dispatched to memory. The exception handler can either emulate the memory access in software or discard the instruction.

The misaligned access exception can be masked by setting the MXM bit in the PSR. If this exception is masked and a misaligned access is attempted, the processor performs the access to the next lower properly aligned boundary (e.g., a half-word read operation attempted to address $401 returns the half word at location $400).

7.5.2.2 UNIMPLEMENTED OPCODE EXCEPTION (VECTOR OFFSET $28). This exception occurs when an instruction with an unimplemented opcode is loaded into the instruction pipeline. The exception handler can fetch, decode, and process the instruction, thereby emulating unimplemented opcodes in software. If instruction emulation is not needed, the handler can discard the instruction or perform other appropriate processing. Unimplemented special function unit 1 (SFU1) or special function unit 2 (SFU2) instructions do not cause this exception but generate the corresponding SFU1 or SFU2 exception.

7.5.2.3 PRIVILEGE VIOLATION EXCEPTION (VECTOR OFFSET $30). A
privilege violation occurs when software attempts to perform a privileged operation while in user mode. Privilege violations are caused by the following three conditions:

1. Accessing a control register other than the FPCR or FPSR while in user mode.
2. Using the **.usr** option while in user mode.
3. Specifying exception vectors 0-127 in a trap instruction while in user mode.

When a privilege violation occurs, the privileged operation is not performed.

7.5.2.4 TRAP INSTRUCTION EXCEPTIONS (VECTOR OFFSET $400-$7F8).
Trap instructions are MC88110 instructions that explicitly cause an exception.

The MC88110 instruction set includes four trap instructions: **tcnd, tb1, tb0,** and **tbnd**. The **tcnd, tb1,** and **tb0** instructions can initiate any exception handler by specifying the appropriate vector number (see Table 7-1); however, in user mode, a trap to vectors 0–127 will cause a privilege violation whether or not the trap condition is met. The **tcnd, tb1,** and **tb0** instructions cause the MC88110 to serialize before the instruction is issued.

A bounds-check violation exception (vector offset $38) occurs when **tbnd** detects a value that is outside of the bounds specified by the instruction.

The value stored in the EXIP at the beginning of the exception handler is the address of the trap instruction. The handler should change this value so that it points to the next instruction after the trap in the program; otherwise, the trap instruction will be reissued when normal processing begins. For the EXIP to point to the next instruction after the trap, the value of the EXIP should be the original value of the EXIP at the beginning of the handler plus four.

7.5.2.5 INTEGER OVERFLOW EXCEPTION (VECTOR OFFSET $48). The
integer overflow exception occurs when the result of a signed integer arithmetic instruction cannot be represented as a 32-bit signed number. The EXIP points to the instruction that caused the exception. The destination register and carry bit are unchanged by an instruction that causes an integer overflow exception.

7.5.3 Memory Access Exceptions

Memory access exceptions occur when a data memory access or an instruction prefetch fails to complete normally. The following paragraphs describe the situations that cause exceptions for instruction accesses and for data accesses.

7.5.3.1 INSTRUCTION ACCESS EXCEPTION (VECTOR OFFSET $10). An
instruction access exception can occur if an instruction fetch is terminated with a bus error, or a hardware table search is terminated with a fault. A fault occurs during a table search if a privileged descriptor is accessed in user mode (supervisor privilege violation) or an invalid segment or page descriptor is encountered. A bus error on any of the accesses during the table search will also terminate the table search and cause an instruction access exception.

When an instruction access exception occurs, the logical address of the original instruction access is placed in the instruction access logical address register (ILAR). If the exception was caused by a bus error on either the actual instruction access or an access during a table search, then the physical address where the bus error occurred is placed in the instruction access physical address register (IPAR). If the exception was caused by either a supervisor privilege violation or an invalid segment or page descriptor, then the physical address of the descriptor is placed in the IPAR. For more information on the ILAR and IPAR, see **Section 8 Memory Management Units**.

Both the ILAR and IPAR are frozen when the EFRZ bit in the PSR is set during exception processing. The exception handling software must clear the EFRZ bit to allow any future updates to these registers.

The MC88110 sets individual bits in the instruction access status register (ISR) to indicate the cause of the instruction access exception. This register is frozen when the EFRZ bit in the PSR is set. The exception handler must clear the EFRZ and then clear the ISR. The format of the ISR is shown in the following illustration:

31	22	21	20	19	18	17	12	11	10	9	8	1	0
		TBE	SI	PI	SP			PH	BH	S/U			BE

▓ UNDEFINED-RESERVED FOR FUTURE USE
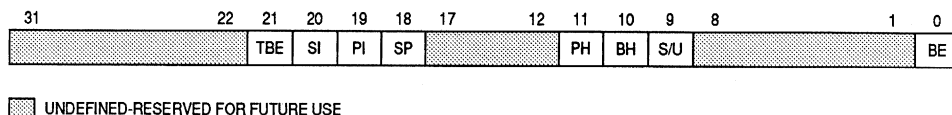
Figure 7-10. Instruction Access Status Register (ISR)

TBE—Table Search Bus Error
  Indicates that a bus error was encountered during a table search.

SI—Segment Descriptor Invalid
  Indicates that an invalid segment descriptor was encountered during a table search.

PI—Page Descriptor Invalid
  Indicates that an invalid page descriptor was encountered during a table search.

SP—Supervisor Privilege Violation
  Indicates that a hardware table search resulted in a supervisor privilege violation. The physical address of the faulting descriptor is located in the IPAR. The logical address of the original access is located in the ILAR.

PH—Page Address Translation Cache (PATC) Hit
    1 = Probed address resulted in a PATC hit.
    0 = Probed address was not found in the PATC.

BH—Block Address Translation Cache (BATC) Hit
  1 = Probed address resulted in a BATC hit.
  0 = Probed address was not found in the BATC.

S/U—Supervisor/User Status
  Indicates the supervisor/user status of the instruction access in error.

BE—Bus Error
  Indicates that a bus error occurred.

The exception handler must determine the cause of the exception and then optionally retry the instruction fetch. For example, if the exception was caused by a page fault, the requested memory page must be read in from memory by the exception handler. Upon exiting, the EXIP will already point to the address of the faulting instruction so the instruction can be reexecuted. If the exception was caused by a privilege violation or a nonexistent memory fault, the exception handler may opt to abort the instruction fetch. In this case, the handler should change the address in the EXIP to point to another instruction.

7.5.3.2 DATA ACCESS EXCEPTION (VECTOR OFFSET $18). A data access exception occurs in response to a bus error during a data transaction or in response to one of several MMU conditions. A data transaction which misses in the data cache can initiate a data copyback or a line read operation. A bus error (signaled by assertion of the $\overline{\text{TEA}}$ signal) on either the copyback or the line read will cause a data access exception.

If a write data access is attempted on a page or block which is designated as write-protected (the write protect (WP) bit is set in the descriptor) then a data access exception is generated. A data access exception will occur if a supervisor violation, an invalid segment or page descriptor, or a bus error is detected during a hardware table search. A breakpoint exception will also cause a data access exception. This is described further in **Section 8 Memory Management Units**.

When a data access exception is generated by a bus error, the faulting physical address is placed in the data access physical address register (DPAR). When an exception is caused by a privilege violation or an invalid descriptor during a table search, the physical address of the descriptor is placed in the DPAR. This register remains undefined for write and for breakpoint exceptions.

For privilege and invalid descriptor violations during table searches, the logical address of the original access is placed in the data access logical address register (DLAR). This register is also updated with the address of the faulting access in the cases of breakpoint and write exceptions. The DLAR remains undefined for exceptions caused by bus errors during a copyback or a snoop copyback.

7

Both the DLAR and DPAR are frozen when the EFRZ bit in the PSR is set during exception processing. The exception handling software must clear the EFRZ bit to allow any updates to these registers. For more information on the DPAR and the DLAR, see **Section 8 Memory Management Units**.

The MC88110 sets individual bits in the data access status register (DSR) to indicate the cause of the instruction access exception. This register is frozen when the EFRZ bit in the PSR is set. The exception handler must clear the EFRZ and then clear the DSR. The format of the DSR is shown in the following Figure 7-11.

31	22	21	20	19	18	17	16	15		12	11	10	9	8	7	3	2	1	0
		TBE	SI	PI	SP	WE	BPE				PH	BH	S/U	R/W			CP	WA	BE

▨ UNDEFINED-RESERVED FOR FUTURE USE
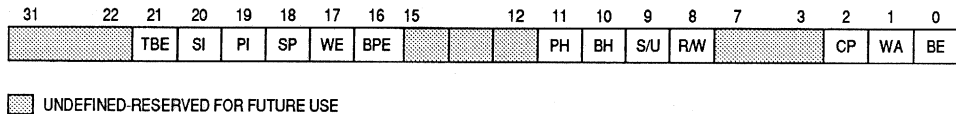
Figure 7-11. Data Access Status Register (DSR)

TBE—Probe Table Search Bus Error
  Indicates that a bus error was encountered during a table search.

SI—Segment Descriptor Invalid
  Indicates that an invalid segment descriptor was encountered during a hardware table search.

PI—Page Descriptor Invalid
  Indicates that an invalid page descriptor was encountered during a hardware table search.

SP—Supervisor Privilege Violation
  Indicates that a hardware table search resulted in a supervisor privilege violation. The physical address of the faulting descriptor is located in the DPAR. The logical address of the original access is located in the DLAR.

WE—Write Exception
  Indicates that the data access resulted in a write exception.

BPE—Breakpoint Exception
  Indicates that a breakpoint exception occurred.

PH—Page Address Translation Cache (PATC) Hit
    1 = Probed address resulted in a PATC hit.
    0 = Probed address was not found in the PATC.

BH—Block Address Translation Cache (BATC) Hit

    1 = Probed address resulted in a BATC hit.
    0 = Probed address was not found in the BATC.

S/U—Supervisor/User Status

    Indicates the supervisor/user status of the data access in error.

R/W—Read/Write Status

    Indicates the read/write status of the data access in error.

CP—Copyback Error

    Indicates that an error occurred during a cache copyback initiated by the normal replacement of a dirty cache entry or that a cache flush was unsuccessful.

WA—Write-Allocate Bus Error

    Indicates that a bus error occurred during the line read operation of a write cache miss implementing the write allocation policy.

BE—Bus Error

    Indicates that a bus error occurred during a data access.

Once the cause of the data access fault is determined, the exception handler may correct the fault condition or may ignore the memory transaction. For example, if the exception was caused by a page fault, the requested memory page must be read in from memory by the exception handler. Upon exiting, the EXIP is pointing to the address of the faulting instruction so the instruction will be reexecuted. If the exception was caused by a privilege violation, a write protection, or a nonexistent memory fault, the exception handler may opt to ignore the transaction. In this case, the handler should change the address in the EXIP to point to another instruction (most likely the instruction after the faulting one).

7.5.4 Floating-Point Unit Exceptions

There are 8 types of floating-point unit exceptions. Each of these exceptions cause the SFU1 vector to be taken. Table 7-2 depicts a summary of all the floating-point instructions of the MC88110 and the exceptions that each of these instructions can cause. The exceptions are itemized by setting the corresponding bit in the floating-point exception cause register (FPECR). There are no exceptions referenced for the FPRV bit because this bit is only set when there is an attempt to access a privileged (implemented or unimplemented) floating-point register from user mode and does not directly correspond to a particular instruction. For more information on the floating-point exceptions and how they conform to the ANSI/ IEEE standard, refer to **Section 4 Floating-Point Implementation**.

Table 7-2. Exceptions Caused by Floating-Point Instructions

Instructions	Exceptions						
	FIOV	FUNIMP	FROP	FDVZ	FUNF	FOVF	FINX
fmul		SFU1 Disabled	NaN, Invalid, Denorm, or Unnorm		Underflow	Overflow	Inexact
fadd		SFU1 Disabled	NaN, Invalid, Denorm, or Unnorm		Underflow	Overflow	Inexact
fsub		SFU1 Disabled	NaN, Invalid, Denorm, or Unnorm		Underflow	Overflow	Inexact
fcvt		SFU1 Disabled	NaN, Invalid, Denorm, or Unnorm		Underflow	Overflow	Inexact
fcmp		SFU1 Disabled	NaN, Invalid, Denorm, or Unnorm				
fcmpu		SFU1 Disabled	NaN, Invalid, Denorm, or Unnorm				
flt		SFU1 Disabled					Inexact
int	$rS2 < -2^{31}$, $rS2 > 2^{31} - 1$	SFU1 Disabled	NaN, Invalid, Denorm, or Unnorm				Inexact
nint	$rS2 < -2^{31}$, $rS2 > 2^{31} - 1$	SFU1 Disabled	NaN, Invalid, Denorm, or Unnorm				Inexact
trnc	$rS2 < -2^{31}$, $rS2 > 2^{31} - 1$	SFU1 Disabled	NaN, Invalid, Denorm, or Unnorm				Inexact
fdiv		SFU1 Disabled	NaN, Invalid, Denorm, or Unnorm	$rS2 = 0$	Underflow	Overflow	Inexact
fsqrt		Always					
mov		SFU1 Disabled					

7.5.4.1 FLOATING-POINT UNIMPLEMENTED. This exception occurs when one of the following situations occurs:

1. If a floating-point operation is attempted when SFU1 is disabled
2. If there is an attempt to execute an unimplemented floating-point opcode (including the **fsqrt** instruction)
3. If there is an attempt from supervisor mode to access an unimplemented floating-point control register
4. If there is an attempt to access a double-precision floating-point number which is aligned on an odd-numbered register pair.

When this exception occurs, the FUNIMP bit (bit 6) of the FPECR is set by hardware. The unimplemented instruction has no effect on the register scoreboard. When this exception occurs, all other bits in the FPECR are undefined; therefore this bit should be checked first.

7.5.4.2 FLOATING-POINT PRIVILEGE VIOLATION. This exception occurs when an attempt is made to access any of the privileged floating-point control registers (**fcr0-fcr61**) from user mode. The instructions which can cause this exception are **fldcr**, **fxcr** and **fstcr**. This exception causes the FPRV bit (bit 5) of the FPECR to be set.

7.5.4.3 FLOATING-POINT TO INTEGER CONVERSION OVERFLOW. This exception occurs when a source operand of a floating point to integer conversion instruction is too large to be represented as a signed 32-bit integer. The instructions which can invoke this exception are **int**, **nint**, and **trnc**. This exception causes the FIOV bit (bit 7) of the FPECR to be set.

7.5.4.4 FLOATING-POINT RESERVED OPERAND. This exception occurs when either of the source operands of an instruction is a reserved operand, or the operation being performed on the given operand is invalid according to the IEEE 754 standard. This exception causes the FROP bit (bit 4) in the FPECR to be set.

7.5.4.5 FLOATING-POINT OVERFLOW. This exception occurs when the rounded result of the operation is too large to be represented as a finite number in the destination format (single-, double-, or double-extended). This exception causes the FOVF bit (bit 1) as well as the FINX bit (bit 0) of the FPECR to be set.

7.5.4.6 FLOATING-POINT UNDERFLOW. This exception occurs when the rounded result of the operation is too small to be represented as a finite normalized number in the destination format (single-, double-, or double-extended). This exception causes the FUNF bit (bit 2) in the FPECR to be set.

7.5.4.7 FLOATING-POINT DIVIDE-BY-ZERO. This exception occurs when the denominator (**rS2**) operand of an **fdiv** instruction is a zero and the numerator is a nonzero finite normalized number. This exception causes the FDVZ bit (bit 3) in the FPECR to be set.

7

7.5.4.8 FLOATING-POINT INEXACT. This exception occurs when the rounding of a result causes a loss of accuracy or when significance is lost by the occurrence of an overflow condition. This exception causes the FINX bit (bit 0) in the FPECR to be set if the EFINX bit in the FPCR is set. Otherwise, the hardware sets the AFINX bit in the FPSR and does not take an exception.

7.5.5 Graphics Unit Exceptions (Vector Offset $3A0)

There are two types of graphics unit exceptions. Both of these exceptions cause the same SFU2 exception vector to be taken. The causes of these exceptions are described below.

7.5.5.1 SFU2 DISABLED. This exception occurs when SFU2 is disabled and a graphics instruction attempts to issue. The unit is disabled when SFD2 (bit 4) bit of the PSR is set.

7.5.5.2 SFU2 UNIMPLEMENTED. This exception occurs when an attempt is made to execute an unimplemented instruction in the SFU2 opcode class. This exception will also occur if an odd register is specified for a double word operand.

7.5.6 Error Exception

The error exception provides the means to terminate processing when catastrophic situations are encountered. The error exception occurs if another exception occurs when the EFRZ bit in the PSR is set. The EFRZ bit is only set during exception processing. In other words, the error exception will be taken if the MC88110 is processing one exception whose handler has not cleared the EFRZ bit and a second exception occurs. The error exception will also be taken if the MC88110 encounters a fault while fetching an exception handler.

The error handler routine can initialize the processor and resume execution at address $0 or signal external hardware to perform a reset operation. If the error exception vector cannot be fetched successfully (e.g., because of a memory error on the vector table page), then the error exception cannot be taken. This situation causes the MC88110 to loop on fetching the error exception vector. This loop can only be exited by a processor reset.

7.5.7 Reset

Processor reset is a special exception case that occurs when the $\overline{\text{RST}}$ signal is asserted. The $\overline{\text{RST}}$ signal cannot be masked. Reset exception processing forces the MC88110 into a predefined initial state. No pending exceptions or partially executed instructions are retained, the VBR is cleared, and the PSR and bus signals enter predefined states.

The exception vector for reset is vector zero. Since the VBR is forced to zero, the reset exception vector resides at logical memory address zero.

7.5.8 Address Translation Cache (ATC) Miss Exception

ATC miss exceptions are provided in support of software table searches. Software table searches are enabled by setting the software table search enable (STEN) bit in the data MMU/cache control register (DCTL) or the instruction MMU/cache control register (ICTL) for data and instruction accesses, respectively. When the STEN bit is set, ATC misses trap through either the instruction or data MMU ATC miss vectors and the handler routine performs the table search. The virtual address of the faulting reference is stored in the ILAR or the DLAR.

7

7

SECTION 8
MEMORY MANAGEMENT UNITS

This section provides a description of the MC88110 instruction and data memory management units (MMUs). Features described in this section are implemented by both MMUs unless explicitly stated otherwise.

The primary functions of each MMU are to translate logical to physical addresses for memory accesses and to provide access protection on a page basis. Instruction accesses are always read accesses generated by the instruction unit of the MC88110 to fetch instructions for execution, and data accesses are generated by the load and store instructions of the MC88110 programming model. The MC88110 does not support instruction cache coherence with data accesses (load and store instructions can not generate instruction accesses).

The MC88110 contains independent instruction and data MMUs that each provide separate 4G-byte supervisor and user logical address spaces with a 4K-byte page size and software selectable 512K-byte–64M-byte block size capability. Each MMU contains a 32-entry fully associative page address translation cache (PATC) and an 8-entry fully associative block address translation cache (BATC). BATC entries are loaded by software, and PATC entries may be loaded by the MC88110 hardware table search algorithm or optionally by software. The hardware table search algorithm used by the MC88110 supports two-level page tables with optional indirection. Additionally, the data MMU contains two breakpoint registers that trap on selected data accesses.

This section describes the address translation mechanisms provided by the MMUs as well as the various MMU conditions that cause MC88110 exceptions. In addition, the use of data breakpoints and the probing of address translation cache (ATC) entries is described. Refer to **Section 7 Exceptions** for more detailed information on exception processing with the MC88110.

8.1 MMU OVERVIEW

Logical address spaces can be divided into large regions called blocks, small regions called pages, or a combination of the two. For each block or page, the operating system creates an address descriptor that is used by the appropriate MMU to generate the physical address and the protection and other access control information when an address within the block or page is accessed. Address descriptors reside in tables in external physical memory; for faster accesses, the MMUs maintain on-chip copies of recently used descriptors in ATCs.

The MC88110 MMUs and exception model support demand paged virtual memory. Virtual memory management permits execution of programs larger than the size of physical memory; demand paged implies that individual pages are loaded into physical memory from backing storage only when they are first accessed by an executing program.

The following paragraphs provide an overview of the high level organization and operational concepts of the MC88110 MMUs. In addition, a summary of all MMU control registers is provided. **8.2 Selection of Address Translation Mode** through **8.10 MC88110 and MC88200 MMU Differences** provide more detailed descriptions of the specific features of the MMUs and a detailed description of the MMU control registers.

8.1.1 MMU Organization

Figure 8-1 shows the conceptual organization of the instruction and data memory management units and their relationships to the other functional units in the MC88110. The instruction memory management unit (IMMU) and the instruction cache comprise the instruction memory unit (IMU). Similarly, the data memory unit (DMU) is comprised of the data memory management unit (DMMU) and the data cache. The IMU supports instruction fetches requested by the instruction unit, and the DMU supports data accesses performed by the data unit. The arrows in Figure 8-1 represent address paths within the MC88110.

Addresses generated under program control are called logical addresses. Physical addresses are used to access external memory and to access the on-chip instruction and data caches. The MMUs translate the higher order bits of logical addresses into the higher order bits of the corresponding physical address. The logical address consists of a 32-bit effective address plus a supervisor/user mode bit that corresponds to the supervisor/user mode bit in the processor status register (PSR) when the access was generated. Refer to **Section 2 Programming Model** for a detailed description of the PSR and the supervisor/user mode bit.

The lower order bits of logical addresses are always untranslated (i.e., logical equals physical for the lower order address bits). For cacheable accesses, the 12 lower order address bits are immediately available to the instruction cache or data cache, so the cache lookup begins concurrently with the address translation performed by the IMMU or DMMU.
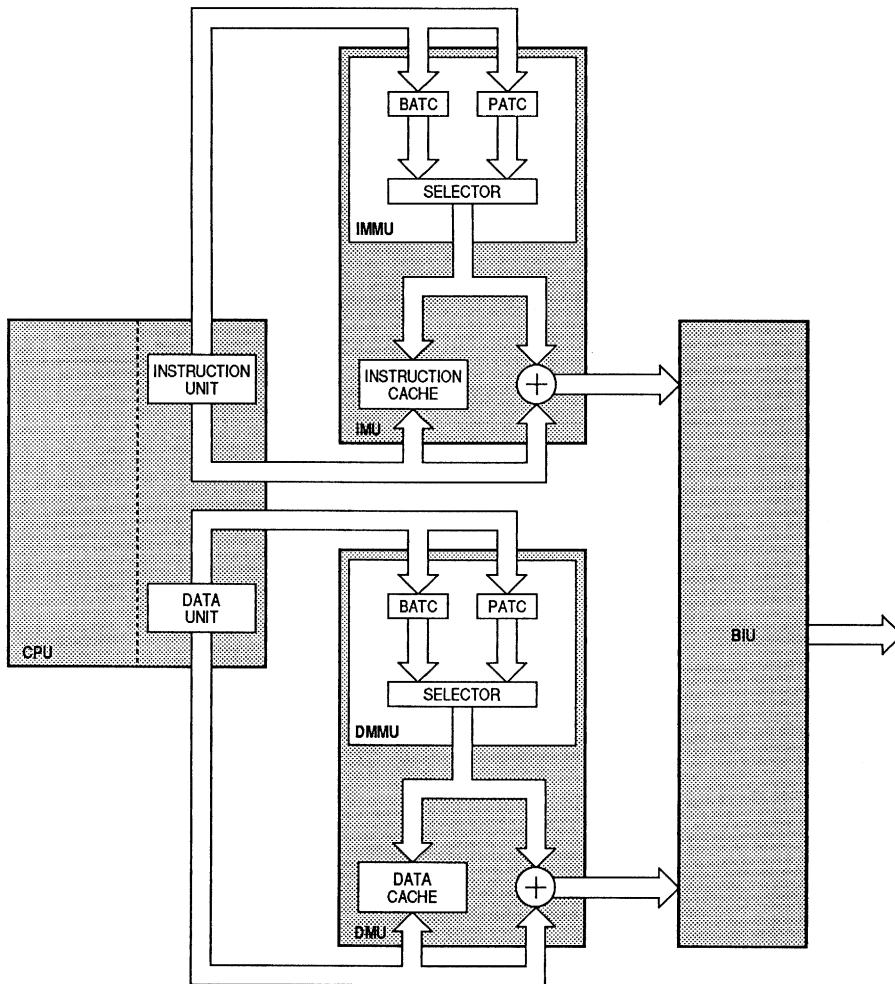
Figure 8-1. MC88110 MMU Block Diagram

Because the IMU and DMU each have their own MMU, address translations can be performed concurrently for instruction fetches and data accesses. After performing an address translation, the MMU passes the higher order bits of the physical address to the appropriate cache, and the cache lookup completes. For noncacheable accesses or accesses that miss in the instruction or data cache, the untranslated lower order address bits are concatenated with the translated higher order address bits. The resulting 32-bit physical address is then used by the bus interface unit (BIU), which performs an access to external memory. If an MMU is disabled (see **8.9 MMU/Cache Control Registers)**, the entire logical address is used untranslated to access the appropriate cache or external memory.

8.1.2 Block and Page Translation Capability

The MC88110 supports two forms of address translation: Address Translation:block address translation and page address translation. For block address translations, the logical address space of a program and physical memory are subdivided into regions called blocks. The size of a block is selectable by software, and can be in the range 512K-byte–64M-byte, varying by powers of two. Once a block size is selected, the same size applies for all blocks. For page address translations, the logical and physical memory spaces are subdivided into regions that are 4K-byte in size.

Within each MMU is a BATC and a PATC to support block and page address translations, respectively. Each BATC maintains 8 entries called block descriptors, each of which contains address translation information for a block. Each PATC maintains 32 page descriptors, which contain address translation information for each of 32 pages. Both of the ATCs are fully associative caches providing maximum hit rates.

When a logical address for an instruction or data access is generated, it is used concurrently by the BATC and PATC in the appropriate MMU. The ATCs compare the higher order bits of the logical address with the equivalent logical address bits of blocks or pages described within the ATC entries; if a comparison matches (ATC hit), then the address translation information in the matched descriptor is used to generate the corresponding physical address.

8.1.3 ATC Descriptor Concept

Figure 8-2 shows the page address descriptors located in PATC entries and how they are used to generate physical addresses. The page address descriptors contain four fields: a valid bit (V), a logical page address, a page frame address, and access control bits. The valid bit qualifies the remaining fields. If it is set, the ATC compares the logical page address field with the higher order bits of logical addresses generated by program accesses. The logical page address field contains the higher order bits of the address of a page in a program's logical address space. The page frame address contains the higher order bits of the address where the page resides in physical memory.

If an address comparison results in a match, the physical address for the access is formed by concatenating the contents of the page frame address field with the 12 lower order logical address bits. Additionally, the access control bits regulate the types of cache and external memory accesses that are performed to that page. For example, there is a write protect access control bit which can be used to force the MMU to abort write accesses to the described page.

BATC entries are conceptually identical to PATC entries. The only difference is that blocks are larger; more than 12 lower order address bits are untranslated to form the offset into a block. All blocks are 512K-byte–64M-byte in size and the block number fields have correspondingly fewer bits. Also, whereas the MC88110 may automatically load new PATC entries from memory, the system software is always responsible for loading the required block descriptors into the BATCs.
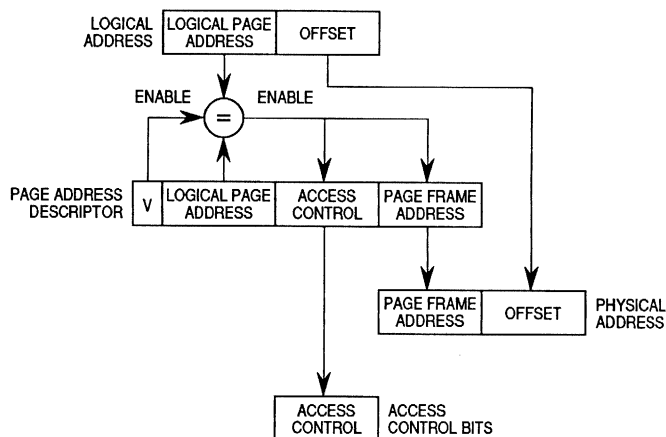
Figure 8-2. Address Translation with Page Address Descriptors in PATC

8.1.4 Table Search Options

Since it is unlikely that all descriptors for all pages fit within the PATC at one time, operating system software maintains tables of page descriptors in physical memory. When the comparisons performed by its BATC and PATC do not result in a match (ATC miss), an MMU cannot perform the address translation and access control with the address descriptors it contains on-chip. The MMUs of the MC88110 have the ability to automatically generate accesses to the page descriptor tables in physical memory (perform a hardware table search operation), in an attempt to locate a page descriptor for the logical address required by the program.

If the MC88110 locates a valid page descriptor when it is performing a hardware table search operation, the MMU automatically loads it into the PATC and resumes the address translation. However, if the operating system designer prefers a different structure for the table hierarchy (such as more or fewer levels, a different number of descriptors in tables at different levels, or compatibility with page descriptor tables shared with a non88000 family processor), then software table searching can be used.

When software table searching is desired, then hardware table searching must be disabled (see **8.4.4.1 Software Table Search Operations**). When table searching is enabled and the required descriptor is not resident in the ATCs, the MMU aborts the access. Software in an exception handler must search through the page tables and explicitly load the page descriptor into the PATC before restarting the aborted instruction.

8.1.5 Address Translation Modes

The MMUs of the MC88110 provide flexibility in controlling the mechanism for address translation. There are four address translation modes available to an operating system: the identity translation mode, the block-exclusive translation mode, the page-exclusive translation mode, and the combined BATC/PATC translation mode. Each mode is selected independently for the IMMU and DMMU. See **8.2 Selection of Address Translation Mode** for information on how the translation mode is selected.

In the identity translation mode, physical addresses are identical to logical addresses, so the MMU passes logical addresses directly to the cache or BIU with no address translation. Access control for accesses performed in this mode is regulated by the appropriate bits in the MMU control registers (see **8.9 MMU/Cache Control Registers**). Identity translation is always selected if the DEBUG signal is asserted, overriding software selection of other address translation modes.

In the block-exclusive translation mode, the MMU uses only the BATC to translate logical addresses into physical addresses and to obtain access control bits. This mode is often useful while executing programs that are permanently resident or completely swapped into physical memory prior to being executed. If an appropriate entry is not found in the BATC (BATC miss), then identity translation occurs, and no fault or exception is taken.

In the page-exclusive translation mode, the MMU translates logical addresses into physical addresses and obtains access control bits using only the PATC. In the event of a PATC miss, the MMU performs a hardware table search operation or causes the appropriate ATC miss exception to be taken (see **8.4.4.1 Software Table Search Operations**), depending on whether hardware table searching is enabled. If the hardware table search operation succeeds, the MMU automatically loads the necessary page descriptor into the PATC and uses it to complete the address translation; if the hardware table search operation fails, the MMU causes the instruction or data memory access exception (see **8.7 MMU/Cache Faults**) to be taken.

In combined block/page translation mode, the MMU attempts to translate logical addresses and obtain access control bits simultaneously in the BATC and in the PATC. This mode is often useful if a program executes in a demand paged virtual memory environment, but needs to access some permanently resident instructions or data or perform memory mapped I/O.

For example, an application with paged code/data may still need to access a frame buffer or call functions in a large shared library. The register file within a memory mapped I/O device, a frame buffer, or a shared library is permanently resident at a known physical address and can be described permanently with a single block descriptor, even if the remainder of the program's code and data are dynamically allocated in physical memory. In the combined block/page translation mode, in the event of a BATC hit, the MMU operates as in block-exclusive translation mode. In the event of a BATC miss, the MMU operates as in page-exclusive translation mode. Therefore, if a BATC hit and a PATC hit both occur for a logical address, the BATC entry is used.

8.1.6 General Flow of MMU Address Translation

Figure 8-3 shows the flow used by the MC88110 MMUs for address translation. When an instruction or data access is generated and the appropriate MMU is disabled, the logical address is used untranslated as the physical address, and the access continues in the instruction or data cache or on the external bus.
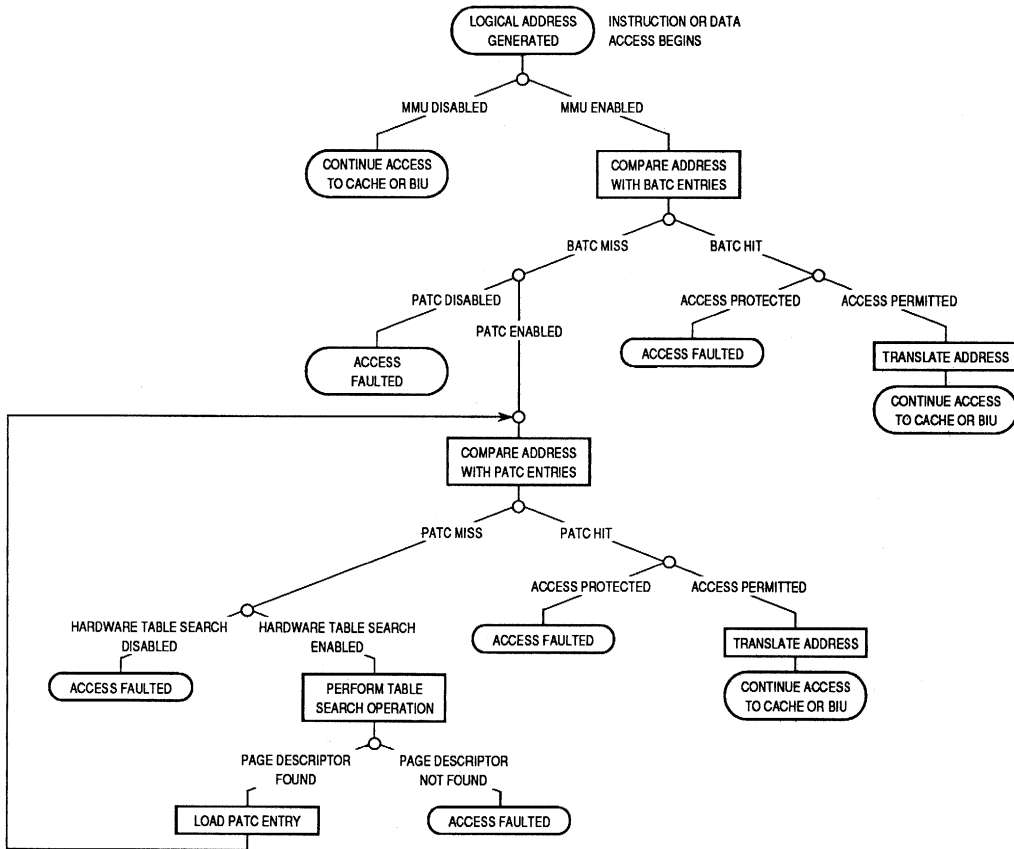
Figure 8-3. MMU Address Translation Flow

If the corresponding MMU is enabled and there is a hit in the BATC, a block address translation is performed unless the access is protected by an access control bit. Protected block accesses are faulted by causing the instruction or data memory access exception to be taken. If an appropriate entry is not found in the BATC (BATC miss), then identity translation occurs, and no fault or exception is taken. If the BATC misses, but the PATC is enabled and hits, a page address translation is performed unless the access is protected by an access control bit. If the access is protected, the access is faulted by causing the instruction or data memory access exception to be taken.

If the PATC misses and hardware table searching is disabled, then the access causes the instruction PATC miss, read data PATC miss, or write data PATC miss exception to occur. If the PATC misses but hardware table searching is enabled, then the MMU performs a search of the external page tables. If a valid page descriptor for the logical address is found, the MMU hardware loads it into the PATC, replaces the oldest entry (first-in-first-out (FIFO) replacement), and retries the PATC lookup. If the hardware table search operation fails to find a valid page descriptor, the access is faulted and the instruction or data memory access exception occurs.

8.1.7 MMU Exceptions and Faults Summary

Table 8-1 summarizes the exceptions caused by the MMUs. A more detailed description of the conditions that cause the exceptions is provided in **8.7 MMU/Cache Faults.** Refer to **Section 7 Exceptions** for a more detailed description of exception processing.

Table 8-1. MMU Exceptions Summary

Exception Number	Vector Base Address Offset	Exception
2	$10	Read DMMU PATC Miss
3	$18	Write DMMU PATC Miss
12	$60	IMMU PATC Miss
13	$68	Instruction Access
14	$70	Data Access

As shown in Table 8-1, the MMUs of the MC88110 can cause five exceptions. The PATC miss exceptions occur when a PATC miss occurs and hardware table search operations are disabled. These exceptions are used to vector to the exception handlers that perform the software table searches for these three conditions.

There are 9 conditions that can cause an MMU/Cache fault to occur. The faults then map to either the instruction or data access exception, depending on whether the access was an instruction or data access, as shown in Table 8-2.

Table 8-2. MMU/Cache Fault/Exception Mapping

Condition	Class	MC88110 Exception
Page Translations Enabled, No BATC Hit, and Hardware Table Searches Disabled	MMU PATC Miss	Read DMMU PATC Miss
Page Translations Enabled, No BATC Hit, and Hardware Table Searches Disabled	MMU PATC Miss	Write DMMU PATC Miss
Page Translations Enabled, No BATC Hit, and Hardware Table Searches Disabled	MMU PATC Miss	IMMU PATC Miss
Table Search Bus Error	MMU Fault	Instruction or Data Access
Segment Descriptor Invalid	MMU Fault	Instruction or Data Access
Page Descriptor Invalid	MMU Fault	Instruction or Data Access
Supervisor Protection Violation	MMU Fault	Instruction or Data Access
Write Protect Violation	MMU Fault	Instruction or Data Access
Data Breakpoint	MMU Fault	Data Access
Copyback Error	Cache Fault	Data Access
Write-Allocate Error	Cache Fault	Data Access
Bus Error (During Access)	Cache Fault	Instruction or Data Access

8.1.8 MMU Control Register Summary

Table 8-3 lists the control registers used by the IMMU. See **8.9 MMU/Cache Control Registers** for a detailed description of all the MMU control registers.

Table 8-3. Instruction MMU/Cache Control Register Summary

Register	Mnemonic	Description
cr25	ICMD	Instruction MMU/Cache/TIC Command Register —Invalidates ATC, cache, and TIC entries; used for probe commands
cr26	ICTL	Instruction MMU/Cache Control Register —Enables IMMU, cache, TIC, hardware table search, branch prediction, double instruction issue; freezes cache; selects BATC block size
cr27	ISAR	Instruction System Address Register —Specifies physical address for invalidate or logical address for probe
cr28	ISAP	IMMU Supervisor Area Pointer Register —Contains current supervisor instruction area descriptor
cr29	IUAP	IMMU User Area Pointer Register —Contains current user instruction area descriptor
cr30	IIR	IMMU ATC Index Register —Entry number for ATC accesses; R/W user attribute bits
cr31	IBP	IMMU BATC R/W Port —BATC descriptor
cr32	IPPU	IMMU PATC R/W Port - Upper —Upper word of PATC entry
cr33	IPPL	IMMU PATC R/W Port - Lower —Lower word of PATC entry
cr34	ISR	Instruction Access Status Register —Indicates status information for IMMU fault
cr35	ILAR	Instruction Access Logical Address Register —Logical address for IMMU fault
cr36	IPAR	Instruction Access Physical Address Register —Physical address related to IMMU fault

8

MC88110 USER'S MANUAL MOTOROLA

Table 8-4 lists the control registers used by the DMMU. See **8.9 MMU/Cache Control Registers** for a detailed description of all the MMU control registers.

Table 8-4. Data MMU/Cache Control Register Summary

Register	Mnemonic	Description
cr40	DCMD	Data MMU/Cache/TIC Command Register —Invalidates ATC and cache entries; copyback cache lines; used for probe commands
cr41	DCTL	Data MMU/Cache Control Register —Enables MMU, cache, cache snooping, hardware table search, breakpoint registers, decoupled cache accesses; freezes cache; forces write-through; controls **xmem** order; selects BATC block size
cr42	DSAR	Data System Address Register —Specifies physical address for invalidate or logical address for probe
cr43	DSAP	DMMU Supervisor Area Pointer Register —Contains current supervisor data area descriptor
cr44	DUAP	DMMU User Area Pointer Register —Contains current user data area descriptor
cr45	DIR	DMMU ATC Index Register —Entry number for ATC accesses; R/W user attribute bits
cr46	DBP	DMMU BATC R/W Port —BATC descriptor
cr47	DPPU	DMMU PATC R/W Port—Upper —Upper word of PATC entry
cr48	DPPL	DMMU PATC R/W Port—Lower —Lower word of PATC entry
cr49	DSR	Data Access Status Register —Indicates status information for DMMU fault
cr50	DLAR	Data Access Logical Address Register —Logical address for DMMU fault
cr51	DPAR	Data Access Physical Address Register —Physical address related to DMMU fault

8

8.2 SELECTION OF ADDRESS TRANSLATION MODE

Figure 8-4 shows the decision-making flow used by the MMUs to select the address translation mode.
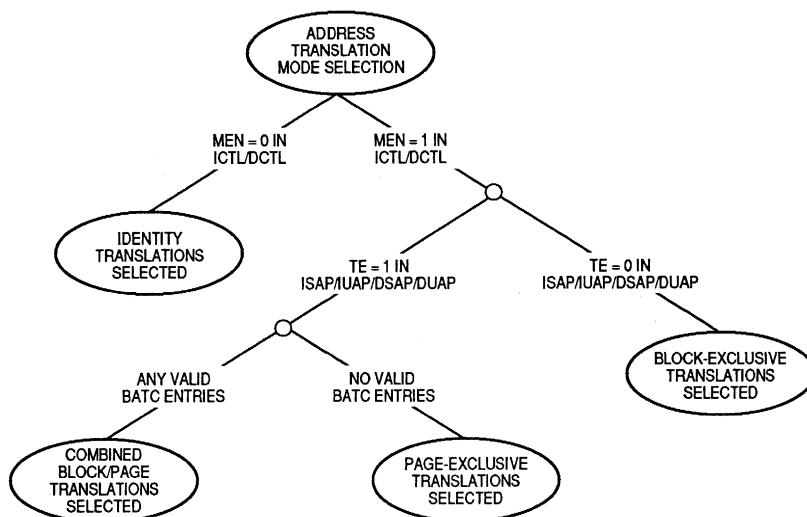
Figure 8-4. Address Translation Mode Selection

Table 8-5 summarizes the address translation and access control rules used for the various address translation modes.

Table 8-5. Address Mappings For Address Translation Modes

Translation Mode	MEN Bit in ICTL/DCTL	TE Bit in Area Pointer	Valid BATC Entry with Hit	Access Control Source	Address Mapping Source
Identity	0	x	x	Appropriate Area Pointer	1:1
Block-Exclusive	1	0	Yes	BATC Entry	BATC Entry
Block-Exclusive	1	0	No	Appropriate Area Pointer	1:1
Page-Exclusive	1	1	No	PATC Entry	PATC Entry
Combined Block/Page	1	1	x	BATC Entry or PATC Entry	BATC Entry or PATC Entry

x: don't care

8.2.1 Identity Translation

For the identity translation mode, access control is regulated by the appropriate area descriptor found in the supervisor and user area pointer registers (ISAP, DSAP, IUAP or IUDP). The identity translation mode is selected if the MEN bit in the ICTL or DCTL is clear. Identity translation mode is always selected if the DEBUG signal is asserted, overriding software selection of other address translation modes.

8.2.2 Block-Exclusive Translation

The block-exclusive translation mode is selected if the MEN bit in the ICTL or DCTL is set and the TE bit in the ISAP, IUAP, DSAP, or DUAP is clear. When this mode is selected, logical to physical address translation and access control bits are located in the BATC. Note that if a BATC miss occurs, the MMU generates an identity address translation and uses the access control bits in the ISAP, DSAP, IUAP, or DUAP for the access.

8.2.3 Page-Exclusive Translation

The page-exclusive translation mode is selected if the MEN bit in the ICTL or DCTL is set, the TE bit in the ISAP, IUAP, DSAP, or DUAP is set, and all BATC entries are marked as invalid. When this mode is selected and a PATC hit occurs, logical to physical address translation and access control bits are located in page descriptors in PATC entries. In the event of a PATC miss, the MMU performs a hardware table search operation or causes the appropriate ATC miss exception to be taken, depending on the value of the HTEN bit in the ICTL or DCTL.

8.2.4 Combined Block/Page Translation

The combined block/page translation mode is selected if the MEN bit in the ICTL or DCTL is set and the TE bit in the ISAP, IUAP, DSAP, or DUAP is set, and at least some BATC entries are marked as valid.

8.3 BLOCK ADDRESS TRANSLATION

This section describes block address translation in detail, including organization of the BATCs, formats of the block address descriptors, and software manipulation of BATC entries.

8.3.1 BATC Organization

The BATC in each MMU is an eight-entry fully associative cache. Figure 8-5 shows the conceptual organization of the BATCs for both the instruction and data MMUs. Each BATC entry contains a block descriptor, and because the BATC is fully associative, each entry is associated with a comparator. Each entry is shown as separated into a tag portion and a data portion.
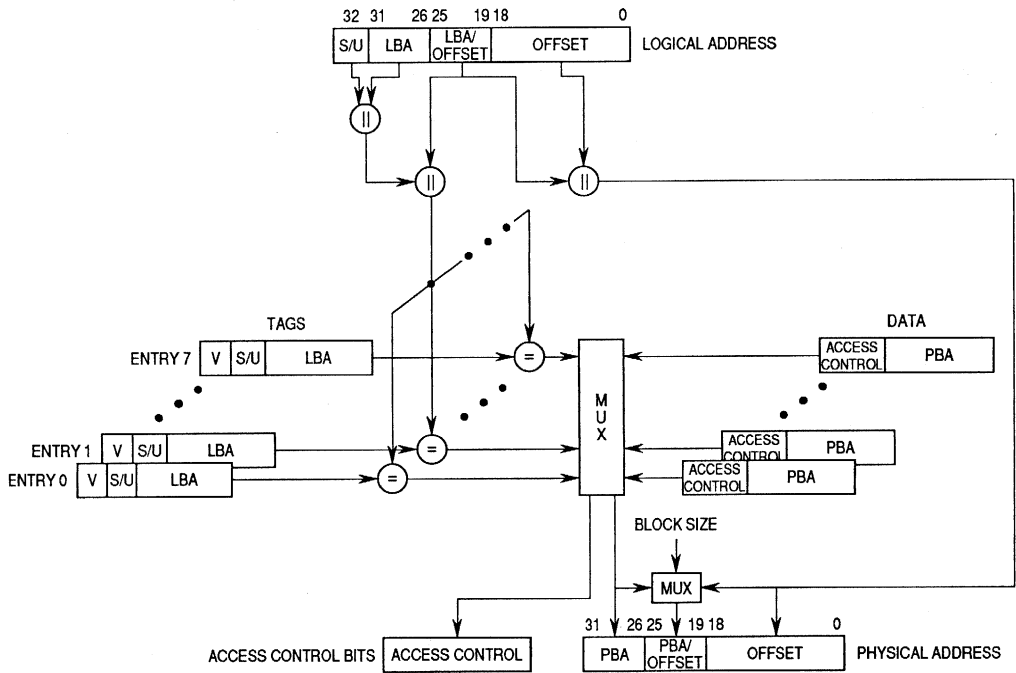
Figure 8-5. BATC Organization

The higher order logical address bits of an access, including the supervisor/user mode bit (determined by the mode bit of the PSR—see **Section 2 Programming Model**), comprise an input to each comparator. The other input to each comparator is comprised of the supervisor/user mode bit (S/U) and the logical block address field (LBA) from the tag of the associated entry. Comparators are enabled if the valid bit (V) for the associated entry is set.

If a comparison matches the access address with its associated descriptor tag (BATC hit), the data portion of the descriptor is multiplexed to obtain the higher order physical address bits (PBA) and access control bits for the access. Lower order address bits are not translated. For a block size of 512K-byte, bits 0–18 are untranslated and represent the offset of the access into a 512K-byte block (since $2^{19}=512K$). As the block size increases by powers of two, additional lower order address bits are untranslated. For example, for a block size of 1M-byte, 20 lower order bits are untranslated; for a block size of 64M-byte, 26 lower order address bits are untranslated.

8.3.2 Block Address Translation Flow

Figure 8-6 shows the detailed flow used for block address translations. In identity translation and page-exclusive translation modes, the BATC is unused. In block-exclusive and combined block/page translation modes, the logical address of the access is compared with BATC entry tags, as described in **8.3.1 BATC Organization**.
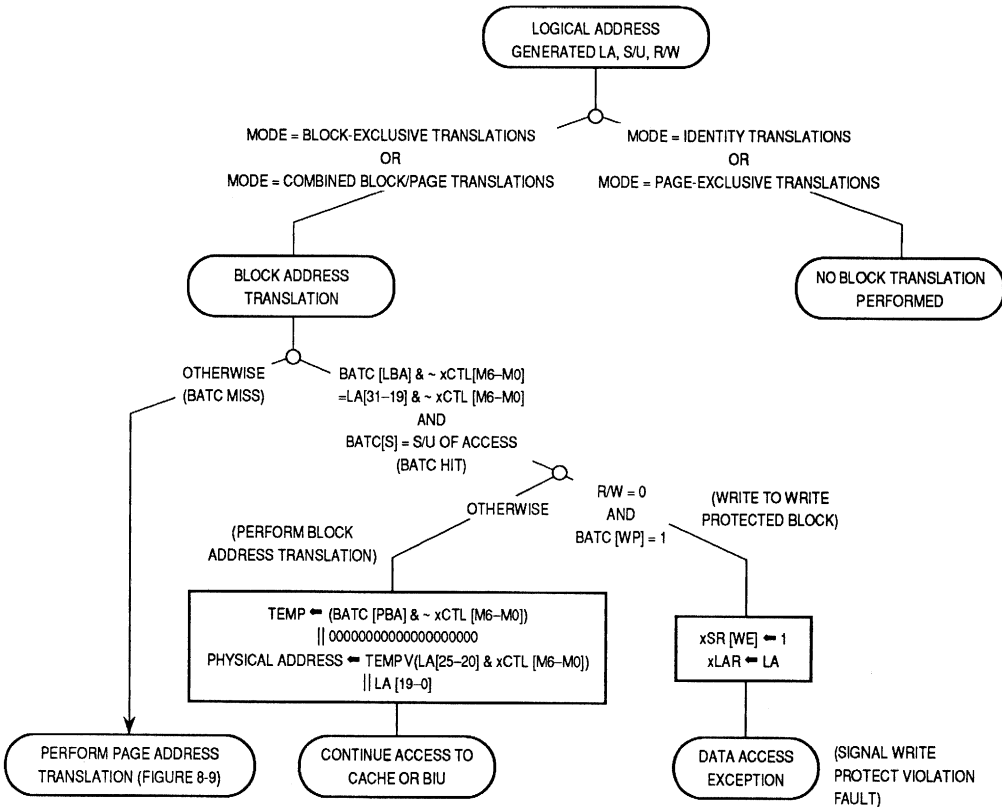
Figure 8-6. Block Address Translation Flow

If the BATC misses, page address translations may still be possible. If the BATC hits, but the access is a write operation to a write protected block, the MMU aborts the access and causes the data access exception to occur. Otherwise, the translated address is used to access the cache and/or physical memory.

8.3.3 BATC Descriptor Format

Both the instruction and data BATCs contain eight 34-bit entries each; each entry contains a single block descriptor. The format of a BATC entry is shown in Figure 8-7. All bits in all BATC entries, including the valid (V) bits, are undefined after a processor reset.

33	32	31	26	25	19	18	13	12	6	5	4	3	2	1	0
U1	U0	LBA			PBA					S/U	WT	G	CI	WP	V
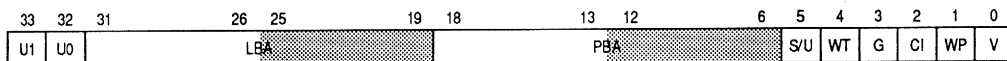
IGNORED DEPENDING ON BLOCK SIZE MASK. SIZE MASK WRITTEN VIA THE ICTL OR DCTL.

Figure 8-7. BATC Descriptor Format

U1, U0—User Attribute 1, 0

User attribute bits 0 and 1 are designated for use by the operating system. These bits are broadcast externally as the user attribute signals during external bus cycles if the physical address was generated by an MMU address translation. However, in some situations such as copyback operations, the physical address driven externally is not generated at that time by the MMU; thus, the user attribute signals are not driven to match the access in this case. Also, the user attribute bits are not stored in cache lines and are not checked by bus snooping logic. These two characteristics prevent their use as additional physical address bits in many environments. However, because they are driven externally on the first access to a block, it is possible for external hardware to use these signals to fault external bus cycles if a user-defined access is not permitted.

Note that the user attribute signals are active low, so a value of 1 for a user attribute bit is driven externally as a low voltage.

LBA—Logical Block Address

This field contains the most significant bits of the logical address of the block within a program's logical address space. The most significant bits of this field contain the most significant 6–13 bits of the logical address that map to the corresponding physical address, depending on the current block size selected via the ICTL or DCTL. Table 8-6 specifies the bits that are ignored for each possible block size.

Table 8-6. BATC LBA Bit Definition

Block Size	LBA Bits Ignored
512K-byte	None
1M-byte	Bit 19
2M-byte	Bits 20–19
4M-byte	Bits 21–19
8M-byte	Bits 22–19
16M-byte	Bits 23–19
32M-byte	Bits 24–19
64M-byte	Bits 25–19

PBA—Physical Block Address

This field describes the address of the block within the system's physical address space. It contains the most significant 6–13 bits of the physical block address, depending on the current block size selected via the ICTL or DCTL. Table 8-7 specifies which bits are ignored for each possible block size.

Table 8-7. BATC PBA Bit Definition

Block Size	Bits Ignored
512K-byte	None
1M-byte	Bit 6
2M-byte	Bits 7-6
4M-byte	Bits 8-6
8M-byte	Bits 9-6
16M-byte	Bits 10-6
32M-byte	Bits 11-6
64M-byte	Bits 12-6

S/U—Supervisor/User Mode

This bit is an extension to the LBA field.

1—LBA is for the supervisor logical address space
0—LBA is for the user logical address space

WT—Write-Through

This bit has no effect on the on-chip instruction cache; it is used only by the BATC in the DMMU. If the WT bit is set, then accesses that use this ATC entry use the write-through memory update policy. If this bit is clear, then the write-back memory update policy is used. The value of this bit is broadcast to the write-through signal of the external bus during single-beat accesses and read line fills. This permits write-through accesses to be extended to a secondary cache. Refer to **Section 6 Instruction and Data Caches** for more information on the memory update policies for the data cache.

1—Write-through memory update policy
0—Write-back memory update policy

8

G—Global

This bit has no effect on the on-chip instruction cache; it is used only by the BATC in the DMMU. If the G bit is set, then the block described by this entry is marked as containing globally shared data. The value of this bit is broadcast onto the global signal of the external bus during single-beat accesses, line fills, and invalidate cycles. This permits notification of global accesses to be broadcast to other caches in a multiprocessor system. If this bit is set, other caches may perform cache coherency checking (bus snooping). Refer to **Section 11 System Hardware Design** for more information on bus snooping for global accesses.

1—Block contains globally shared data
0—Block contains only locally referenced data

CI—Cache Inhibit

If the CI bit is set, then accesses through this entry are forced to miss in the on-chip cache and access external memory. The CI bit in the descriptor is broadcast onto the cache inhibit signal of the external bus during single-beat accesses, touch loads, and allocate loads. This permits cache inhibited accesses to be extended to a secondary cache. Refer to **Section 6 Instruction and Data Caches** for more information on cache inhibited accesses.

1—Block accesses are cache inhibited
0—Block accesses are cacheable

WP—Write Protect

This bit has no effect on read accesses, including all instruction fetches. If the WP bit is set, then the MMU aborts write accesses mapped through this entry by causing the data access exception to occur. If the bit is clear, write accesses are permitted to this block.

1—Write accesses are not allowed
0—Write accesses are allowed

V—Valid

This bit qualifies the validity of the BATC entry. If this bit is clear, then address translation and access control is not performed by the BATC using this entry.

1—Entry is valid
0—Entry is invalid

8.3.4 Sharing Blocks Between Programs

In order for multiple programs to share a block of physical memory, supervisor software must load a descriptor entry into the BATC before dispatching any of the programs. If the logical block address and access control bits are the same for all programs sharing the block, only one block descriptor is needed.

Because the S/U bit is an extension of the logical block address, separate block descriptors must be loaded in order for both a supervisor mode program and user mode

program to share a block of physical memory. Although both entries can specify the same physical block address, one specifies the user logical block address and the other specifies the supervisor logical block address.

8.3.5 Block Descriptor Maintenance

The MC88110 hardware never automatically modifies block descriptors in the BATC. All maintenance, including block descriptor invalidations, must be performed by supervisor mode software using the IIR and the IBP or the DIR and the DBP as described in the following paragraphs. Each of these registers is described in detail in **8.9 MMU/Cache Control Registers**.

It is considered a programming error if the system software loads more than one valid block descriptor with different address translation or access control bits for the same logical block address. It is unpredictable which BATC entry will be used when this situation occurs.

After a processor reset, all fields (including the valid bit) of the BATC entries are undefined. The system software must initialize each BATC with eight valid or invalid block descriptors before setting the MEN bit the ICTL or DCTL, enabling address translation.

8.3.5.1 SELECTING THE BLOCK SIZE. The block size used by the IMMU BATC is selected by programming the M6–M0 bits of the ICTL. The block size used by the DMMU BATC is selected by programming the M6–M0 bits of the DCTL. Table 8-8 shows the block sizes selected for the different encodings of these bits:

Table 8-8. Block Size Mask Bits in ICTL and DCTL

Block Size Mask Bits							Block Size
M6	M5	M4	M3	M2	M1	M0	
1	1	1	1	1	1	1	64M-byte
0	1	1	1	1	1	1	32M-byte
0	0	1	1	1	1	1	16M-byte
0	0	0	1	1	1	1	8M-byte
0	0	0	0	1	1	1	4M-byte
0	0	0	0	0	1	1	2M-byte
0	0	0	0	0	0	1	1M-byte
0	0	0	0	0	0	0	512K-byte
Any Other Combination							Undefined

8

8.3.5.2 LOADING BATC ENTRIES. The following steps describe the actions required for the system software to load a block descriptor into a BATC entry:

1. Select a BATC entry (0–7) to load with the new descriptor. If the block descriptor in the BATC that is to be replaced must be saved, the content of the selected entry can be read as described in **8.3.5.3 Reading BATC Entries**.

2. Create all 34 bits of the block descriptor, including the two user attribute access control bits.

3. Write the selected entry number and user attribute bits to the BATC index and BATC user attribute fields, respectively, in the IIR or DIR.

4. Write the remaining 32 descriptor bits to the IBP or DBP.

When the IIR or DIR is modified, the two user attribute bits are buffered and are not written into the BATC until the remaining 32 bits of the entry are written into the IBP or DBP.

8.3.5.3 READING BATC ENTRIES. The following steps describe the actions required for the system software to read a block descriptor from an entry in a BATC:

1. Select the BATC entry (0–7) to be read.

2. Write the selected entry number to the BATC index field in the IIR or DIR.

3. Read the IBP or DBP to cause the selected entry to be read from the BATC.

4. (Optional) Read the IIR or DIR to receive the two user attribute access control bits of the block descriptor.

8.3.5.4 INVALIDATING BATC ENTRIES. BATC entries are invalidated by loading a block descriptor with V=0 over the current contents of an entry, as described in **8.3.5.2 Loading BATC Entries**.

8.4 PAGE ADDRESS TRANSLATION

The following paragraphs describe page address translation in detail, including organization of the PATCs, formats of the page address descriptors, and maintenance of the PATCs.

8.4.1 PATC Organization

The PATC in each MMU is a 32-entry fully associative cache. Figure 8-8 shows the conceptual organization of the PATCs for both the instruction and data MMUs. Each PATC entry contains a page descriptor, and because the PATC is fully associative, each entry is associated with a comparator. Each entry is shown as separated into a tag portion and a data portion.
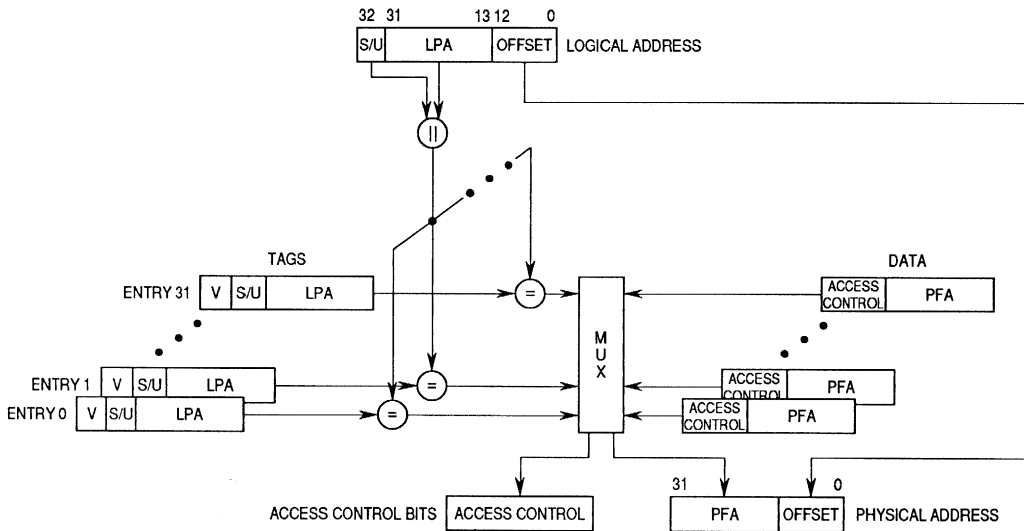
Figure 8-8. PATC Organization

The higher order logical address bits of an access, including the S/U bit (determined by the mode bit of the PSR—see **Section 2 Programming Model**), comprise an input to each comparator. The other input to each comparator is comprised of the S/U bit and LPA field from the tag of the associated entry. Comparators are enabled if the V bit for the associated entry is set.

If a comparison matches the access address with its associated descriptor tag (PATC hit), the data portion of the descriptor is multiplexed to obtain the higher order PFA bits and access control bits for the access. Lower order address bits are not translated. With a page size of 4K-byte, bits 0–11 are untranslated and represent the offset of the access into a 4K-byte page.

8.4.2 Page Address Translation Flow

Figure 8-9 shows the flow used for page address translations. In identity translation and block-exclusive translation modes, the PATC is unused. In combined block-page translation mode, if the BATC hits, the PATC is ignored. Otherwise, the logical address of the access is compared with PATC entry tags, as described in **8.4.1 PATC Organization**.
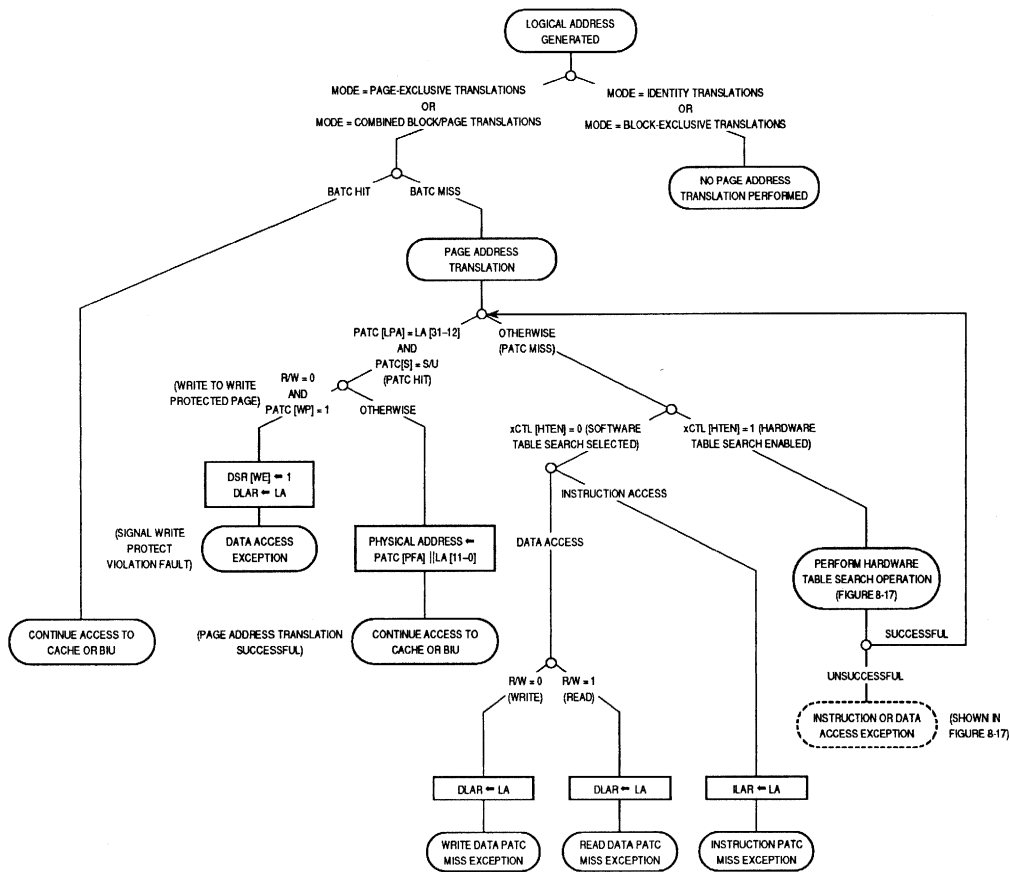
Figure 8-9. Page Address Translation Flow

If the PATC hits, but the access is a write operation to a write-protected page, the MMU aborts the access and causes the data access exception to occur. Otherwise, the translated address is used to access the cache and/or physical memory.

If the PATC misses and hardware table searching is disabled (xCTL[HTEN]=0), the MMU aborts the access and causes either the DMMU write miss exception, the DMMU read miss exception, or the IMMU miss exception to occur. If the PATC misses and hardware table searching is enabled (xCTL[HTEN]=1), the MMU performs a hardware table search operation. If the table search operation succeeds in loading the required page descriptor into the PATC, the address translation is retried. If the hardware table search operation fails, the MMU aborts the access and causes either the data or instruction access exception to occur.

8.4.3 PATC Descriptor Format

The instruction and data PATCs each contain thirty-two page descriptor entries. The format of a PATC entry is shown in Figure 8-10. All bits in all PATC entries, including the valid (V) bits, are undefined after a processor reset.

63		44	43	42	41	40	39	38	37	36	35	34	33	32
	LPA		0	0	0	0	0	0	0	0	0	0	0	S/U

UPPER WORD

31		12	11	10	9	8	7	6	5	4	3	2	1	0
	PFA		U1	U0	WT		G	CI				WP		V

LOWER WORD

▒ UNDEFINED-RESERVED FOR FUTURE USE
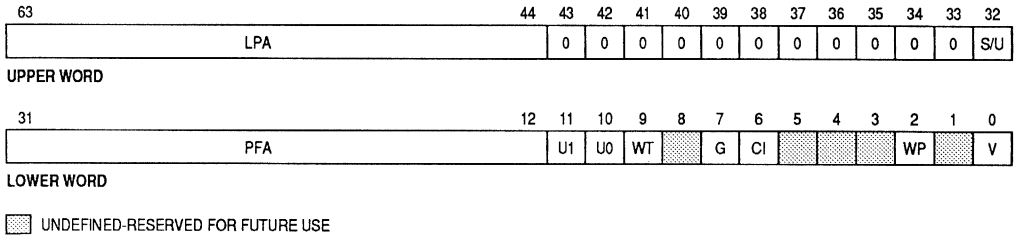
Figure 8-10. PATC Descriptor Format

LPA—Logical Page Address
 This field describes the address of a page within a program's logical address space. The most significant 20 bits of the logical address are in this field.

S/U—Supervisor Mode
 This bit is an extension to the LPA field.
  1—LPA is for the supervisor logical address space
  0—LPA is for the user logical address space

PFA—Page Frame Address
 This field contains the most significant 20 bits of the page frame address.

U1, U0—User Page Attribute 1, 0
 U0 and U1 are designated for use by the operating system. These bits are broadcast externally as the user attribute signals during external bus cycles if the physical address was generated by an MMU address translation. However, in some situations, such as copyback operations, the physical address presented externally is not generated at that time by the MMU; thus, the user attribute signals are negated in this case. Also, the user page attribute bits are not stored in cache lines, and so are not checked by bus snooping logic. These two characteristics prevent their use as additional physical address bits in many environments. However, because they are driven externally on the first access to a page, it is possible for external hardware to use these signals to fault external bus cycles if a user-defined access is not permitted.

 Note that the user attribute signals are active low, so a value of '1' for a user attribute bit is driven externally as a low voltage.

8

WT—Write-Through

This bit has no effect on the on-chip instruction cache and it is used only by the PATC in the DMMU. If the WT bit is set, then accesses that use this ATC entry use the write-through memory update policy. If this bit is not set, then the write-back memory update policy is used. The value of this bit is broadcast to the write-through signal of the external bus for single-beat accesses and read line fills. This permits write-through accesses to be extended to a secondary cache. Refer to **Section 6 Instruction and Data Caches** for more information on the memory update policies for the data cache.

    1—Write-through memory update policy
    0—Write-back memory update policy

G—Global

This bit has no effect on the on-chip instruction cache and it is used only by the PATC in the DMMU. If this bit is set, then the page described by this entry is marked as containing globally shared data. The value of this bit is broadcast on the global signal of the external bus during single-beat accesses, line fills, and invalidate cycles. This permits notification of global accesses to be broadcast to other caches in a multiprocessor system. If this bit is set, other caches may perform cache coherency checking (bus snooping). Refer to **Section 11 System Hardware Design** for more information on bus snooping for global accesses.

    1—Page contains globally shared data
    0—Page contains only locally referenced data

CI—Cache Inhibit

If this bit is set, then accesses through this entry are forced to miss in the on-chip cache and access external memory. The CI bit in the descriptor is broadcast on the cache inhibit signal of the external bus during the bus transaction for this access for single-beat accesses, touch loads, and allocate loads. This permits cache inhibited accesses to be extended to a secondary cache. Refer to **Section 6 Instruction and Data Caches** for more information on cache inhibited accesses.

    1—Page accesses are cache inhibited
    0—Page accesses are cacheable

WP—Write Protect

This bit has no effect on read accesses, including all instruction fetches. If the WP bit is set, then the MMU aborts write accesses mapped through this entry by causing the data access exception to occur. If the WP bit is clear, write accesses are permitted to this page.

    1—Write accesses are not allowed
    0—Write accesses are allowed

V—Valid

This bit qualifies the validity of the PATC entry. If this bit is clear, then address translation and access control is not performed by the PATC using this entry.

    1—Entry is valid
    0—Entry is invalid

8.4.4 Software Maintenance of PATC Entries

The MC88110 permits supervisor mode software to maintain PATC entries by loading new or updated page descriptors, reading existing entries, or invalidating entries. Software maintenance of PATC entries is possible whether or not hardware table search operation is selected.

Software maintenance is performed by supervisor mode software using the following registers: ICMD, IIR, IPPU, IPPL, DCMD, DIR, DPPU, and DPPL. Each of these registers is described in detail in **8.9 MMU/Cache Control Registers**.

It is considered a programming error if system software loads more than one valid page descriptor with different address translation or access control bits for the same logical page address. It is unpredictable which PATC entry will be used when this situation occurs.

After a processor reset, all fields (including the valid bit) of the PATC entries are undefined. The system software must initialize all entries in each PATC (typically by invalidating all entries) before enabling page address translations (see **8.2 Selection of Address Translation Mode**). Invalidation of all PATC entries can be performed most efficiently with the invalidate supervisor PATC and invalidate user PATC commands available through the ICMD and DCMD registers.

8.4.4.1 SOFTWARE TABLE SEARCH OPERATIONS.

If a PATC miss occurs and software table search operations are enabled (DCTL[HTEN] = 0, or ICTL[HTEN] = 0), either the read data MMU PATC miss, write data MMU PATC miss, or instruction MMU PATC miss exception occurs. The software table search operation should then be performed by the corresponding PATC miss exception handler.

If the PATC miss occurred in the IMMU, the ILAR contains the upper 27 bits of the logical address that could not be translated and the instruction MMU PATC miss exception is taken. If the PATC miss occurred in the DMMU, the DLAR contains the 32-bit logical address that could not be translated and either the read data MMU PATC miss or the write data MMU PATC miss exception occurs. The EPSR (see **Section 7 Exceptions**) and the ISR or DSR contain the supervisor/user indication for all PATC miss exceptions.

The MC88110 places no restrictions on the page descriptor table structure or table search algorithms to be used when software table search operation is selected (HTEN=0 in the ICTL or DCTL). The operating system designer is free to implement the translation table structure that best fits the system environment. However, the table search flow described in **8.5.3.2 Detailed Flow of Hardware Table Search Operation** can be used as a reference in order to ensure that the software table search process in the exception handler returns appropriate results to the PATCs. Also refer to **Section 7 Exceptions** for more information on exception processing and returning from exceptions.

8

8.4.4.2 LOADING PATC ENTRIES. The following steps describe the actions required for the system software to load a page descriptor into a PATC entry:

1. Select a PATC entry to load with the new descriptor. Possible entry numbers range from 0–31. If the page descriptor that is to be replaced must be saved, the contents of the selected entry can be read as described in **8.4.4.3 Reading PATC Entries**.

2. Create all 64 bits of the page descriptor, as described in **8.4.3 PATC Descriptor Format**.

3. Write the selected entry number to the PATC index field of the IIR or the PATC/breakpoint index field of the DIR, depending on whether the page descriptor is being written for the IMMU or DMMU.

4. Write the upper 32 descriptor bits to the IPPU or DPPU.

5. Write the lower 32 descriptor bits to the IPPL or DPPL.

Steps 3, 4, and 5 must occur in the sequence shown because the upper 32 descriptor bits are buffered until the lower 32 descriptor bits are written.

Step 3 can be omitted when operating in software table search mode. Any time an MMU causes a PATC miss exception, the MMU preloads the logical page address and S/U bit into the IPPU or DPPU. Consequently, only the least significant 32 bits of the PATC entry need to be written into the IPPL or DPPL in order to load the new entry. Preloading the PATC R/W port upper register occurs regardless of the setting of the HTEN bit in the ICTL or the DCTL.

In order to successfully write a selected PATC entry while performing any software table search operation, ATC probe commands must not be issued and PATC misses must not be encountered between steps 3 through 5 described above. Either of these events can cause the contents of the IIR, IPPU and IPPL, or the DIR, DPPU and DPPL registers to change.

8.4.4.3 READING PATC ENTRIES. The following steps describe the actions required for the system software to read a page descriptor from an entry in a PATC:

1. Select the PATC entry to read. Possible entry numbers range from 0–31.

2. Write the selected entry number to the PATC index field of the IIR or the DIR, depending on whether an instruction or data page descriptor is required.

3. Read the lower 32 descriptor bits from the IPPL or DPPL.

4. Read the upper 32 descriptor bits from the IPPU or DPPU.

The steps outlined above must be performed in sequence. In order to successfully read a selected PATC entry, an ATC probe command must not be issued and PATC misses must not be encountered between steps 3 and 4 described above. Either of these events can cause the contents of the IIR, IPPU and IPPL, or the DIR, DPPU and DPPL registers to change.

8.4.4.4 INVALIDATING PATC ENTRIES. There are two types of PATC invalidation: invalidation of a single entry or invalidation of all PATC entries for the supervisor or user logical address space.

Before allocating a new logical page to a page frame, it is necessary to mark the appropriate page descriptor as invalid. It is also necessary to ensure that a copy of the page descriptor no longer remains in the corresponding PATC. This is achieved with a two step process:

1. Use the appropriate probe command to check whether the page descriptor is resident in the PATC. See **8.8.1 ATC Probe Commands**. If the probe operation returns a PATC miss (PH=0) in the ISR or DSR, then the descriptor is not resident in the PATC.

2. If the probe operation indicates that a PATC Hit (PH=1) occurred, then the PATC entry number is returned in the IIR or DIR. To invalidate the entry, it is necessary to clear the entry's valid bit by writing to the IPPL or DPPL (see **8.4.4.2 Loading PATC Entries**).

Following a task switch, if it is necessary to establish a new set of address descriptors, invalidation of all page descriptors in the PATCs that are associated with the previous task can be performed most efficiently by initiating the invalidate supervisor PATC or invalidate user PATC commands via the ICMD and DCMD (see **8.9.1.1 Instruction MMU/Cache/TIC Command Register (ICMD)** and **8.9.2.1 Data MMU/Cache Command Register (DCMD)**).

8.5 PAGE DESCRIPTOR TABLES

The following paragraphs describe the structure of the page descriptor tables and the format of the descriptors that are used by the MC88110 when hardware table search operation is selected (HTEN = 1 in ICTL or DCTL).

8.5.1 Page Translation Table Structure

If hardware table search operation is selected while operating in either page-exclusive or combined page/block translation mode and an MMU miss occurs, the MMU automatically creates a new PATC entry by performing a page table search operation in physical memory. The tables are partitioned into two levels: segment and page, with the area pointers to the two segment tables residing on-chip. Figure 8-11 shows the address translation table hierarchy used by the MC88110.
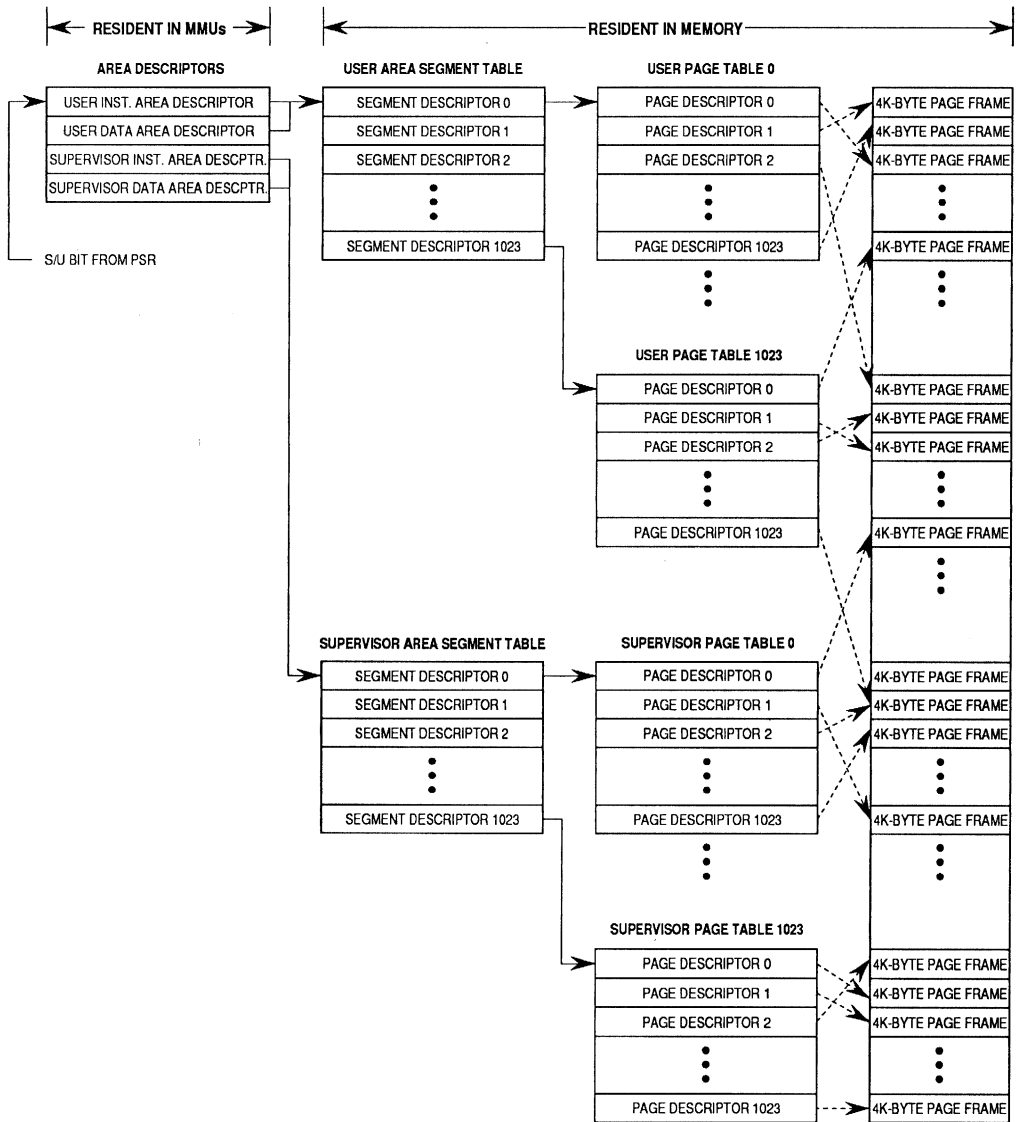
8

Figure 8-11. Page Translation Table Structure

At the top of the table hierarchy are area descriptors. Area descriptors comprise the root of the translation tables and are kept in on-chip registers. Four area descriptors are maintained, one each for user instruction, user data, supervisor instruction and supervisor data logical address spaces. Each area descriptor points to a table of 1024 segment descriptors in memory. Each valid segment descriptor then points to a table of 1024 page descriptors. Each valid page descriptor describes either the physical address for a page or points (via indirection) to another page descriptor.

Figure 8-12 illustrates how logical addresses are used to select address translation descriptors at each level of the table hierarchy. Within the MMUs, the S/U bit of the logical address of the access selects between the supervisor and user area descriptors. Refer to **8.5.3 Hardware Table Search Algorithm** for a detailed description of the flow used by the MC88110 for hardware table search operations.

Area descriptors contain a 20-bit segment table base address (STBA) field. The STBA field is concatenated with bits 31–22 of the logical address of the access to form the word-aligned physical address of the segment descriptor needed to continue the table search operation. Valid segment descriptors contain a 20-bit page table base address (PTBA) field. The PTBA field is concatenated with bits 21–12 of the logical access address to form the word-aligned physical address of the page descriptor needed to continue the table search.

A valid page descriptor contains a 20-bit page frame address (PFA) field, which is concatenated with bits 11–0 of the logical address of the access to form the translated physical address. Indirection descriptors are defined as descriptors at the page descriptor level of the translation tables that contain a 30-bit page descriptor address field, which is the word-aligned physical address of the actual page descriptor used to form the translated physical address.

In addition to address information, area, segment, and page descriptors contain access control bits. As the MMU performs the table search operation, it accumulates the access control bits by logically ORing them in order to create a PATC entry. If the table search completes successfully, the accumulated information is loaded automatically into a PATC entry.

Invalid descriptors (see **8.5.2 Translation Table Descriptor Formats**) can be used at any level of the hierarchy except at the area descriptor level. When a hardware table search operation encounters an invalid segment or page descriptor, the MMU causes the corresponding instruction or data access exception to occur.

8
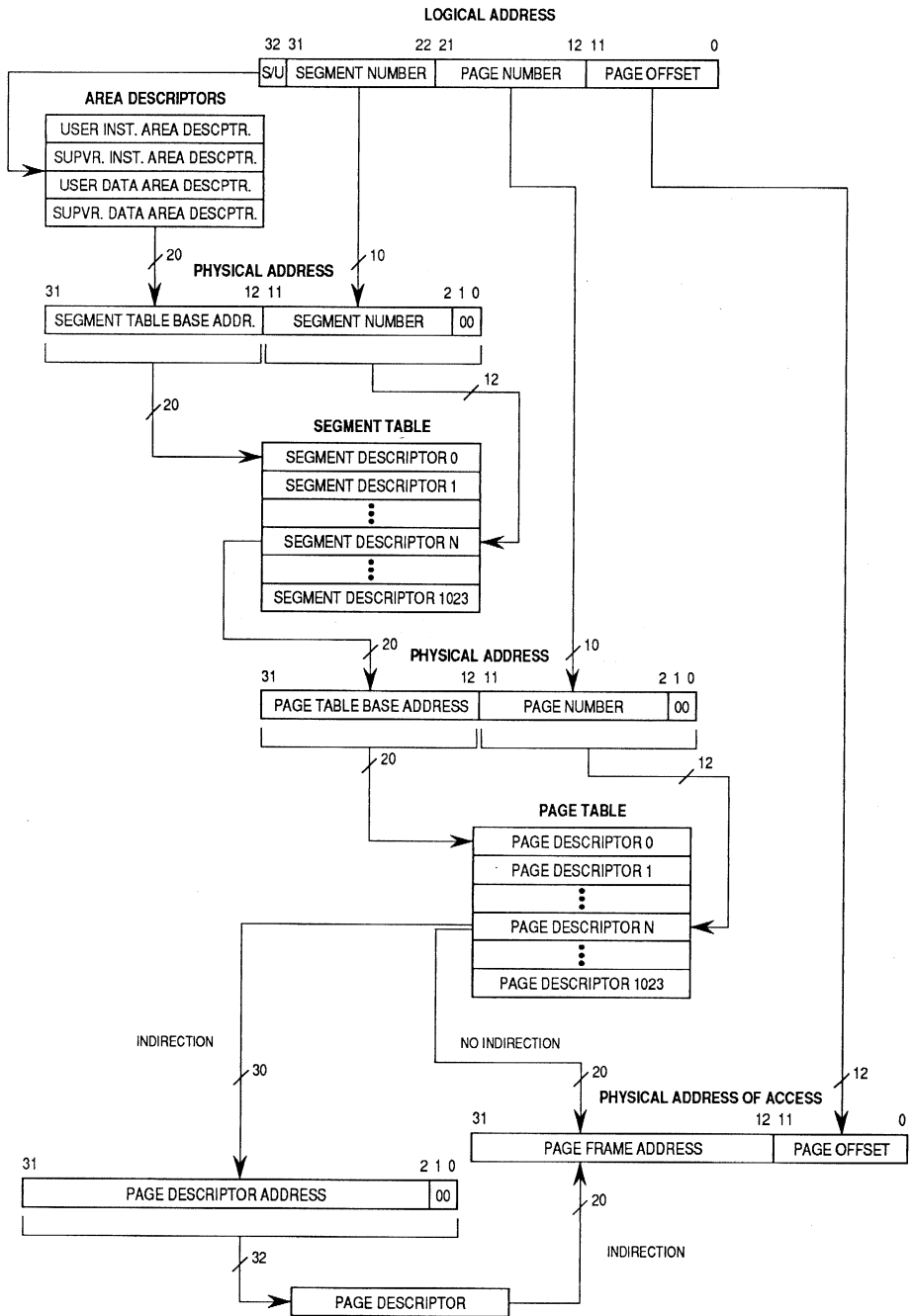
Figure 8-12. Page Table Lookup

8.5.2 Translation Table Descriptor Formats

The following paragraphs describe the formats for area, segment, page, and indirection descriptors to be used by system software when creating and maintaining the translation tables in memory in order to ensure correct searching of the tables and correct interpretation of status bits by the MC88110 hardware.

8.5.2.1 AREA DESCRIPTOR FORMAT. Area descriptors are maintained in on-chip registers (ISAP, IUAP, DSAP and DUAP). An area descriptor contains the physical base address of a segment table and access control bits for all pages within the area.

The TE bit of the area descriptors can be used to enable or disable the page address translation mechanism for an area, regardless of whether hardware or software table searching is selected. However, if hardware table searching is enabled, the TE bit of the area descriptor must not be set until a segment table has been created for the area. The access control bits of the area descriptor have no effect during a software table search.

Area descriptors have no effect on block address translations.

In identity translation mode, the access control bits in the area descriptor are used to control all accesses to the entire area.

The format of an area descriptor is shown in Figure 8-13.
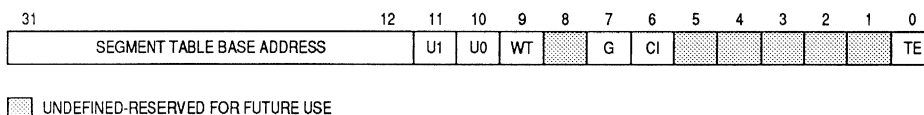
Figure 8-13. Area Descriptor Format

STBA—Segment Table Base Address

This field contains the most significant 20 bits of the segment table base address for a program's logical address space.

U1, U0—User Attribute 1, 0

U0 and U1 are designated for use by the operating system. They are driven onto the external bus during all bus cycles that comprise a hardware table search operation.

Note that the user attribute signals are active low, so a value of 1 for a user attribute bit is driven externally as a low voltage.

WT—Write-Through

If the WT bit is set, then cacheable accesses to pages that map through this area descriptor use the write-through memory update policy. This bit is an access control bit and is therefore accumulated (logically ORed) by the MC88110 with other WT bits encountered during a table search operation as the MMUs create PATC entries. Refer to **Section 6 Instruction and Data Caches** for more information on the memory update policies for the data cache.

    1—Write-through memory update policy in effect for the entire area
    0—Write-through versus write-back memory update policy controlled by the WT bit
        of the segment and page descriptors

G—Global

If this bit is set, the entire area contains globally shared data requiring bus snooping to maintain data cache coherency. This bit is an access control bit and is therefore accumulated (logically ORed) by the MC88110 with other G bits encountered during a table search operation as the MMUs create PATC entries. Refer to **Section 11 System Hardware Design** for more information on bus snooping for global accesses.

    1—Area contains globally shared data
    0—Global indication controlled by G bit of segment and page descriptors

CI—Cache Inhibit

If the CI bit is set, then accesses for the entire area are designated as noncacheable. Therefore, all accesses to the area are forced to miss in the on-chip cache and access external memory. This bit is an access control bit and is therefore accumulated (logically ORed) by the MC88110 with other CI bits encountered during a table search operation as the MMUs create PATC entries. Refer to **Section 6 Instruction and Data Caches** for more information on cache inhibited accesses.

    1—Entire area is cache inhibited
    0—Cache inhibit controlled by CI bit of segment and page descriptors.

TE—Translation Enable

The TE bit enables the page address translation mechanism. If TE is clear, then address translation and access control are not performed by the PATC for this area. This bit has no effect on the BATC.

    1—PATC enabled
    0—PATC disabled

8.5.2.2 SEGMENT DESCRIPTOR FORMAT. Each segment descriptor contains the physical base address of a page table. In addition, a segment descriptor contains access control bits, which are logically ORed with the access control bits from area descriptors. If a segment descriptor is valid when it is encountered by a hardware table search operation, then the table search continues to the page descriptor. If the segment descriptor is not valid, the table search operation is aborted and the appropriate instruction or data access exception occurs. The format of a segment descriptor is shown in Figure 8-14.

31	12	11	10	9	8	7	6	5	4	3	2	1	0
PAGE TABLE BASE ADDRESS				WT	SP	G	CI				WP		V
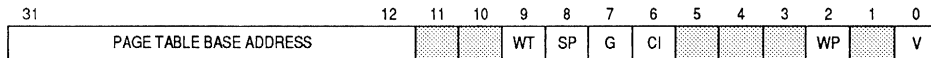
▨ UNDEFINED-RESERVED FOR FUTURE USE

Figure 8-14. Segment Descriptor Format

PTBA—Page Table Base Address

This field contains the most significant 20 bits of the page table base address for a program's logical address space.

WT—Write-Through

If this bit is set, then all cacheable accesses to pages that map through this segment descriptor use the write-through memory update policy. This bit is an access control bit and is therefore accumulated (logically ORed) by the MC88110 with other WT bits encountered during a table search operation as the MMUs create PATC entries. Refer to **Section 6 Instruction and Data Caches** for more information on the memory update policies for the data cache.

    1—Write-through memory update policy in effect for the entire segment
    0—Write-back memory update policy selected unless overridden by the WT bit of
        the area or page descriptors

SP—Supervisor Protection

If this bit is set, then hardware table search operations for user logical addresses within this segment are faulted by causing the instruction or data access exception to occur. If this bit is clear, the table search operation for a user access continues. This bit has no effect on supervisor logical address translations. When this bit is encountered during a table search operation, its value is saved for exception purposes, but it is not used to create the S/U bit in PATC entries.

    1—Translations can be performed only for supervisor accesses
    0—Translations continue for supervisor or user accesses

G—Global

If this bit is set, the entire segment contains globally shared data requiring bus snooping to maintain data cache coherency. This bit is an access control bit and is therefore accumulated (logically ORed) by the MC88110 with other G bits encountered during a table search operation as the MMUs create PATC entries. Refer to **Section 11 System Hardware Design** for more information on bus snooping for global accesses.

    1—Area contains globally shared data
    0—Segment contains only locally referenced data unless overridden by the G bit of
        the area or page descriptors

8

CI—Cache Inhibit

If the CI bit is set, then accesses for the entire segment are designated as noncacheable. Therefore, all accesses to the segment are forced to miss in the on-chip cache and access external memory. This bit is an access control bit and is therefore accumulated (logically ORed) by the MC88110 with other CI bits encountered during a table search operation as the MMUs create PATC entries. Refer to **Section 6 Instruction and Data Caches** for more information on cache inhibited accesses.

1—Entire segment is cache inhibited
0—Segment accesses are cacheable unless overridden by the CI bit of the area or page descriptors

WP—Write Protect

The WP bit selects whether write accesses to the segment are allowed. This bit is an access control bit and is therefore accumulated (logically ORed) by the MC88110 with the page WP bit encountered during a table search operation as the MMUs create PATC entries.

1—Entire segment is write protected
0—Segment write accesses are allowed unless overridden by the WP bit of the page descriptors

V—Valid

This bit indicates the validity of the segment descriptor. If clear, then address translation is not possible for the segment and the instruction or data access exception occurs when it is encountered in a table search operation.

1—Segment descriptor is valid
0—Segment descriptor is invalid and hardware table search is aborted

8.5.2.3 PAGE DESCRIPTOR FORMAT. A valid page descriptor contains the physical address of a page frame and access control bits that are logically ORed with the access control bits from area and segment descriptors. If an invalid page descriptor is encountered during a table search operation, the appropriate instruction or data access exception occurs. If the page descriptor is an indirect page descriptor, as specified by the DT field, the hardware table search continues by fetching another page descriptor. The format of valid and invalid page descriptors is shown in Figure 8-15. See **8.5.2.4 Indirection Descriptor Format** for a complete description of indirection descriptors.

31		12	11	10	9	8	7	6	5	4	3	2	1	0
PAGE FRAME ADDRESS			U1	U0	WT	SP	G	CI		M	U	WP	DT	
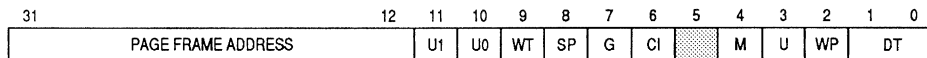
▓ UNDEFINED-RESERVED FOR FUTURE USE

Figure 8-15. Page Descriptor Format

PFA—Page Frame Address

This field contains the most significant 20 bits of the page frame address within a program's physical address space.

U1, U0—User Page Attribute 1, 0

U0 and U1 are designated for use by the operating system. They are driven onto the external bus for bus cycles that are mapped through this page descriptor.

Note that the user page attribute signals are active low, so a value of '1' for a user page attribute bit is driven externally as a low voltage.

WT—Write-Through

If the WT bit is set, then cacheable accesses to the page use the write-through memory update policy.This bit is an access control bit and is therefore accumulated (logically ORed) by the MC88110 with other WT bits encountered during a table search operation as the MMUs create PATC entries. Refer to **Section 6 Instruction and Data Caches** for more information on the memory update policies for the data cache.

    1—Write-through memory update policy in effect for the page
    0—Write-back memory update policy selected unless overridden by the WT bits of
        the area or segment descriptors

SP—Supervisor Protection

If this bit is set, then hardware table search operations for user logical addresses within this page are faulted by causing the instruction or data access exception to occur. If this bit is clear, the table search operation for a user access continues. This bit has no effect on supervisor logical address translations.

    1—Translations can be performed only for supervisor accesses
    0—Translations continue for supervisor or user accesses

G—Global

If this bit is set, the page contains globally shared data requiring bus snooping to maintain data cache coherency. This bit is an access control bit and is therefore accumulated (logically ORed) by the MC88110 with other G bits encountered during a table search operation as the MMUs create PATC entries. Refer to **Section 11 System Hardware Design** for more information on bus snooping for global accesses.

    1—Page contains globally shared data
    0—Page contains only locally referenced data unless overridden by the G bit of the
        area or segment descriptors

8

CI—Cache Inhibit

If the CI bit is set, then accesses for the page are designated as noncacheable. Therefore, all accesses to the page are forced to miss in the on-chip cache and access external memory. This bit is an access control bit and is therefore accumulated (logically ORed) by the MC88110 with other CI bits encountered during a table search operation as the MMUs create PATC entries. Refer to **Section 6 Instruction and Data Caches** for more information on cache inhibited accesses.

    1—Page is cache inhibited
    0—Page accesses are cacheable unless overridden by the CI bit of the area or
        segment descriptors

M—Modified

This bit indicates that write accesses have occurred within the page. This bit is maintained by supervisor software. See **8.5.4.2 Maintaining Modified Status** for more information on possible uses for the M bit.

    1—Page has been modified
    0—Page has not been modified

U—Used

This bit indicates that an access (read or write) has occurred to the page. This bit is maintained by supervisor software. See **8.5.4.1 Maintaining Used Status** for more information on possible uses for the U bit.

    1—Page has been accessed
    0—Page has not been accessed

WP—Write Protect

The WP bit selects whether write accesses to the page are allowed. This bit is an access control bit and is therefore accumulated (logically ORed) by the MC88110 with the segment WP bits encountered during a table search operation as the MMUs create PATC entries.

    1—Page is write protected
    0—Page write accesses are allowed unless overridden by the WP bit of the
        segment descriptor

DT—Descriptor Type

This field indicates the validity of the page descriptor and identifies the page descriptor type. If this field is clear, then address translation is not possible for the page and the instruction or data access exception occurs when it is encountered in a table search operation. For more information on indirection descriptors, see **8.5.2.4 Indirection Descriptor Format**.

    00—Descriptor is invalid and hardware table search fails
    01—Descriptor is a valid page descriptor with the format shown above
    10—Descriptor is a masked protection indirection descriptor
    11—Descriptor is a nonmasked protection indirection descriptor

8.5.2.4 INDIRECTION DESCRIPTOR FORMAT. Indirection descriptors (with or without masked protection) indicate that the hardware table search operation should perform another read operation to fetch another page descriptor. The indirection descriptor contains the physical address for the actual page descriptor. If the page descriptor is valid, it is used to obtain the physical address of the page frame. If it is another type of page descriptor, including another indirection descriptor, then an address translation is not possible for this access and the instruction or data access exception occurs.

The difference between the two types of indirection descriptors (with or without masked protection) lies in the location of the access control bits used to access the page. For nonmasked protection indirection descriptors (DT=11), the logical OR of the access control bits in the area, segment, and page descriptors are used in creating the PATC entries for the page. For masked protection indirection descriptors (DT=10), the access control bits in the page descriptor are ignored, and only the access control bits found in area and segment descriptors are used in creating the PATC entries for the page.

Figure 8-16 shows the format of an indirection descriptor.

31	2	1	0
PAGE DESCRIPTOR ADDRESS		DT	
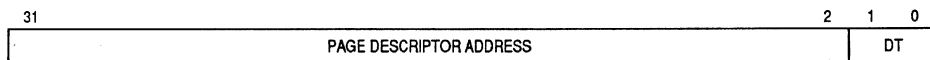
Figure 8-16. Indirection Descriptor Format

PDA—Page Descriptor Address
  This field contains the most significant 30 bits of the physical address of the actual page descriptor address.

DT—Descriptor Type
  This field indicates the validity of the page descriptor and identifies the page descriptor type. If this field is clear, then address translation is not possible for the page and the instruction or data access exception occurs when it is encountered in a table search operation. For more information on validity, see **8.5.2.3 Page Descriptor Format**.
    00—Descriptor is invalid and hardware table search fails
    01—Descriptor is a valid page descriptor with the format shown above
    10—Descriptor is a masked protection indirection descriptor
    11—Descriptor is a nonmasked protection indirection descriptor

8.5.3 Hardware Table Search Algorithm

The following paragraphs describe in detail the table search algorithm used by the MC88110 when hardware table searching is enabled (HTEN bit of ICTL or DCTL is set). The faults caused by table search operations and some timings for table search operations are also described.

8.5.3.1 TABLE SEARCH FAULTS. When a table search operation causes an MMU fault, state information is saved in MMU/cache control registers and either the instruction or data access exception occurs.

The state information saved for the possible table search fault conditions is summarized in Table 8-9. The fault state information can then be used by the appropriate exception handler in order to determine the corrective action to be taken for each fault condition. Refer to **8.9 MMU/Cache Control Registers** for a detailed description of the bits in the MMU/cache control registers. Note that Table 8-9 is a subset of Table 8-14 provided in **8.7 MMU/Cache Faults** that describes the state information saved for all MMU/Cache fault conditions.

Table 8-9. Table Search Fault Saved State Summary

Table Search Fault	Status Register Bit (ISR, DSR)	ILAR/DLAR	IPAR/DPAR
Table Search Bus Error	TBE = 1	Logical address of initial instruction or data access	Physical address of faulted bus cycle
Segment Descriptor Invalid	SI = 1	Logical address of initial instruction or data access	Physical address of invalid segment descriptor
Page Descriptor Invalid	PI = 1	Logical address of initial instruction or data access	Physical address of invalid page descriptor
Supervisor Protection Violation	SP = 1	Logical address of initial instruction or data access	Physical address of violation segment or page descriptor
Write Protect Violation	WE = 1 (DSR only)	Logical address of initial data write access	DPAR and IPAR undefined

The following paragraphs describe the faults that are signaled by the MMUs of the instruction and data memory units and that cause instruction or data access exceptions to occur.

8.5.3.1.1 Table Search Bus Error. If a bus error occurs during an external memory access for a hardware table search operation, the MMU aborts the table search operation and saves state information for the table search bus error fault by setting the TBE bit in the ISR or DSR. The logical address of the initial instruction fetch or data access is automatically saved in the ILAR or DLAR. The physical address that was driven onto the external bus for the faulted bus cycle is automatically saved in the IPAR or DPAR.

8.5.3.1.2 Segment Descriptor Invalid. If a hardware table search operation fetches a segment descriptor which is marked as invalid (V=0 in the segment descriptor), the MMU aborts the table search and saves state information for the segment descriptor invalid fault by setting the SI bit in the ISR or DSR. The logical address of the initial instruction fetch or data access is automatically saved in the ILAR or DLAR. In addition, the physical address of the fetched invalid segment descriptor is automatically saved in the IPAR or DPAR.

8.5.3.1.3 Page Descriptor Invalid. If a hardware table search operation fetches a page descriptor which is marked as invalid (DT=00) or a second indirect page descriptor (DT=10 or DT=11), the MMU aborts the table search and saves state information for the page descriptor invalid fault by setting the PI bit in the ISR or DSR. The logical address of the initial instruction fetch or data access is automatically saved in the ILAR or DLAR. In addition, the physical address of the fetched invalid or second indirect page descriptor is automatically saved in the IPAR or DPAR.

This condition is often used by the system software as an indication that a page fault has occurred and that a new page frame must be created in physical memory for the accessed page. Refer to **8.5.4 Page Descriptor Table Considerations** for more information on possible uses for the invalid page descriptor condition.

8.5.3.1.4 Supervisor Protection Violation. If a hardware table search operation for a user logical address fetches a segment or page descriptor marked as supervisor protected (SP=1 in either descriptor), the MMU aborts the table search and saves state information for the supervisor protection violation fault by setting the SP bit in the ISR or DSR. The logical address of the initial instruction fetch or data access is automatically saved in the ILAR or DLAR. In addition, the physical address of the fetched segment or page descriptor is automatically saved in the IPAR or DPAR.

8.5.3.1.5 Write Protect Violation. If a write access is attempted to a memory location marked as write protected (WP=1 in the ATC entry mapping the logical address), the DMMU faults the access and saves state information for the write protect violation fault by setting the WE bit in the DSR. The logical address of the initial data write is automatically saved in the DLAR. The contents of the DPAR are undefined after this fault occurs. Write protection violation faults are not applicable for the IMMU.

8.5.3.2 DETAILED FLOW OF HARDWARE TABLE SEARCH OPERATION. Figure 8-17 shows the logical flow used by the MMUs to perform hardware table search operations. The table search operation begins with the selection of the appropriate area descriptor according to the value of the S/U bit that is part of the logical address. A pointer to the required segment descriptor is created by concatenating the STBA field of the area descriptor with bits 31–22 of the logical address for the access. Access control bit accumulation begins by initializing a temporary status accumulator with the values of the access control bits in the selected area descriptor.

8
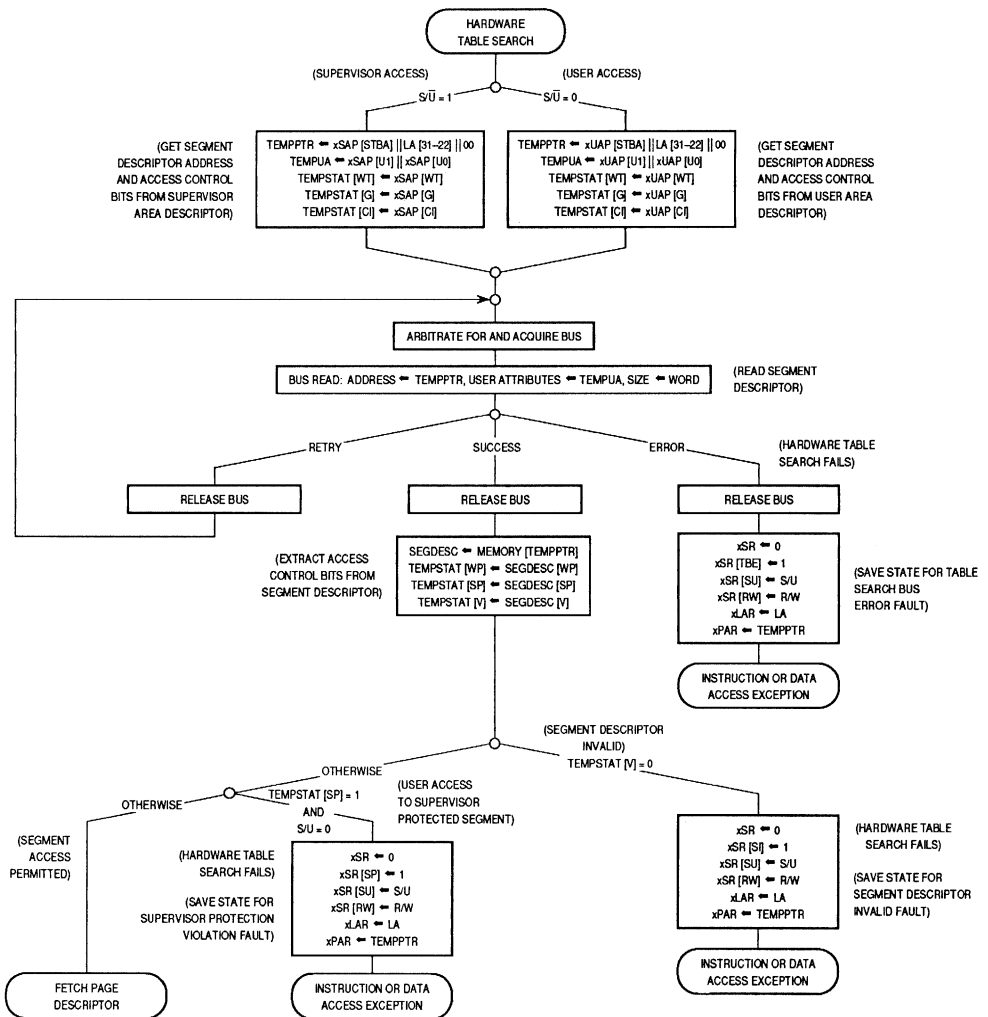
Figure 8-17. Hardware Table Search Flow (Sheet 1 of 3)
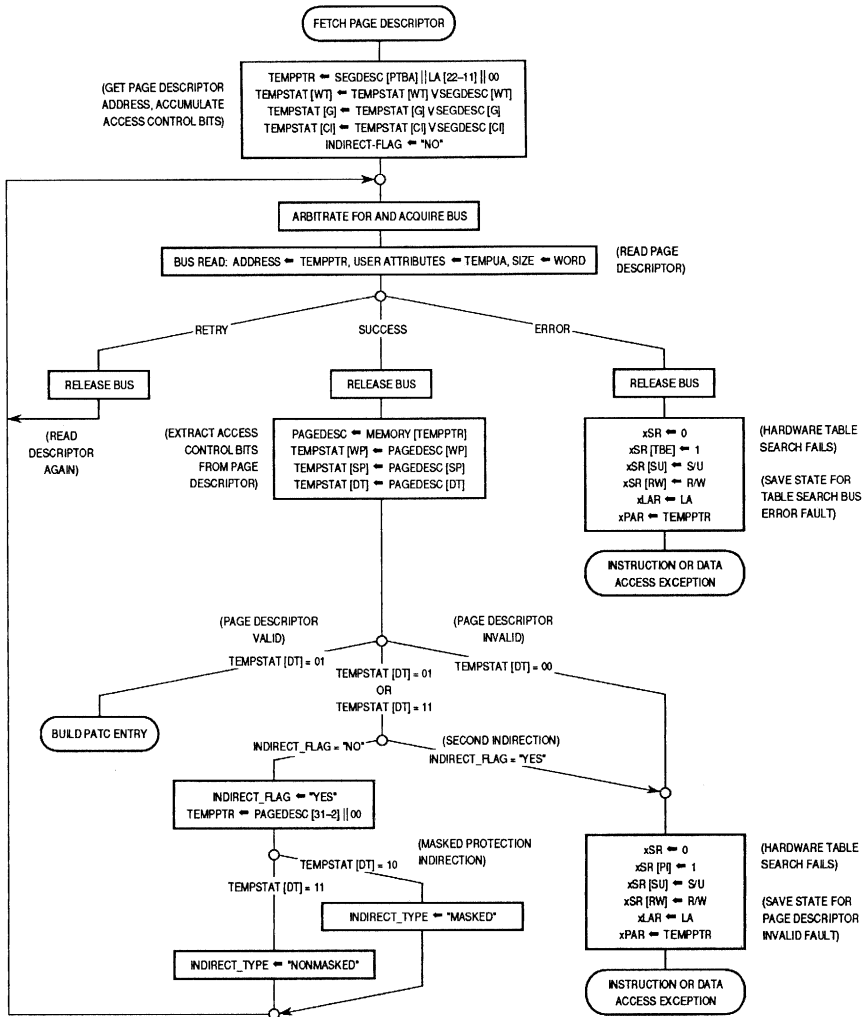
Figure 8-17. Hardware Table Search Flow (Sheet 2 of 3)
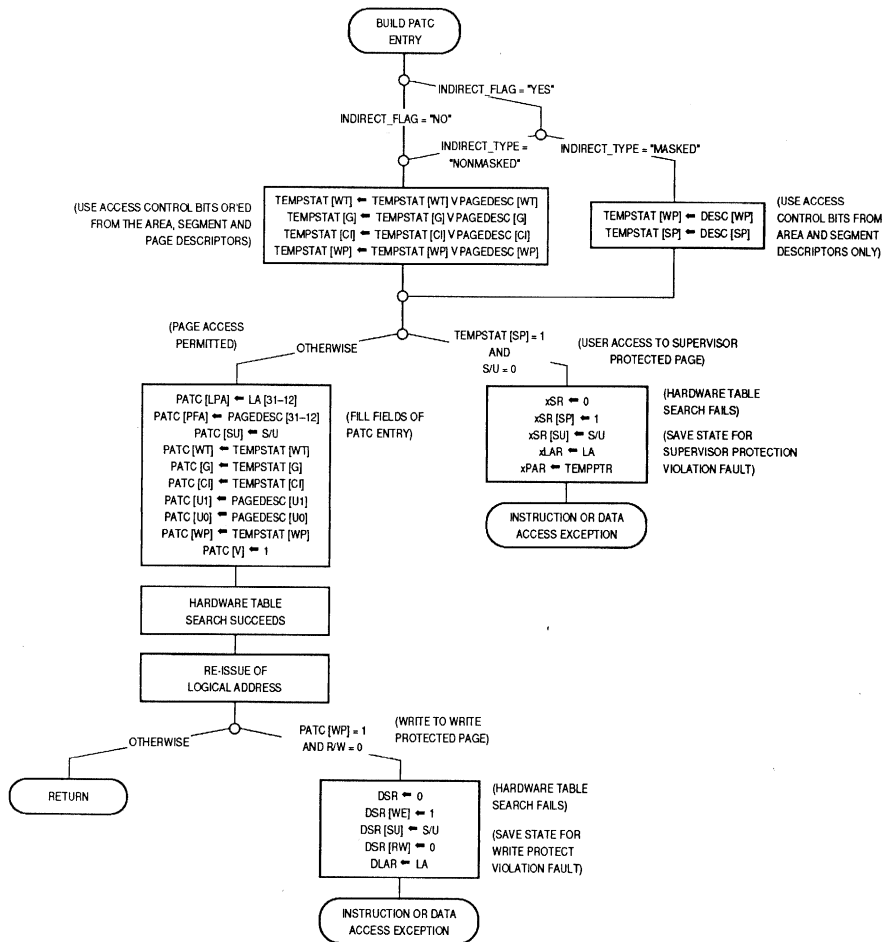
Figure 8-17. Hardware Table Search Flow (Sheet 3 of 3)

The table search operation continues as the BIU arbitrates for mastership of the external bus and fetches the required segment descriptor. If a bus error occurs during the bus transaction, the MMU saves state information related to the bus error and causes the instruction or data access exception to occur with the status bits in the ISR or DSR indicating a table search bus error fault.

If the bus cycle terminates with a retry, the segment descriptor fetch is repeated. If the segment descriptor fetch succeeds but the valid bit of the segment descriptor is clear, the MMU saves state information for the segment descriptor invalid fault, and the table search is aborted (instruction or data access exception occurs). If the access is a write access and the segment is marked as write protected, the MMU completes the table search and loads the new PATC descriptor; upon reissue of the logical address, a write protection violation fault occurs.

If the original access is made by a user mode program but the segment is marked as supervisor protected, the MMU saves state information for the supervisor protection violation fault, and the table search is aborted (instruction or data access exception occurs). Otherwise, the segment descriptor is valid and access to the segment is permitted. A pointer to the required page descriptor is created by concatenating the PTBA field of the segment descriptor with bits 21–12 of the logical address of the access. Access control bits are accumulated by logical ORing the temporary status accumulator with the values of the access control bits from the segment descriptor.

The table search operation continues as the BIU arbitrates for mastership of the external bus again and fetches the required page descriptor. If a bus error occurs during the bus transaction, the MMU saves state information related to the bus error and causes the instruction or data access exception to occur with the status bits in the ISR or DSR indicating a table search bus error fault.

If the bus cycle terminates with a retry request, the page descriptor fetch is repeated. If the page descriptor fetch succeeds, but its type is invalid, the MMU saves state information for the page descriptor invalid fault, and the table search is aborted (instruction or data access exception occurs).

If the descriptor type is one of the indirection types and this is the first indirection descriptor encountered during this table search operation, an internal indirection flag is set. However, if another indirection descriptor has already been encountered in this table search operation (the internal indirection flag was set prior to this bus access), the MMU saves state information for the page descriptor invalid fault, and the table search is aborted (instruction or data access exception occurs). If this is the first indirection descriptor within this table search operation, the address of the required page descriptor is extracted from the indirection descriptor. An internal flag is set to indicate whether the indirection descriptor is a masked protection indirection descriptor or a nonmasked indirection descriptor, and the steps described for fetching a page descriptor are repeated.

At this point in the table search operation, the descriptor type must be valid page descriptor (DT = 01). Access control bits are accumulated by logically ORing the temporary status accumulator with the values of the access control bits from the valid page descriptor if there has been no indirection or if there has been nonmasked indirection. If a masked protection indirection descriptor has been encountered, the access control bits from the page descriptor are ignored.

If the access is a write access and the page is marked as write protected, the MMU completes the table search and loads the new PATC descriptor; upon reissue of the logical address, a write protection violation fault occurs. If the access is made by a user mode program but the page is marked as supervisor protected, the MMU saves state information for the supervisor protection violation fault, and the table search is aborted (instruction or data access exception occurs). Otherwise, access to the page is permitted, and a PATC entry can be created. The S/U bit and LPA field of the selected PATC entry are overwritten (FIFO replacement) with the 21 higher order bits of the logical address

(including the S/U bit) of the access. The PFA field of the PATC entry is filled from the PFA field of the valid page descriptor. The access control bits are filled from the temporary status accumulator and the page descriptor User Page Attribute bits. The PATC entry is marked as valid, and the table search operation completes successfully.

8.5.3.3. HARDWARE TABLE SEARCH OPERATION TIMING. Table 8-10 summarizes the clock cycle counts for a hardware table search operation. The table assumes bus availability (the bus is available when the arbitration occurs) and no-wait-state external memory.

Table 8-10. Hardware Table Search Operation Timing

Table Search Type	Clocks
No Indirection	9
Indirection	13

8.5.4 Page Descriptor Table Considerations

The following paragraphs describe the sharing of pages between programs, the paging of page descriptors, and some of the actions required by the system software to maintain current status bits in the page descriptor tables.

To prevent coherency problems with used and modified bits in multiprocessor environments, it is advised that exception handlers perform the updates to these bits in page descriptors as indivisible bus transactions through the use of the **xmem** instruction as described in **8.5.4.3 Sharing Pages.**

8.5.4.1 MAINTAINING USED STATUS. The U bit in the page descriptor tables can be used to indicate that the page has been referenced since it was loaded from backing storage. If the U bit is clear, the page has not been accessed. This bit is not maintained automatically by MC88110 MMU hardware; if its use is required, it must be maintained by operating system software.

Existing exception mechanisms can be used to maintain the used bit in software. This can be accomplished by encoding the U and V bits of the page descriptor as shown in Table 8-11.

Table 8-11. Used/Valid Bit Interpretations

Used Bit	Valid Bit	Interpretation
0	0	Not Used; Invalid
0	1	Does Not Apply
1	0	Not Used; Valid
1	1	Used; Valid

When the U and V bits are both zero, the descriptor is invalid. However when these bits are U = 1 and V = 0, the descriptor is valid but not used. Thus, when the descriptor is first encountered during a hardware table search operation, it is detected as an invalid descriptor and the instruction or data access exception occurs. The exception handler can then set both the U and V bits (U = 1 and V = 1) to designate the descriptor as valid and used. Note that when using the U and V bits in this way, the encoding of U = 0 and V = 1 is not used and should be avoided.

8.5.4.2 MAINTAINING MODIFIED STATUS. The M bit in a page descriptor can be used to indicate whether or not the page has been modified (dirtied) since it was loaded from backing storage. This bit is not maintained automatically by MC88110 MMU hardware.

The M bit can be maintained by software to update the status of a page descriptor when a write operation occurs to a page. Existing exception mechanisms can be used to maintain the modified status by using the M and WP bits of the page descriptor as shown in Table 8-12.

Table 8-12. Modified/Write Protect Bit Interpretations

Modified Bit	Write Protect Bit	Interpretation
0	0	Unmodified; Writeable (Causes Exception on Write)
0	1	Write Protected
1	0	Modified; Writeable
1	1	Does Not Apply

When the M and WP bits are encoded in this way, the page is designated as unmodified and writeable when the bits are M = 0 and WP = 0. When the page descriptor is loaded into the PATC, the WP bit in the PATC is automatically set by the DMMU (PATC WP = 1). Therefore, when the first write operation to this page occurs, the data access exception occurs. The exception handler can then set the M bit in the page descriptor (M = 1 and WP = 0) to indicate that the page has been modified and is writeable. When these bits are M = 0 and WP = 1, the page is write protected. Note that when using the M and WP bits in this way, the encoding of M = 1 and WP = 1 is not used and should be avoided.

8.5.4.3 SHARING PAGES. It is sometimes desirable for two or more program tasks to share the same physical page, perhaps with different logical addresses and/or access control bits for each task. In order to simplify maintenance of the M and U bits, it may be desirable to have only one valid page descriptor for a shared page. Additionally, many operating systems prefer to describe shared regions of memory in shared memory tables, rather than in distinct page descriptor tables.

The MC88110 supports page sharing using a single page descriptor with indirection descriptors (DT=11) and masked protection indirection descriptors (DT=10). Figure 8-18 shows a possible use of indirection to support a shared page of physical memory.

TASK A'S DESCRIPTORS

AREA DESCRIPTOR

SEGMENT DESCRIPTOR

INDIRECTION PAGE DESCRIPTOR

SEGMENT TABLE

PAGE TABLE

SHARED PAGE

TASK B'S DESCRIPTORS

AREA DESCRIPTOR

SEGMENT DESCRIPTOR

MASKED PROTECTION INDIRECTION PAGE DESCRIPTOR

SEGMENT TABLE

PAGE TABLE

VALID PAGE DESCRIPTOR

SHARED MEMORY TABLE
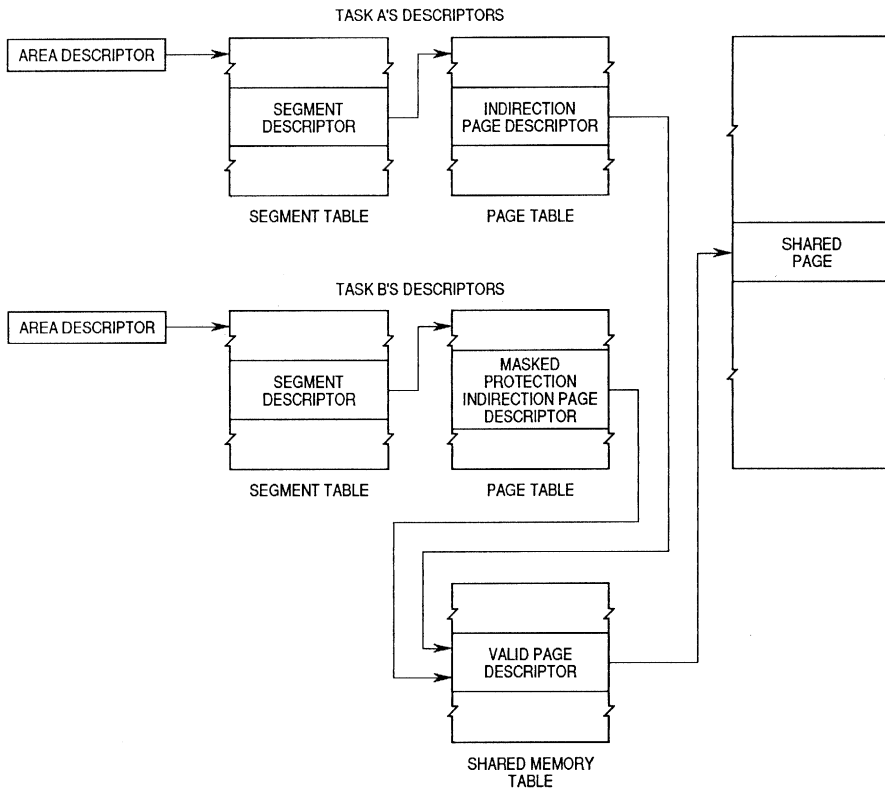
Figure 8-18. Shared Pages with Indirection Descriptors

For sharing pages of physical memory, the operating system can maintain a shared memory table, which includes a valid page descriptor. The PFA field of the valid page descriptor points to the physical address of the page shared by program tasks A and B.

Elsewhere in physical memory, the operating system has built segment and page descriptor tables describing the logical to physical mappings and access control bits for each task's address space that are used while the task is executing. The page descriptors in these tables for the logical addresses of the shared page are indirection or masked protection indirection descriptors. These indirection descriptors point to the actual address mapping located in the page descriptor of the shared memory table.

Note that the logical addresses of the shared page may be different for each task, since each task has its own indirection descriptor in the appropriate position in its descriptor table hierarchy for the desired logical address.

The difference between indirection and masked protection indirection lies in the access control bits used for the access. With indirection descriptors, the access control bits used are accumulated during a hardware table search operation in the same way as valid

page descriptors: by logically ORing access control bit values for the area, segment, and valid page descriptors. With masked protection indirection descriptors, the access control bits in the valid page descriptor are ignored. Use of indirection descriptors permits each program sharing the physical page to have unique access restrictions selected via its area and segment descriptors.

It is not uncommon in multiprocessor systems for processors in the same physical address space to share a common set of page tables. In this case, it is necessary that the page descriptors be kept in a consistent state for each processor. Inconsistent page descriptors can arise when U or M bits are updated for a shared task. For example, if two processors were to attempt to read a page descriptor simultaneously, one could immediately set the M bit while the second processor is attempting to clear the M bit. In this case, the modified status of the page would be lost. To prevent this situation, exception handlers should perform the update to the U and M bits in the page descriptor as indivisible bus transactions through the use of the **xmem** instruction.

8.5.4.4 PAGING SETS OF PAGE DESCRIPTORS. It is not necessary to keep all valid page descriptors resident in physical memory for an executing program, just as it is not necessary for all pages of program code or data to be resident in physical memory at one time. In other words, it is possible to dynamically load page tables into physical memory as demanded by program execution.

The paging of page tables may be performed by interpreting an invalid segment descriptor (V=0 for the segment descriptor) in two ways: first, as an indication that the segment cannot be accessed by the program, and second, as an indication that the segment is temporarily inaccessible because its subordinate page descriptor table is not currently resident in physical memory. If a hardware table search operation encounters an invalid segment descriptor, then the instruction or data access exception occurs. If the exception handler software determines that the faulted access lies within a segment that should be accessible, it can load the corresponding page descriptor table into physical memory, load the base address of the page descriptor table into the segment descriptor, mark the segment descriptor as valid (V=1), and retry the original access.

8

8.6 DATA BREAKPOINTS

The MC88110 DMMU contains two data breakpoint registers that can be used to transfer program control to a debugger program when accesses are made to specified logical addresses. If data breakpoints are enabled, the DMMU compares logical addresses of accesses to logical breakpoint addresses in the data breakpoint registers. When a comparison results in a match, the DMMU causes the Data Access exception to occur, and sets the BPE bit in the DSR.

Figure 8-19 shows the algorithm used by the DMMU to check for data breakpoints.
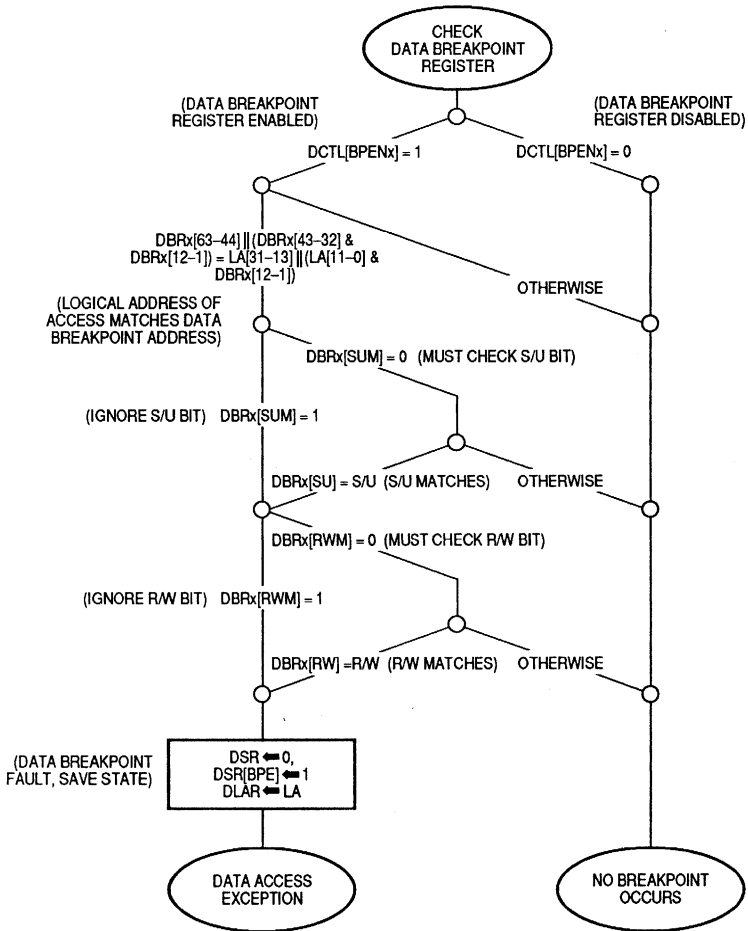
Figure 8-19. Data Breakpoint Algorithm

8.6.1 Data Breakpoint Descriptors

Data breakpoint registers contain entries that describe data breakpoints. The format of a data breakpoint descriptor is shown in Figure 8-20.
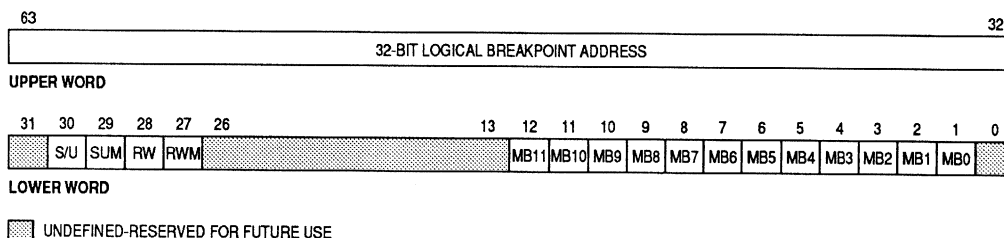
Figure 8-20. Data Breakpoint Descriptor Format

LBA—Logical Breakpoint Address

This field contains the 32-bit logical address of the data breakpoint.

S/U—Supervisor/User

This bit determines whether the breakpoint descriptor should be used to compare with user or supervisor accesses.

    1—Only supervisor logical addresses are compared
    0—Only user logical addresses are compared

SUM—Supervisor/User Mask

This bit is used to enable masking of the S/U bit.

    1—LBA resides in either the supervisor or user logical address map
    0—LBA resides in the address map specified by the S/U bit

R/W—Read/Write

This bit determines whether the breakpoint descriptor should be used to compare with read or write accesses.

    1—Only read accesses are compared
    0—Only write accesses are compared

RWM—Read/Write Mask

This bit is used to enable masking of the R/W bit.

    1—Both read and write accesses are compared
    0—Data breakpoint is compared for read or write accesses as specified by the R/W bit

8

MB11–MB0—Address Mask Bits

The MB11-MB0 bits can be used to specify the size of logical address comparisons that cause data breakpoints to occur. These bits determine the bits of the LBA field that are ignored for breakpoint address comparisons as described in Table 8-13.

1—Ignore corresponding LBA bit in data breakpoint address comparisons
0—Use corresponding LBA bit in data breakpoint address comparisons

Table 8-13. Example Address Mask Bits and Corresponding LBA Bits

MB11–MB0	Compare Address Bits	Address Size
000000000000	31–0	Byte Addresses
000000000001	31–1	Half-Word Addresses
000000000011	31–2	Word Addresses
000000000111	31–3	Double Word Addresses
000000001111	31–4	Quad Word Addresses
111111111111	31–12	Page Addresses

8.6.2 Enabling Data Breakpoints

Each data breakpoint register is enabled separately by setting BPEN0 or BPEN1 in the DCTL. It is not necessary to enable DMMU address translations in order to enable the operation of data breakpoints.

8.6.3 Loading Data Breakpoint Registers

Data breakpoint registers 0 and 1 are accessed as entries 32 and 33, respectively, in the DMMU PATC. Data breakpoint descriptors are loaded into the two data breakpoint registers in the same manner as software loads page descriptors into PATC entries. Refer to **8.4.4.2 Loading PATC Entries** for more detailed information on loading PATC entries and **8.9 MMU/Cache Control Registers** for a complete description of the data breakpoint registers.

To load the data breakpoint registers, the following steps should be performed sequentially:

1. Save the original contents of the DIR.
2. Store the number 32 or 33 into the PATC/breakpoint index field of the DIR, depending on which data breakpoint register is to be modified.
3. Store the upper word of the data breakpoint descriptor into the DPPU.
4. Store the lower word of the data breakpoint descriptor into the DPPL.
5. Restore the original contents of DIR.

8.6.4 Reading Data Breakpoint Registers

Data breakpoint descriptors are read from the two data breakpoint registers in the same manner as page descriptors are read from PATC entries. The data breakpoint registers are accessed as entries 32 and 33, respectively, in the DMMU PATC. Refer to **8.4.4.3 Reading PATC Entries** for more detailed information on reading PATC entries and **8.9 MMU/Cache Control Registers** for a complete description of the data breakpoint registers.

To read the data breakpoint registers, the following steps must be performed sequentially:

1. Save the original contents of the DIR.
2. Store the number 32 or 33 into the PATC/breakpoint index field of the DIR, depending on which data breakpoint register must be read.
3. Load the lower word of the data breakpoint descriptor from the DPPL.
4. Load the upper word of the data breakpoint descriptor from the DPPU.
5. Restore the original contents of DIR.

8.6.5 Data Breakpoint Fault

If data breakpoints are enabled and a data access matches a data breakpoint described by either of the data breakpoint registers, the DMMU signals the data breakpoint fault by setting the BPE bit in the DSR and causing the data access exception to occur. The logical address of the data access is automatically saved in the DLAR. The contents of the DPAR are undefined after this fault occurs. Data breakpoint faults do not occur in the IMMU.

8

8.7 MMU/CACHE FAULTS

Table 8-14 provides a complete listing of the state information saved for all of the MMU/cache faults that cause the instruction or data access exception to occur.

Table 8-14. Saved State For All MMU/Cache Faults

Fault	Status Register Bit (ISR, DSR)	ILAR/DLAR	IPAR/DPAR
Table Search Bus Error	TBE = 1	Logical Address of Initial Instruction or Data Access	Physical Address of Faulted Bus Cycle
Segment Descriptor Invalid	SI = 1	Logical Address of Initial Instruction or Data Access	Physical Address of Invalid Segment Descriptor
Page Descriptor Invalid	PI = 1	Logical Address of Initial Instruction or Data Access	Physical Address of Invalid Page Descriptor
Supervisor Protection Violation	SP = 1	Logical Address of Initial Instruction or Data Access	Physical Address of Violation Segment or Page Descriptor
Write Protect Violation	WE = 1 (DSR only)	Logical Address of Initial Data Write Access	DPAR and IPAR Undefined
Data Breakpoint	BPE = 1 (DSR only)	Logical Address of Initial Data Access	DPAR and IPAR Undefined
Copyback Error	CP = 1 (DSR only)	Logical Address of Initial Data Access That Missed in Cache	Physical Address of Faulted Bus Cycle (DPAR); Bits 4–0 Undefined.
Write-Allocate Error	WA = 1 (DSR only)	Logical Address of Initial Data Write Access	Physical Address of Faulted Bus Cycle (DPAR); Bits 4–0 Undefined
Bus Error	BE = 1	Logical Address of Initial Instruction or Data Access	Physical Address of Faulted Bus Cycle

Refer to **8.5.3.1 Table Search Faults** for detailed information about the saved state and conditions that cause the first five faults listed in Table 8-14. Refer to **8.6.5 Data Breakpoint Fault** for detailed information about the saved state and the conditions that cause the data breakpoint fault to occur. Refer to **8.9 MMU/Cache Control Registers** for a detailed description of the bits in the MMU/cache control registers.

The following paragraphs describe the last three faults listed in Table 8-14 and the conditions that cause them to occur. These faults are detected by the caches of the instruction and data memory units or the BIU in processing instruction and data accesses. The cache-detected faults cause either the instruction or data access exception to occur and the saved state resides in the ISR/DSR, ILAR/DLAR, and IPAR/DPAR registers.

8.7.1 Copyback Error

If external memory returns a bus error in response to a copyback operation before the MC88110 attempts an external memory access to satisfy a data cache miss, the data cache causes state information to be saved for the copyback error fault by setting the CP bit in the DSR and causes the data access exception to occur. The logical address of the initial data access that missed in the data cache is automatically saved in the DLAR. The physical address of the copyback operation that was faulted by the external memory system is automatically saved in the DPAR. Bits 4–0 of the DPAR are undefined when this fault occurs. Copyback error faults do not occur in the IMU. Refer to **Section 6 Instruction and Data Caches** for more information on data cache copyback operations.

8.7.2 Write-Allocate Error

If the write-allocate policy is selected (WT=0 and CI=0 in the ATC entry mapping the access) and a read from main memory to satisfy a write miss in the data cache results in a fault, the data cache causes state information to be saved for the cache write-allocate bus error fault by setting the WA bit in the DSR and causing the data access exception to occur. The logical address of the initial data write access is automatically saved in the DLAR. The physical address that was driven onto the external bus for the faulted bus cycle is automatically saved in the DPAR. Cache write-allocate bus error faults do not occur in the IMU. Refer to **Section 6 Instruction and Data Caches** for more information on the write-allocate data cache policy.

Write-allocate errors can occur in the case of uncorrectable memory errors.

8.7.3 Bus Error

If an access to external memory results in a bus error, the BIU signals the bus error fault by setting the BE bit in the ISR or DSR. The logical address of the initial instruction fetch or data access is saved automatically in the ILAR or DLAR. The physical address that was driven onto the external bus for the faulted bus cycle is automatically saved in the IPAR or DPAR.

8.8 ATC PROBE CAPABILITY

ATC probe commands in the MC88110 allow operating system software to determine whether a descriptor for a specific logical address is present within the ATC. If so, the MC88110 returns the index for the correct ATC entry. This simplifies the steps required if the operating system modifies a descriptor in the descriptor tables in main memory and must make the same changes to the image of the descriptor on-chip.

For example, if a page frame is re-allocated for a different logical page, the operating system must mark its page descriptor as invalid in the page descriptor tables. The ATC probe command can determine if the page descriptor is resident in the PATC. If so, the MC88110 locates which PATC entry contains the descriptor, so that the operating system can also mark the page descriptor as invalid within the PATC.

The ATC probe commands may be used even if the MMU is currently disabled (MEN = 0 in ICTL/DCTL). ATC probe commands in the MC88110 never cause hardware table search operations to occur, even if hardware table searching is enabled.

8.8.1 ATC Probe Commands

The ATC probe commands occur when the system software performs the following steps sequentially:

1. Store the logical address of interest into the ISAR or DSAR.
2. Store the command code value for MMU probe supervisor or MMU probe user (depending on whether the specified logical address is within the supervisor or user logical address map) into the command code field of the ICMD or DCMD.

The ATC probe command codes are listed in Table 8-15.

Table 8-15. ATC Probe Command Codes

Code	Command
1000	MMU Probe Supervisor (see Note)
1001	MMU Probe User (see Note)

NOTE: The logical address probed by the MMU probe supervisor or MMU probe user command is specified in the ISAR/DSAR.

8.8.2 ATC Probe Results

After an ATC probe command is invoked, the MMU compares the logical address specified with the logical addresses of all valid ATC entries as shown in Figure 8-21.
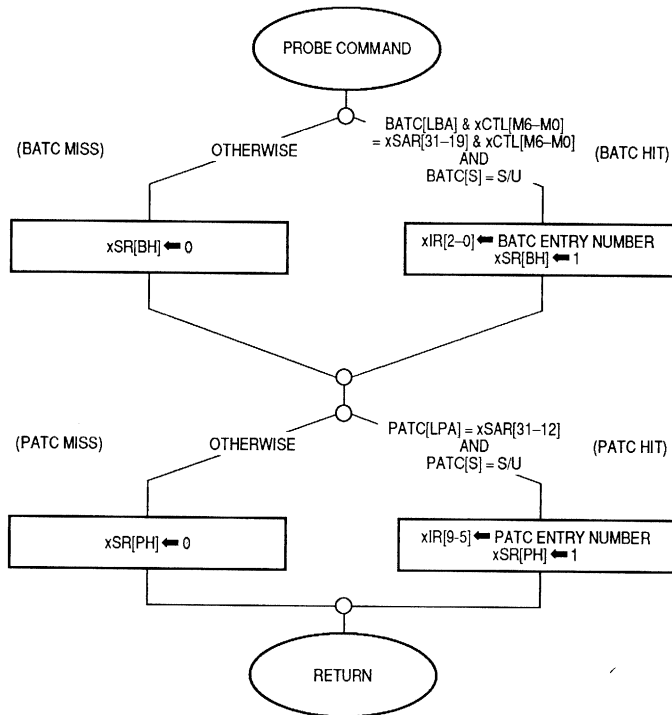
Figure 8-21. ATC Probe Algorithm

If a comparison results in a match in the BATC, the MMU sets the BH bit in the ISR or DSR and loads the index of the entry into the BATC index field of the IIR or DIR. Similarly, if a comparison results in a match in the PATC, the MMU sets the PH bit in the ISR or DSR and loads the index of the entry into the PATC index field of the IIR or the PATC/breakpoint index field of the DIR.

The ATC probe commands always check both the BATC and PATC, so it is possible for both PH and BH to be set and both ATC index fields to be updated following a probe operation. The ATC entry may then be accessed as described in **8.3.5 Block Descriptor Maintenance** or **8.4.4 Software Maintenance of PATC Entries**.

If the comparisons fail to find an ATC entry matching the logical address, then the MMU clears both the BH and PH bits in the ISR or DSR, and the IIR and DIR index fields are undefined.

8.9 MMU/CACHE CONTROL REGISTERS

The following paragraphs describe the control registers within the IMU and DMU. All of these registers are read/write registers within the general control register file, and access to all of these control registers is privileged. Note that these registers can only be accessed by the **ldcr** and **stcr** instructions and not by the **xcr** instruction.

8.9.1 Instruction MMU/Cache Registers

The following paragraphs describe the general control registers which permit supervisor mode software to control the operation of the IMU.

8.9.1.1 INSTRUCTION MMU/CACHE/TIC COMMAND REGISTER (ICMD). The ICMD, **cr25**, permits the system software to issue commands to invalidate IMMU PATC entries, lines in the instruction cache and the TIC, and to probe the IMMU. Writing to the 4-bit command code field in the ICMD with the **stcr** instruction initiates the requested command. Figure 8-22 illustrates the format of the ICMD. Table 8-16 lists the command codes defined for the ICMD. Reading the ICMD always returns all ones.

```
         31                                              4  3      0
cr25    [                                              ][COMMAND CODE]

  [::] UNDEFINED-RESERVED FOR FUTURE USE
```
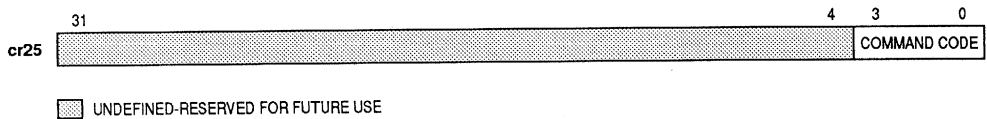
Figure 8-22. ICMD Format

Table 8-16. ICMD Command Codes

Code	Command
0000	Reserved
0001	Invalidate Instruction Cache and TIC
0010	Invalidate TIC
0011	Reserved
0100	Reserved
0101	Invalidate Instruction Cache Line (see Note 1)
0110	Reserved
0111	Reserved
1000	MMU Probe Supervisor (see Note 2)
1001	MMU Probe User (see Note 2)
1010	Invalidate All Supervisor PATC Entries
1011	Invalidate All User PATC Entries
11xx	Reserved

NOTES:
1. The physical address of the cache line affected by the invalidate instruction cache line command is specified in the ISAR.
2. The logical address probed by the MMU probe supervisor or MMU probe user command is specified in the ISAR.

8.9.1.2 INSTRUCTION MMU/CACHE CONTROL REGISTER (ICTL).

The ICTL, **cr26**, selects the different possible operating modes of the instruction cache, TIC, and the IMMU. Figure 8-23 illustrates the format of the ICTL. The default state after reset is denoted in the following paragraphs with an asterisk (*).
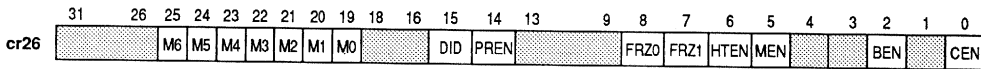
UNDEFINED-RESERVED FOR FUTURE USE

Figure 8-23. ICTL Format

M6–M0—IMMU BATC Block Size Selection Bits

The block sizes mapped by the BATC can be programmed by setting bits M6–M0 according to Table 8-17. Note that this table is the same as Table 8-8. After a processor reset, the selected block size is undefined.

Table 8-17. IMMU BATC Block Size Selection Settings

Block Size Mask Bits							Block Size
M6	M5	M4	M3	M2	M1	M0	
1	1	1	1	1	1	1	64M-byte
0	1	1	1	1	1	1	32M-byte
0	0	1	1	1	1	1	16M-byte
0	0	0	1	1	1	1	8M-byte
0	0	0	0	1	1	1	4M-byte
0	0	0	0	0	1	1	2M-byte
0	0	0	0	0	0	1	1M-byte
0	0	0	0	0	0	0	512K-byte
Any Other Combination							Undefined

DID—Double Issue Disable

When double issue mode is enabled, the instruction unit attempts to issue two instructions in each clock cycle. When double issue is disabled, the instruction unit attempts to issue only one instruction per clock.

0—Double instruction issue enabled*
1—Double instruction issue disabled

PREN—Branch Prediction Enable

When branch prediction is disabled, the branch reservation station is disabled. In this case, if a branch instruction with a data dependency is encountered, instruction issue stalls until the data dependency is resolved. When branch prediction is enabled, branches with data dependencies issue to the branch reservation station, and conditional instruction issue occurs in the predicted direction.

0—Branch prediction disabled*
1—Branch prediction enabled

FRZ0—Instruction Cache Freeze Bank 0 Enable

When instruction cache freeze bank 0 is enabled, the first line (line 0) in each set in the instruction cache is frozen.

0—Instruction cache freeze bank 0 disabled*
1—Instruction cache freeze bank 0 enabled

FRZ1—Instruction Cache Freeze Bank 1 Enable

When instruction cache freeze bank 1 is enabled, the first line (line 1) in each set in the instruction cache is frozen.

0—Instruction cache freeze bank 1 disabled*
1—Instruction cache freeze bank 1 enabled

HTEN—IMMU Hardware Table Search Enable

When hardware table search operations are enabled, a hardware table search operation is performed when a PATC miss occurs. When software table search operations are selected, the IMMU PATC miss exception occurs on an PATC miss, and no hardware table search operation occurs.

    0—IMMU hardware table search operation is disabled; software table search operations are selected

    1—IMMU hardware table search operation is enabled*

MEN—IMMU Enable

When the IMMU is enabled, address translations can occur via the BATC or PATC. If the IMMU is disabled, then the logical address for each memory location is the same as the physical address (identity translation), and the access control information (e.g., memory update mode, global/local page designations, etc.) is taken from the ISAP or IUAP.

    0—Instruction MMU disabled*

    1—Instruction MMU enabled

BEN—TIC Enable

When the TIC is disabled, no instructions are fetched from the TIC and the TIC is not accessed or updated.

    0—TIC disabled*

    1—TIC enabled

CEN—Instruction Cache Enable

When the instruction cache is disabled, all instruction fetches pass directly to the BIU and the instruction cache is not accessed or updated.

    0—Instruction cache disabled*

    1—Instruction cache enabled

8.9.1.3 INSTRUCTION SYSTEM ADDRESS REGISTER (ISAR). The ISAR, **cr27**, indicates the logical address for an ATC probe command or the physical address of an instruction cache line to be invalidated during a line invalidate operation. The ISAR must be written before the ICTL for correct operation of these commands. Figure 8-24 shows the format of the ISAR.

31	0
cr27	LOGICAL ADDRESS FOR ATC PROBE OR PHYSICAL ADDRESS OF CACHE LINE
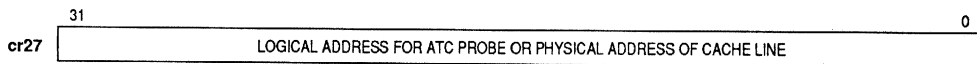
Figure 8-24. ISAR Format

8.9.1.4 IMMU SUPERVISOR AREA POINTER REGISTER (ISAP). The ISAP, **cr28**, contains the currently active area descriptor for supervisor logical instruction addresses. Figure 8-25 illustrates the contents of the ISAP register. For the complete format of an area descriptor, refer to **8.5.2.1 Area Descriptor Format**.

```
      31                                                                          0
cr28 |              INSTRUCTION SUPERVISOR AREA DESCRIPTOR                        |
```
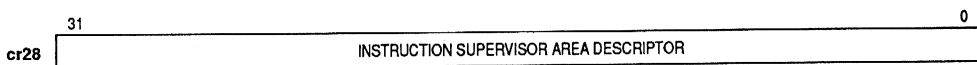
Figure 8-25. ISAP Format

8.9.1.5 IMMU USER AREA POINTER REGISTER (IUAP). The IUAP, **cr29**, contains the currently active area descriptor for user logical instruction addresses. Figure 8-26 illustrates the contents of the IUAP register. For the complete format of an area descriptor, refer to **8.5.2.1 Area Descriptor Format**.

```
      31                                                                          0
cr29 |                INSTRUCTION USER AREA DESCRIPTOR                            |
```
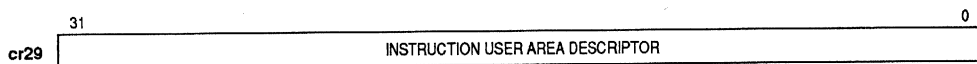
Figure 8-26. IUAP Format

8.9.1.6 IMMU ATC INDEX REGISTER (IIR). The IIR, **cr30**, is a read/write control register. It is used to specify the entry number of BATC and PATC entries to be written into or read out of the IMMU ATCs through the IBP, IPPU, and IPPL registers. It is also used to read and write the user attribute bits in BATC entries. Figure 8-27 illustrates the format of the IIR.

```
      31                            16 15 14 13    10 9        5 4 3 2       0
cr30 |////////////////////////////////| U1 | U0 |//////| PATC INDEX |////| BATC INDEX |
```

▒ UNDEFINED-RESERVED FOR FUTURE USE
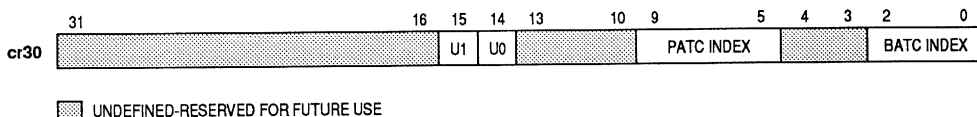
Figure 8-27. IIR Format

U0—User Attribute 0
  The value of U0 in the BATC entry specified by the BATC index field.

U1—User Attribute 1
  The value of U1 in the BATC entry specified by the BATC index field.

PATC Index
  The number (0–31) of the PATC entry accessible through the IPPU and IPPL.

BATC Index
  The number (0–7) of the PATC entry accessible through the IBP.

8.9.1.7 IMMU BATC R/W PORT REGISTER (IBP). The IBP, **cr31**, permits read/write accesses to the instruction BATC entry selected via the IIR. When the IBP is written, the block descriptor (including the user attribute bits in the IIR) is stored into the instruction BATC. When the IBP is read, the block descriptor is read out of the instruction BATC into the IIR and IBP. Figure 8-28 illustrates the contents of the IBP. Refer to **8.3.3 BATC Descriptor Format** for the format of a block descriptor.

```
      31                                                                          0
      ┌─────────────────────────────────────────────────────────────────────────┐
cr31  │                      INSTRUCTION BLOCK DESCRIPTOR                         │
      └─────────────────────────────────────────────────────────────────────────┘
```
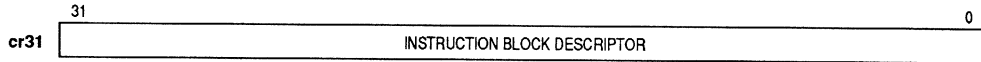
Figure 8-28. IBP Format

8.9.1.8 IMMU PATC R/W PORT UPPER REGISTER (IPPU). The IPPU, **cr32**, permits read/write accesses to the upper 32 bits of the instruction PATC entry selected via the IIR. When the IPPU is written, the upper word of the page descriptor is buffered until the IPPL is written. When the IPPU is read, the page descriptor is read out of the instruction PATC into the IPPU and IPPL. Figure 8-29 illustrates the contents of the IPPU. The format of a PATC entry is described in **8.4.3 PATC Descriptor Format**.

```
      31                                                                          0
      ┌─────────────────────────────────────────────────────────────────────────┐
cr32  │                      UPPER WORD OF PAGE DESCRIPTOR                        │
      └─────────────────────────────────────────────────────────────────────────┘
```
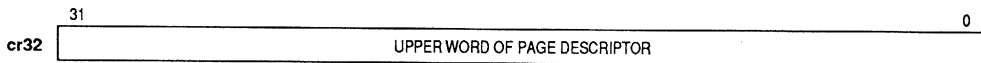
Figure 8-29. IPPU Format

8.9.1.9 IMMU PATC R/W PORT LOWER REGISTER (IPPL). The IPPL, **cr33**, permits read/write accesses to the lower 32 bits of the instruction PATC entry selected via the IIR. When the IPPL is written, it and the upper word of the page descriptor buffered in the IPPU are written into the instruction PATC. When the IPPL is read, the lower 32 bits of the page descriptor buffered by the last read of the IPPU are received. Figure 8-30 illustrates the contents of the IPPL. The format of a PATC entry is described in **8.4.3 PATC Descriptor Format**.

```
      31                                                                          0
      ┌─────────────────────────────────────────────────────────────────────────┐
cr33  │                      LOWER WORD OF PAGE DESCRIPTOR                        │
      └─────────────────────────────────────────────────────────────────────────┘
```
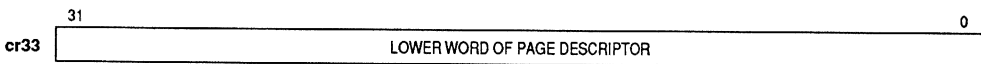
Figure 8-30. IPPL Format

8.9.1.10 INSTRUCTION ACCESS STATUS REGISTER (ISR). The IMMU loads the ISR, **cr34**, with state information for instruction access exceptions or IMMU ATC probe commands. This register is not updated while EFRZ=1 in the PSR (**cr1**). All bits in the ISR are undefined after a processor reset. Figure 8-31 illustrates the format of the ISR. See **Section 2 Programming Model** for a detailed description of the PSR. Refer to **8.7 MMU/Cache Faults** for more information on specific Instruction Access exceptions, and refer to **8.8 ATC Probe Capability** for more information about probe commands.

	31	22	21	20	19	18	17	12	11	10	9	8	1	0
cr34			TBE	SI	PI	SP			PH	BH	S/U			BE

▒ UNDEFINED-RESERVED FOR FUTURE USE
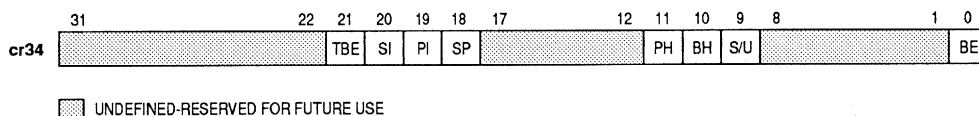
Figure 8-31. ISR Format

TBE—Table Search Bus Error

The MC88110 sets this bit if a bus cycle to external memory results in a bus error during a hardware table search operation.

0—Bus error did not occur during a hardware table search operation
1—Bus error occurred during a hardware table search operation

SI—Segment Descriptor Invalid

The IMMU sets this bit if a hardware table search operation fetches an invalid segment descriptor.

0—Hardware table search operation did not fetch an invalid segment descriptor
1—Hardware table search operation fetched an invalid segment descriptor

PI—Page Descriptor Invalid

The IMMU sets this bit if a hardware table search operation fetches an invalid page, second indirection, or masked protection indirection descriptor.

0—Hardware table search operation did not fetch an invalid page descriptor
1—Hardware table search operation fetched an invalid page descriptor

SP—Supervisor Protection Violation

The IMMU sets this bit if a hardware table search operation for a user logical address fetches a segment or page descriptor with the supervisor protection bit set.

0—Supervisor protected descriptor not fetched
1—Supervisor protected descriptor fetched

PH—PATC Hit

This bit is updated by instruction ATC probe commands to indicate whether the probed logical address is described by the PATC.

0—Probe command did not hit in the PATC
1—Probe command hit in the PATC

BH—BATC Hit

This bit is updated by instruction ATC probe commands to indicate whether the probed logical address is described by the BATC.

0—Probe command did not hit in the BATC
1—Probe command hit in the BATC

S/U—Supervisor/User Address

This bit indicates if the logical address saved by the MC88110 in the ILAR is a user or supervisor logical address.

0—Address is a user logical address
1—Address is a supervisor logical address

BE—Bus Error

The MC88110 sets this bit if a bus cycle to external memory is faulted during an instruction fetch.

0—Bus error has not occurred
1—Bus error has occurred

8.9.1.11 INSTRUCTION ACCESS LOGICAL ADDRESS REGISTER (ILAR).

For instruction access and instruction PATC miss exception conditions, the IMMU loads the upper 27 bits (the lower order 5 bits are undefined) of the logical address of the failed access into the ILAR, **cr35**. The supervisor/user mode bit for the access is located in the ISR. This register is not updated while EFRZ=1 in the PSR (**cr1**). See **Section 2 Programming Model** for a detailed description of the PSR. Figure 8-32 illustrates the format of the ILAR.

```
      31                                                                          0
cr35 |             LOGICAL ADDRESS OF FAILED ACCESS                               |
```
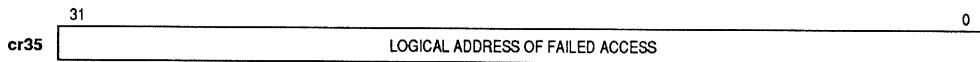
Figure 8-32. ILAR Format

8.9.1.12 INSTRUCTION ACCESS PHYSICAL ADDRESS REGISTER (IPAR).

For instruction access exceptions, the IMMU loads a physical address related to the exception into the IPAR, **cr36**. Figure 8-33 illustrates the format of the IPAR.

Table 8-18 summarizes the contents of the IPAR for the different types of Instruction Access exceptions. This register is not updated while EFRZ=1 in the PSR (**cr1**). See **Section 2 Programming Model** for a detailed description of the PSR.

```
      31                                                                          0
cr36 |                 PHYSICAL ADDRESS FOR FAULT                                 |
```
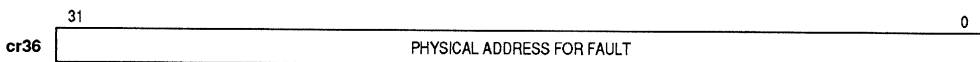
Figure 8-33 IPAR Format

Table 8-18. IPAR Contents for MMU/Cache Faults

Fault That Caused Instruction Access Exception	IPAR Contents
Table Search Bus Error	Physical Address of Faulted Bus Cycle
Segment Descriptor Invalid	Physical Address of Invalid Segment Descriptor
Page Descriptor Invalid	Physical Address of Invalid Page Descriptor
Supervisor Protection Violation	Address of Violation Segment or Page Descriptor (with SP=1)
Bus Error	Address of Faulted Bus Cycle

8.9.2 Data MMU/Cache Registers

The following paragraphs describe the general control registers which permit supervisor mode software to control the operation of the DMU.

8.9.2.1 DATA MMU/CACHE COMMAND REGISTER (DCMD). The DCMD, **cr40**, is used to invalidate DMMU PATC entries and lines in the data cache. In addition, it provides commands that copyback dirty lines in the data cache and probe the DMMU. Writing to the 4-bit command code field in the DCMD with the **stcr** instruction initiates the requested command. Figure 8-34 illustrates the format of the DCMD. Table 8-19 lists the command codes defined for the DCMD. Reading the DCMD always returns all ones.

```
       31                                                    4  3        0
cr40  |                                                      | COMMAND CODE |
```

▒ UNDEFINED-RESERVED FOR FUTURE USE
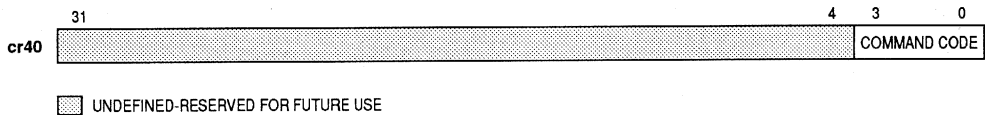
Figure 8-34. DCMD Format

Table 8-19. DCMD Command Codes

Code	Command
0000	Flush Data Cache Page (Copyback Operation) (see Note 1)
0001	Invalidate Entire Data Cache
0010	Flush Entire Data Cache (Copyback Operation Only)
0011	Flush and Invalidate Entire Data Cache (Copyback and Invalidate Operation)
0100	Flush Data Cache Page (Copyback and Invalidate Operation) (see Note 1)
0101	Invalidate Data Cache Line (see Note 1)
0110	Flush Data Cache Line (Copyback Operation Only) (see Note 1)
0111	Flush Data Cache Line (Copyback and Invalidate Operation) (see Note 1)
1000	MMU Probe Supervisor (see Note 2)
1001	MMU Probe User (see Note 2)
1010	Invalidate All Supervisor PATC Entries
1011	Invalidate All User PATC Entries
11xx	Reserved

NOTES:
1. The physical address of the cache line affected by the invalidate data cache line command is specified in the DSAR.
2. The logical address probed by the MMU probe supervisor or MMU probe user command is specified in the DSAR.

8.9.2.2 DATA MMU/CACHE CONTROL REGISTER (DCTL). The DCTL, **cr41**, selects the different possible operating modes of the data cache and the DMMU. Figure 8-35 illustrates the format of the DCTL. In the following paragraphs, the default state after reset is indicated by an asterisk (*).

UNDEFINED-RESERVED FOR FUTURE USE
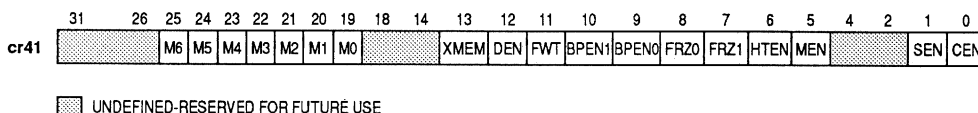
Figure 8-35. DCTL Format

M6–M0—DMMU BATC Block Size Selection Bits

The block sizes mapped by the BATC can be programmed by setting bits M6–M0 according to Table 8-20. Note that this table is the same as Table 8-8. After a processor reset, the selected block size is undefined.

Table 8-20. DMMU BATC Block Size Selection Settings

Block Size Mask Bits							Block Size
M6	M5	M4	M3	M2	M1	M0	
1	1	1	1	1	1	1	64M-byte
0	1	1	1	1	1	1	32M-byte
0	0	1	1	1	1	1	16M-byte
0	0	0	1	1	1	1	8M-byte
0	0	0	0	1	1	1	4M-byte
0	0	0	0	0	1	1	2M-byte
0	0	0	0	0	0	1	1M-byte
0	0	0	0	0	0	0	512K-byte
Any Other Combination							Undefined

XMEM—**xmem** Instruction Control Bit

When this bit is cleared, the MC88110 **xmem** instruction performs a locked bus sequence consisting of a load followed by a store operation. When this bit is set, the **xmem** instruction performs a locked bus sequence consisting of a store followed by a load operation (see **Section 11 System Hardware Design**).

   0—**xmem** causes load followed by store locked bus sequence*
   1—**xmem** causes store followed by load locked bus sequence

DEN—Decoupled Cache Access Enable

When this bit is clear, decoupled accesses to the data cache are disabled, regardless of the type of bus transaction in progress or the status of the PTA input signal. When this bit is set, decoupled accesses are allowed under the control of the PTA signal (see **Section 11 System Hardware Design**).

   0—Decoupled cache accesses disabled*
   1—Decoupled cache accesses enabled

FWT—Force Write-Through

When this bit is set, all store operations are forced to write through the data cache, regardless of the page or block status; however, the FWT bit does not have any affect on the operation of the WT signal.

   0—Write-through vs. write-back mode selected by page or block descriptors*
   1—Force write-through mode for write accesses

BPEN1—Data Breakpoint Register 1 Enable

When data breakpoint register 1 is disabled, a data access exception does not occur when a matching logical address is detected. When data breakpoint register 1 is enabled, it causes a data access exception upon detecting a matching logical address.

   0—Data breakpoint register 1 disabled*
   1—Data breakpoint register 1 enabled

BPEN0—Data Breakpoint Register 0 Enable

When data breakpoint register 0 is disabled, a data access exception does not occur when a matching logical address is detected. When data breakpoint register 0 is enabled, it causes a data access exception upon detecting a matching logical address.

    0—Data breakpoint register 0 disabled*
    1—Data breakpoint register 0 enabled

FRZ0—Data Cache Freeze Bank 0 Enable

When data cache freeze bank 0 is enabled, the first line (line 0) in each set in the data cache is frozen.

    0—Data cache freeze bank 0 disabled*
    1—Data cache freeze bank 0 enabled

FRZ1—Data Cache Freeze Bank 1 Enable

When data cache freeze bank 1 is enabled, the first line (line 1) in each set in the data cache is frozen.

    0—Data cache freeze bank 1 disabled*
    1—Data cache freeze bank 1 enabled

HTEN—DMMU Hardware Table Search Enable

When hardware table search operations are enabled, a hardware table search operation is performed when a PATC miss occurs. When software table search operations are selected, the DMMU PATC read or write miss exception occurs on a PATC miss, and no hardware table search operation occurs.

    0—DMMU hardware table search operation is disabled; software table search
      operations are selected
    1—DMMU hardware table search operation is enabled*

MEN—DMMU Enable

When the DMMU is enabled, address translations can occur via the BATC or PATC. If the DMMU is disabled, then the physical address for each memory location is the same as the logical address (identity translation), and the access control information (e.g., memory update mode, global/local designations, etc.) is taken from the DSAP or DUAP.

    0—Data MMU disabled*
    1—Data MMU enabled

SEN—Data Cache Snooping Enable

When data cache snooping is enabled, the BIU will monitor external global accesses to ensure that all local copies of data are consistent.

    0—Data cache snooping disabled*
    1—Data cache snooping enabled

8

CEN—Data Cache Enable

When the data cache is disabled, load and store operations pass directly to the BIU and the data cache is not accessed or updated.

0—Data cache disabled*
1—Data cache enabled

8.9.2.3 DATA SYSTEM ADDRESS REGISTER (DSAR). The DSAR, **cr42**, is used to indicate the logical address for an ATC probe or the physical address of a data cache line or page to be invalidated and/or copied back during an invalidate, copyback, or copyback and invalidate operation. The DSAR must be written before the DCTL for correct operation of these commands. Figure 8-36 shows the format of the DSAR.

```
        31                                                                      0
cr42  |           LOGICAL ADDRESS FOR ATC PROBE OR PHYSICAL ADDRESS OF CACHE LINE      |
```
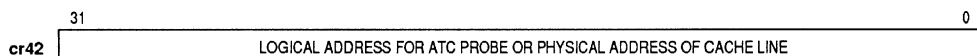
Figure 8-36. DSAR Format

8.9.2.4 DMMU SUPERVISOR AREA POINTER REGISTER (DSAP). The DSAP, **cr43**, contains the currently active area descriptor for supervisor logical data addresses. Figure 8-37 illustrates the contents of the DSAP register. For the complete format of an area descriptor, refer to **8.5.2.1 Area Descriptor Format.**

```
        31                                                                      0
cr43  |                        DATA SUPERVISOR AREA DESCRIPTOR                          |
```

Figure 8-37. DSAP Format

8.9.2.5 DMMU USER AREA POINTER REGISTER (DUAP). The DUAP, **cr44**, contains the currently active area descriptor for user logical data addresses. Figure 8-38 illustrates the contents of the DUAP register. For the complete format of an area descriptor, refer to **8.5.2.1 Area Descriptor Format.**

```
        31                                                                      0
cr44  |                          DATA USER AREA DESCRIPTOR                              |
```
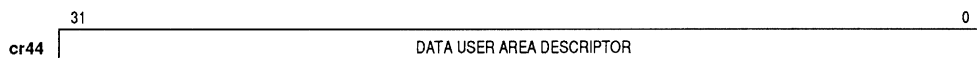
Figure 8-38. DUAP Format

8.9.2.6 DMMU ATC INDEX REGISTER (DIR). The DIR, **cr45**, is a read/write control register. It is used to specify the entry number of BATC and PATC entries to be written into or read out of the DMMU ATCs through the DBP, DPPU and DPPL registers. Data breakpoint register 0 is accessed as if it was PATC entry 32 and data breakpoint register 1 is accessed as if it was PATC entry 33. The DIR is also used to read and write the user attribute bits in BATC entries.

PATC entries 34–63 are unimplemented; attempts to access them may result in unexpected PATC behavior. Figure 8-39 illustrates the format of the DIR.

31	16	15	14	13	11	10	5	4	3	2	0
cr45 | | | U1 | U0 | | | PATC/BREAKPOINT INDEX | | | BATC INDEX | |

▒ UNDEFINED-RESERVED FOR FUTURE USE
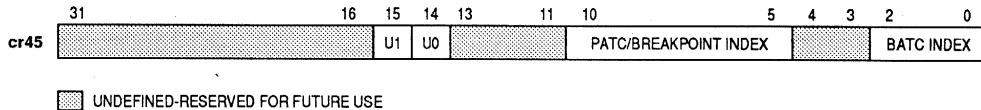
Figure 8-39. DIR Format

U0—User Attribute 0
   The value of U0 in the BATC entry specified by the BATC index field.

U1—User Attribute 1
   The value of U1 in the BATC entry specified by the BATC index field.

PATC/Breakpoint Index
   The number (0–33) of the PATC entry accessible through the DPPU and DPPL.

BATC Index
   The number (0–7) of the PATC entry accessible through the DBP.

8.9.2.7 DMMU BATC R/W PORT REGISTER (DBP). The DBP, **cr46**, permits read/write accesses to the data BATC entry selected via the DIR. When the DBP is written, the block descriptor (including user attribute bits defined in the DIR) is stored into the data BATC. When the DBP is read, the block descriptor is read out of the data BATC into the DIR and DBP. Figure 8-40 illustrates the contents of the DBP. Refer to **8.3.3 BATC Descriptor Format** for the format of a block descriptor.

31	0
cr46 | DATA BLOCK DESCRIPTOR |
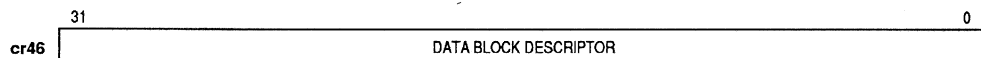
Figure 8-40. DBP Format

8.9.2.8 DMMU PATC R/W PORT UPPER REGISTER (DPPU). The DPPU, **cr47**, permits read/write accesses to the upper 32 bits of the data PATC entry selected via the DIR. When the DPPU is written, the upper word of the page descriptor is buffered until the DPPL is written. When the DPPU is read, the block descriptor is read out of the data PATC into the DPPU and DPPL. Figure 8-41 illustrates the contents of the DPPU. The format of a PATC entry is described in **8.4.3 PATC Descriptor Format**.

```
      31                                                                    0
cr47 |              UPPER WORD OF PAGE DESCRIPTOR                           |
```
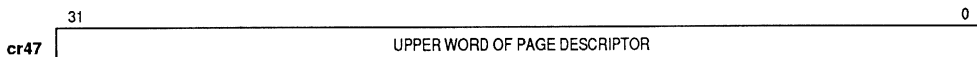
Figure 8-41. DPPU Format

8.9.2.9 DMMU PATC R/W PORT LOWER REGISTER (DPPL). The DPPL, **cr48**, permits read/write accesses to the lower 32 bits of the data PATC entry selected via the DIR. When the DPPL is written, it and the upper word of the page descriptor buffered in the DPPU are written into the data PATC. When the DPPL is read, the lower 32 bits of the page descriptor buffered by the last read of the DPPU are received. Figure 8-42 illustrates the contents of the DPPL. The format of a PATC entry is described in **8.4.3 PATC Descriptor Format**.

```
      31                                                                    0
cr48 |              LOWER WORD OF PAGE DESCRIPTOR                           |
```
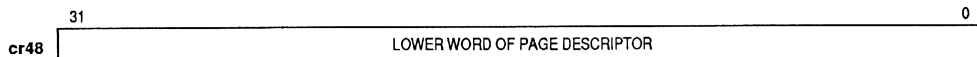
Figure 8-42. DPPL Format

8.9.2.10 DATA ACCESS STATUS REGISTER (DSR). The DMMU loads the DSR with state information for data access exceptions or DMMU ATC probe commands. This register is not updated while EFRZ=1 in the PSR (**cr1**). All bits in the ISR are undefined after a processor reset. Figure 8-43 illustrates the format of the ISR. See **Section 2 Programming Model** for a detailed description of the PSR. Refer to **8.7 MMU/Cache Faults** for more information on specific data access exceptions, and refer to **8.8 ATC Probe Capability** for more information about probe commands.

31		22	21	20	19	18	17	16	15	12	11	10	9	8	7		3	2	1	0
cr49			TBE	SI	PI	SP	WE	BPE			PH	BH	S/U	R/W				CP	WA	BE

☐ UNDEFINED-RESERVED FOR FUTURE USE
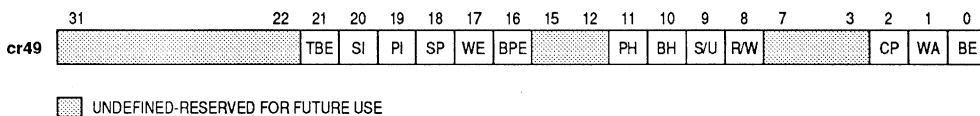
Figure 8-43. DSR Format

TBE—Table Search Bus Error

The MC88110 sets this bit if a bus error occurs for a bus cycle to external memory during a hardware table search operation.

    0—Bus error did not occur during a hardware table search operation
    1—Bus error occurred during a hardware table search operation

SI—Segment Descriptor Invalid

The DMMU sets this bit if a hardware table search operation fetches an invalid segment descriptor.

    0—Hardware table search operation did not fetch an invalid segment descriptor
    1—Hardware table search operation fetched an invalid segment descriptor

PI—Page Descriptor Invalid

The DMMU sets this bit if a hardware table search operation fetches an invalid page descriptor or second indirection descriptor.

    0—Hardware table search operation did not fetch an invalid page descriptor
    1—Hardware table search operation fetched an invalid page descriptor

SP—Supervisor Protection Violation

The DMMU sets this bit if hardware table search operation for a user logical address fetches a segment or page descriptor with the supervisor protection bit set.

    0—Supervisor protected descriptor not fetched
    1—Supervisor protected descriptor fetched

WE—Write Exception

The Data MMU sets this bit if a write access is attempted to write-protected page or a page whose modified bit is clear.

    0—No write fault occurred
    1—Write fault occurred

BPE—Breakpoint Exception

When the data access matches the logical address described by a data breakpoint register and data breakpoints are enabled, the DMU sets this bit.

    0—Data breakpoint not matched
    1—Data breakpoint matched

8

PH—PATC Hit

This bit is updated by data ATC probe commands to indicate whether the probed logical address is described by the PATC.

0—Probe command did not hit in the PATC
1—Probe command hit in the PATC

BH—BATC Hit

This bit is updated by data ATC probe commands to indicate whether the probed logical address is described by the BATC.

0—Probe command did not hit in the BATC
1—Probe command hit in the BATC

S/U—Supervisor/User Address

This bit indicates if the logical address saved by the MC88110 in the DLAR is a user or supervisor logical address.

0—Address is a user logical address
1—Address is a supervisor logical address

R/W—Read/Write

This bit indicates whether the faulted access was a read or write access

0—Data access was a write
1—Data access was a read

CP—Copyback Error

The MC88110 sets this bit if a bus error occurs for a burst write bus cycle to external memory during a copyback operation by the data cache. Refer to **Section 6 Instruction and Data Caches** for more information about data cache operations.

0—Bus error has not occurred
1—Bus error has occurred

WA—Write-Allocate Bus Error

The MC88110 sets this bit if a bus error occurs for a burst read bus cycle to external memory during a write-allocate operation by the data cache. Refer to **Section 6 Instruction and Data Caches** for more information about data cache operations.

0—Bus error has not occurred
1—Bus error has occurred

BE—Bus Error

The MC88110 sets this bit if a bus error occurs for a bus cycle to external memory during a data access.

0—Bus error has not occurred
1—Bus error has occurred

8.9.2.11 DATA ACCESS LOGICAL ADDRESS REGISTER (DLAR). For data access and data PATC miss exception conditions, the DMMU loads the logical address of the failed access into the DLAR, **cr50.** The supervisor/user mode bit for the access is located in the DSR. This register is not updated while EFRZ=1 in the PSR **(cr1).** See **Section 2 Programming Model** for a detailed description of the PSR. Figure 8-44 illustrates the format of the DLAR.

```
        31                                                                    0
cr50   ┌──────────────────────────────────────────────────────────────────────┐
       │               LOGICAL ADDRESS OF FAILED ACCESS                         │
       └──────────────────────────────────────────────────────────────────────┘
```
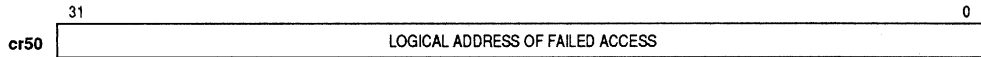
Figure 8-44. DLAR Format

8.9.2.12 DATA ACCESS PHYSICAL ADDRESS REGISTER (DPAR). For most data access exceptions, the DMMU loads a physical address related to the exception into the DPAR, **cr51.** Figure 8-45 illustrates the format of the DPAR.

Table 8-21 summarizes the contents of the DPAR for the different types of data access exceptions. This register is not updated while EFRZ=1 in the PSR **(cr1).** See **Section 2 Programming Model** for a detailed description of the PSR.

```
        31                                                                    0
cr51   ┌──────────────────────────────────────────────────────────────────────┐
       │                   PHYSICAL ADDRESS FOR FAULT                           │
       └──────────────────────────────────────────────────────────────────────┘
```
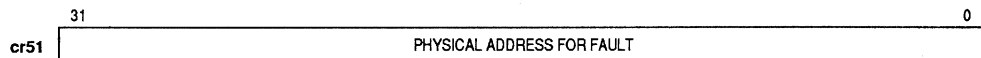
Figure 8-45. DPAR Format

Table 8-21. DPAR Contents for MMU/Cache Faults

Fault That Caused Data Access Exception	DPAR Contents
Table Search Bus Error	Physical Address of Faulted Bus Cycle
Segment Descriptor Invalid	Physical Address of Invalid Segment Descriptor
Page Descriptor Invalid	Physical Address of Invalid Page Descriptor
Supervisor Protection Violation	Physical Address of Violation Segment or Page Descriptor (SP=1)
Write Protect Violation	Undefined
Data Breakpoint	Undefined
Copyback Error	Physical Address of Faulted Bus Cycle
Write-Allocate Bus Error	Physical Address of Faulted Bus Cycle
Bus Error	Physical Address of Faulted Bus Cycle

8

8.10 MC88110 AND MC88200 MMU DIFFERENCES

Table 8-22 summarizes the differences between the MC88110 MMUs and the MMU in the MC88200 Cache/Memory Management Unit.

Table 8-22. MC88110 MMU and MC88200 MMU Differences

MC88110	MC88200
Hardware or Software Table Search	Hardware Table Search Only
32 PATC Entries, 8 BATC Entries	56 PATC Entries, 10 BATC Entries
BATC-Exclusive Address Translation Option	No BATC-Exclusive Address Translation Option
Area Descriptors Do Not Apply to Block Address Translation	Area Descriptors Apply to Block Address Translation
User Attributes in Area, Page, and Block Descriptors	No User Attributes
Indirection and Masked Protection Indirection Descriptors	No Indirection
Block Size 512K-byte to 64M-byte	Block Size 512K-byte
Software Sets Used and Modified Bits	Hardware Sets Used and Modified Bits
Write-Through Broadcast onto External Bus	Write-Through On-Chip Only
12 General Control Registers for Each MMU	26 Memory-Mapped I/O Registers
2 Data Breakpoint Registers	No Data Breakpoint Registers
Probe Command Searches On-Chip ATCs Only	Probe Command Searches On-Chip ATCs and Table Searches Page Descriptor Tables

8

SECTION 9
INSTRUCTION TIMING AND CODE SCHEDULING CONSIDERATIONS

This section describes instruction execution timings for the MC88110 microprocessor. In such a highly parallel machine, exact timing of all possible circumstances cannot be listed; therefore, instruction timings for example code sequences are presented as guidelines only. This guideline approach is used since exact instruction timing depends on variables such as memory speed and instruction sequencing.

Instruction prefetch and execution through all of the execution units of the MC88110 are described in detail. Examples of instruction sequences showing concurrent execution and various register dependencies are provided to illustrate timing interactions. Bus signals described in this section are only accurate to within one-half clock cycle increments. Refer to **Section 11 System Hardware Design** for more specific information regarding bus operation timing. Instruction mnemonics used in this section can be identified by referring to **Section 10 Instruction Set**.

9.1 INSTRUCTION TIMING OVERVIEW

The MC88110 has been designed to minimize average instruction execution latency. Instructions are implemented without micro-code, thus minimizing instruction decode and execution time.

Latency is defined as the number of clock cycles necessary to execute an instruction and make ready the results of that execution. For the majority of instructions in the MC88110, this can be simplified to include only the execute phase for a particular instruction. However, data instructions will require additional clock cycles between the execute phase and the write-back phase due to memory latencies.

In accordance with this definition, logical, bit-field, and most integer and graphics instructions have a latency of one clock cycle (e.g., results for these instructions are ready for use on the next clock cycle after issue). Other instructions, such as the integer multiply, require more than one clock period to complete execution. An example of the term "latency" is shown in Figure 9-1.

9

INSTRUCTION
STREAM

●
●
●

```
fmul.sss  r7, r7, r9
or        r10, r11, r12

add       r2, r3, r10
sub       r9, r4, r4
```

●
●
●

r10 IS READY FOR USE ON
THE CLOCK CYCLE AFTER
THE or INSTRUCTION ISSUES,
THUS THE or INSTRUCTION
IS SAID TO HAVE A LATENCY
OF 1 CLOCK CYCLE.

(a) Instruction Latency

INSTRUCTION
STREAM

●
●
●

```
cmp       r15, r17, r9
fmul.sss  r7, r7, r9

or        r10, r11, r12

add       r2, r3, r10
sub       r9, r4, r4
```

●
●
●

THE or AND THE add CANNOT
ISSUE ON THE SAME CLOCK CYCLE
BECAUSE THE add HAS A DATA
DEPENDENCY ON THE or. THUS,
A SINGLE BUBBLE IS INTRODUCED
INTO THE INSTRUCTION STREAM
AND THE or OPERATION IS SAID
TO HAVE A LATENCY OF 1 CLOCK
CYCLE.

(b) Instruction Latency

Figure 9-1. Instruction Latency

Notice that in Figure 9-1(a), **r10** is used in two different clock cycles. In this case, there is no penalty for the latency of the **or** operation. However, in Figure 9-1(b), the **add** operation could have issued during the same clock cycle as the **or** , but the results of the **or** will not be ready until the next clock cycle, thus a single bubble is introduced into the instruction stream. In this case there is effectively a half-clock penalty induced by the latency of the **or** operation.

Effective throughput of more than one instruction per clock cycle can be realized by the many performance features in the MC88110, including pipelining, superscalar instruction issue, feed forwarding, branch acceleration, and multiple execution units which operate independently and in parallel.

Many of the execution units on the MC88110 are said to be pipelined. This implies that the particular execution unit is broken into stages. Each stage performs a specific step, which contributes to the overall execution of an instruction. The pipelined design is analogous to an assembly line where workers perform a specific task and pass the partially complete product to the next worker.

When an instruction is issued to a pipelined execution unit, the first stage in the pipeline begins its designated work on that instruction. As an instruction is passed from one stage in the pipeline to the next, evacuated stages may accept new instructions. This design allows a single execution unit to be working on several different instructions simultaneously. Once the pipeline has been filled with instructions, the execution unit will complete a multi-cycle instruction every clock. Figure 9-2 shows a graphical representation of a generic pipelined execution unit.
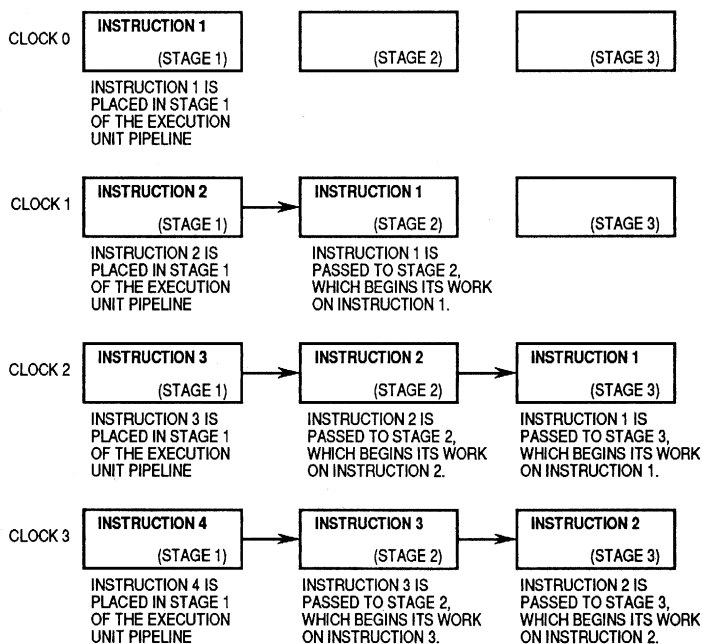
Figure 9-2. Pipelined Execution Unit

If the number of stages in each pipeline is equal to the total latency in clock cycles of its respective execution unit, the processor can continuously issue instructions to the same execution unit without stalling. Thus, when enough instructions have been issued to an execution unit to fill its pipeline, the first instruction will have completed execution and

left the pipeline, allowing subsequent instructions to be issued into the tail of the pipeline without interruption.

The MC88110 is capable of issuing and executing two instructions on every clock cycle. In general, instruction execution is accomplished in three stages: the prefetch and decode stage, the execute stage, and the write-back stage. Often, two instructions are proceeding through each of these stages concurrently, as shown in Figure 9-3.

The instruction prefetch and decode stage consists of the reply phase of the instruction fetch as well as the time to fully decode the instruction. Instruction decode time is minimal since none of the instructions are implemented with micro-code.

In the write-back stage, results are returned to the register file. This stage does not contribute to overall execution time (if write-back slots are available). Instructions are prefetched and executed concurrently with the execution and write-back of previous instructions producing an overlap period between instructions (see Figure 9-3). This overlap decreases the average execution time for a sequence of instructions.
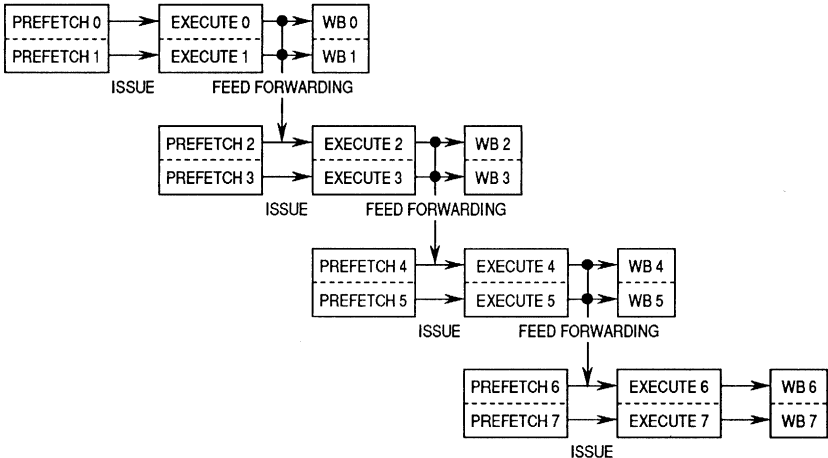
Figure 9-3. Instruction Prefetch and Execute Timing

See **9.2 General Timing Considerations** and **9.3 Execution Unit Timings** for instruction timing details.

9.2 GENERAL TIMING CONSIDERATIONS

A superscalar machine is one which can issue multiple instructions concurrently from a conventional linear instruction stream. The MC88110 is a true superscalar implementation of the 88000 architecture since two instructions are decoded and issued to multiple execution units during each clock cycle. Although a superscalar implementation complicates instruction timing, these complications are transparent to the software. The MC88110 provides the logical functionality of issuing only a single instruction at a time, while providing the performance of issuing two instructions at a time.

To sustain its throughput potential, the instruction unit must be supplied with instructions at a high rate. The instruction unit generates a single address for each prefetch operation but gets two instructions from the memory system on each clock. Instructions are issued to the execution units in strict program sequence. If the first instruction in an issue pair cannot be issued, then neither instruction in the pair is issued. If the first instruction in the pair is issued but the second cannot, then the second instruction is moved into the vacated first-issue position, and a new instruction is placed in the second-issue position. If both instructions in the pair are issued, then two new instructions to be issued in the next clock cycle are fetched from the instruction cache.

When two instructions are considered for issue in the same clock cycle, the MC88110 places no restrictions on instruction type or address alignment for either instruction in the issue pair. Instructions in either slot can be from any word-aligned memory location and can be issued to any execution unit, provided that the execution unit is available and there are no data dependencies. This is known as symmetric superscalar instruction issue.

Figure 9-4 illustrates symmetric superscalar instruction issue. In this illustration, instruction N is not bound to be issued to any particular execution unit. Similarly, instruction N+1 is free to be issued to any available execution unit. This feature frees the compiler/programmer from the restrictions of specific instruction ordering or alignment.

9
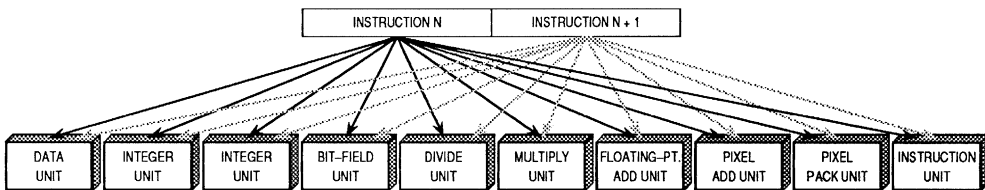
Figure 9-4. Symmetric Superscalar Instruction Issue

The execution unit pipelines are hardware interlocked via a register scoreboard; therefore, data dependencies automatically stall instruction issue without software assistance. The scoreboard mechanism eliminates the need to schedule wasteful no operation (NOP) instructions into empty pipeline delay slots.

When an instruction is issued, the register file places the appropriate source data on the appropriate source bus. The corresponding execution unit then reads the data from the bus. The register files and source buses have sufficient bandwidth to sustain the peak execution rate of two instructions per clock.

The MC88110 contains the following execution units which operate independently and completely in parallel:

- Superscalar Instruction Unit
- 80-Bit (Integer, Floating-Point, and/or Graphics) Multiply Execution Unit
- 80-Bit (Integer and/or Floating-Point) Divide Execution Unit
- 80-Bit Double-Extended-Precision Floating-Point Add Execution Unit
- Two 64-Bit 3D Graphics Execution Units
- Two 32-Bit Integer Arithmetic Logic Execution Units
- 32-Bit Bit-Field Execution Unit
- Data Unit with Load Buffer and Store Reservation Station

Each execution unit contains independent, internally controlled pipelines. All execution units are either single-cycle execution, or fully pipelined (with the exception of the divide unit, which is iterative).

When an execution unit finishes executing an instruction, it places the resulting data, if any, onto one of the destination buses. The appropriate register file then stores it into the correct destination register. If a subsequent instruction is waiting for this data, it is forwarded past the register files directly into the appropriate execution unit(s) for the immediate execution of the waiting instruction. This allows a data-dependent instruction to issue without waiting for the data to be written into the register file and then read back out again. This feature, known as feed forwarding, significantly shortens the time the machine must stall on data dependencies.

9.2.1 Instruction Issue Timing

There are several factors which affect instruction issue timing. These factors include the following:

- Can instructions be prefetched from the instruction cache (a cache hit), or, must they be fetched from main memory (a cache miss)?
- Do dependencies exist which will force an instruction stall while source data is being generated?
- Are execution units available to accept additional instructions?
- Is the history buffer full?
- Is program flow sequential?

If an instruction stall occurs (i.e., an instruction cannot be issued due to any of the factors listed in the previous paragraph), and the offending instruction is in the first issue slot of an instruction pair, then instruction issue is halted until the cause of the stall is resolved. If the offending instruction is in the second issue slot, the instruction in the first issue slot is issued to the appropriate execution unit, the offending instruction is moved into the first issue slot, and another instruction is fetched into the second issue slot.

Figure 9-5 shows an example of a stalled instruction. In this example, instruction 3 is stalled while in the second-issue position. Immediately, it is placed in the first-issue position for the next clock, and instruction 4 is placed in the second-issue position. Notice that instruction 5 is prefetched twice. When only one instruction is issued during a clock cycle, an instruction will be prefetched twice (instruction 5 in this case). The cost of prefetching and decoding an instruction the first time is zero since it is done in parallel with the prefetch and decoding of another instruction.
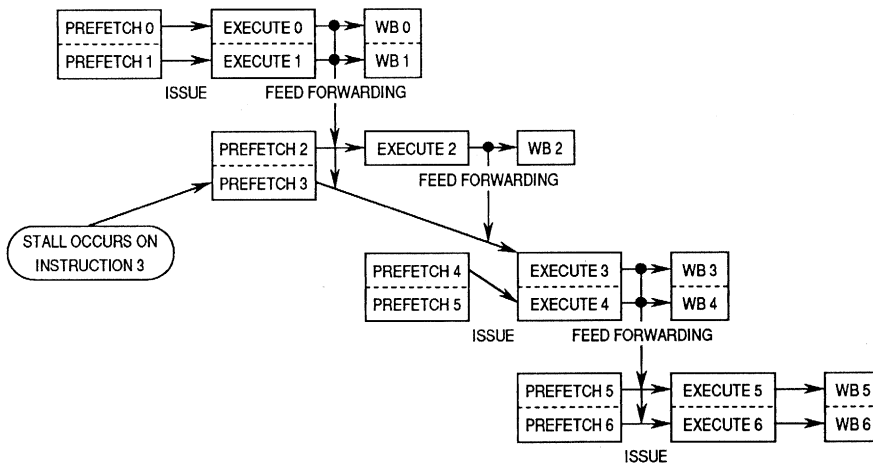
Figure 9-5. Instruction Execution Order

For detailed bus timing information, refer to **Section 11 System Hardware Design** For additional instruction execution timings, refer to **9.3 Execution Unit Timings**.

9.2.1.1 INSTRUCTION CACHE TIMING. If the required instructions are successfully prefetched from the instruction cache (cache hit), and all other requirements are met, then there are no interruptions in dual instruction issue. However, if the required instructions are not found in the instruction cache or target instruction cache, the MC88110 must fetch them from main memory (cache miss).

During the time instructions are being fetched from main memory, previously prefetched instructions continue to be issued. However, even in an ideal memory system, there are not enough prefetched instructions to completely overlap the delay incurred by an instruction fetch from main memory; thus, the MC88110 runs out of instructions to issue while it is waiting for an instruction fetch from main memory to complete. During this

waiting period, opportunities to issue instructions may be lost simply because there are no instructions to issue.

9.2.1.1.1 Instruction Cache Hit. For an instruction cache hit, a pair of instructions is fetched by the instruction unit on each clock, regardless of whether the pair is aligned to an even or odd word position in the cache. Thus, there are no instruction address alignment restrictions imposed on dual instruction issue except when the first instruction of a pair is the last word in a cache line. However, instruction addresses still must be modulo four (the two least significant bits cleared).

When the first instruction of a pair is the last word in a cache line, only one instruction can be fetched during that clock cycle. This case causes a single bubble to occur in the execution sequence. A bubble is defined as a lost opportunity to issue an instruction (see Figure 9-5).

Figure 9-6 shows a brief example of an instruction prefetch from the instruction cache and how that prefetch affects instruction issue. In this example, the first two instruction fetches hit the instruction cache (instructions 0 and 1). On clock 2, the sequencer attempts to fetch two instructions from the instruction cache but the first instruction in the fetch pair (instruction 2) is at the end of a cache line. Only instruction 2 is returned, and a single bubble occurs in the pipeline in place of the instruction 3. The next instruction fetch (instructions 3 and 4) is to the beginning of another cache line, so instruction fetches resume at a rate of two instructions per clock.
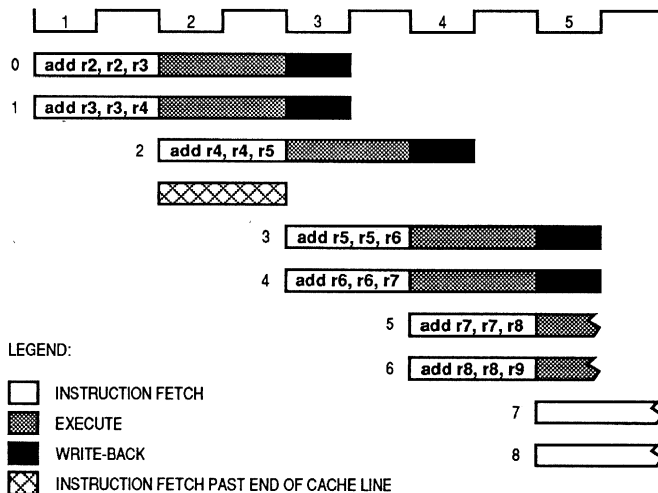
Figure 9-6. Instruction Cache Hit Timing Example

9.2.1.1.2 Instruction Cache Miss. For an instruction cache miss, if the miss is caused by the fetch of the first instruction in an issue pair, then a minimum three clock latency (best case memory access time) is incurred, which results in six lost opportunities to issue instructions (i.e., six bubbles). If the instruction pair straddles a cache line, the instruction which was successfully fetched is issued along with a single bubble in place of the second instruction. At this point, an additional minimum three clock latency occurs. The result is seven lost opportunities to issue instructions.

Figure 9-7 shows a brief example of an instruction prefetch which misses the instruction cache and shows how instruction issue is affected. In this example, a new instruction pair is requested from the instruction cache on clock 2 (instructions 2 and 3) and the first instruction address misses the cache. A bus transaction begins on clock 3 to fetch the missed line into the cache. Assuming an ideal memory system, two instructions are received at the end of clock 5 and are forwarded to the instruction unit in clock 6. During clock 6 both instructions are issued. Meanwhile, the next instruction pair is being received from the bus and streamed into the instruction unit. As long as the bus continues to receive instruction pairs every clock, instruction issue will continue without interruption.

Had there not been an instruction cache miss, instructions 2 and 3 would have been issued on clock 3; thus, the miss had a latency of three clocks and caused a total of 6 instruction bubbles.
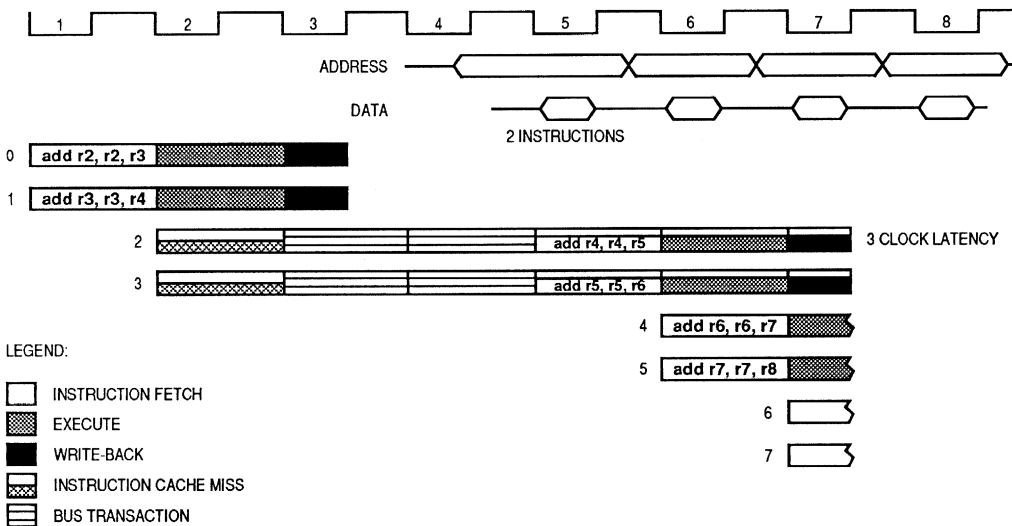
**Figure 9-7. Instruction Cache Miss Timing—
First Instruction in Pair Missed**

In the example shown in Figure 9-8, instruction 2 is the last instruction in a cache line so only a single instruction (instruction 2) is fetched and sent to the instruction unit. In clock 3, during the next instruction fetch, a cache miss occurs. The bus transaction begins on clock 4. Since the missed address is for data at the beginning of a cache line, the address is evenly aligned, so dual instruction issue resumes on clock 7.
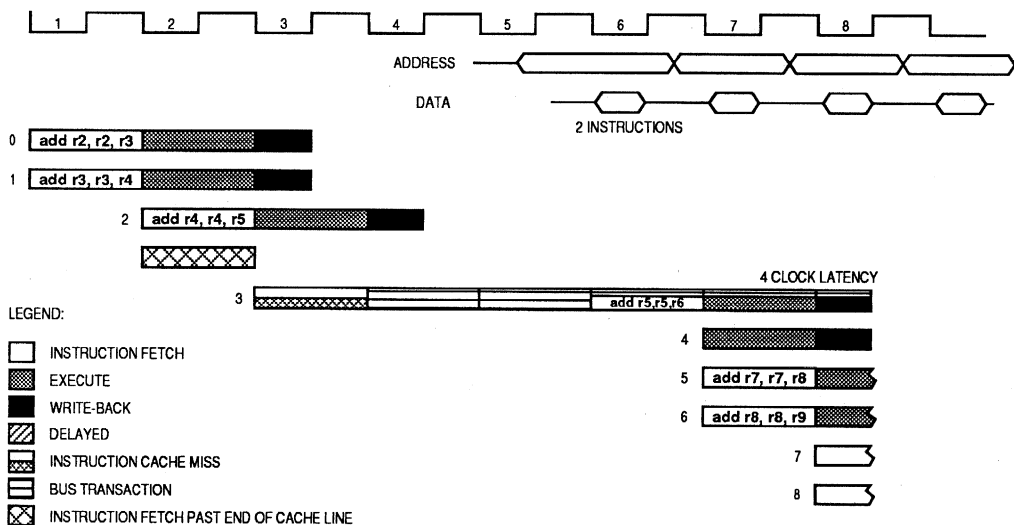
**Figure 9-8. Instruction Cache Miss Timing—
Second Instruction in Pair Missed**

Figure 9-9 shows an example of when the opportunity for instruction streaming is missed. Recall that when instructions or data is read from memory, the information can be forwarded to a waiting execution unit directly from the bus as it is written to an on-chip cache. The information arrives from memory in 64-bit packets, and if the processor only needs 32-bits of data at a time, it is possible that the information which an execution unit would like to have forwarded has already been written to the on-chip cache and new data is coming in on the memory bus. In this case, the processor must wait until the burst read is complete before the data can be read back out of the on-chip cache. This is known as missing the stride of arriving information and can occur when attempting to read instructions or data directly from the bus as they arrive from external memory.

On clock 0 in Figure 9-9, instructions 0 and 1 are completing their fetch and decode stage. On clock 1, instructions 0 and 1 are executed and an instruction cache miss occurs, thus initiating an instruction fetch from main memory. Instructions 2 and 3 arrive from memory on clock 4 and are immediately decoded.

During clock 5, instructions 4 and 5 are arriving from memory and instruction 2 begins execution. Instruction 3 does not issue along with instruction 2 due to a data dependency, and is moved into the first issue position. Since only one instruction was issued during clock 5, the instruction unit will read instruction 4 directly from the bus.

During clock 6, instructions 6 and 7 are arriving from memory and instruction 3 begins execution. However, instruction 4 stalls, due to a data dependency, and is moved into the first issue position. Since only one instruction was issued during clock 6, the instruction unit will attempt to read instruction 5 from the bus, which will fail because instructions 6 and 7 are now on the bus and being sent to the on-chip instruction cache.

During clock 7, instructions 8 and 9 are arriving from memory and instruction 4 begins execution. Instruction 5 can not issue along with instruction 4 because it was not read during the last clock. The instruction unit has no more instructions to decode and execute and cannot fetch from the instruction cache because instructions 8 and 9 are being written.

During clock 8, the instruction cache line fill has completed, and a fetch from the on-chip instruction cache is initiated. Instructions 5 and 6 can be read from the on-chip instruction cache and decoded during clock 8. Instructions 5 and 6 can begin execution on clock 9 while instructions 7 and 8 begin their decode stage. There are a total of two lost opportunities to issue instructions during clocks 8. However, if the processor never had the capability of streaming data directly from the bus, the execution of instruction 2 would be pushed out to clock 9. Thus, the penalty of missing the stride of arriving information only means that the processor cannot take full advantage of data streaming.
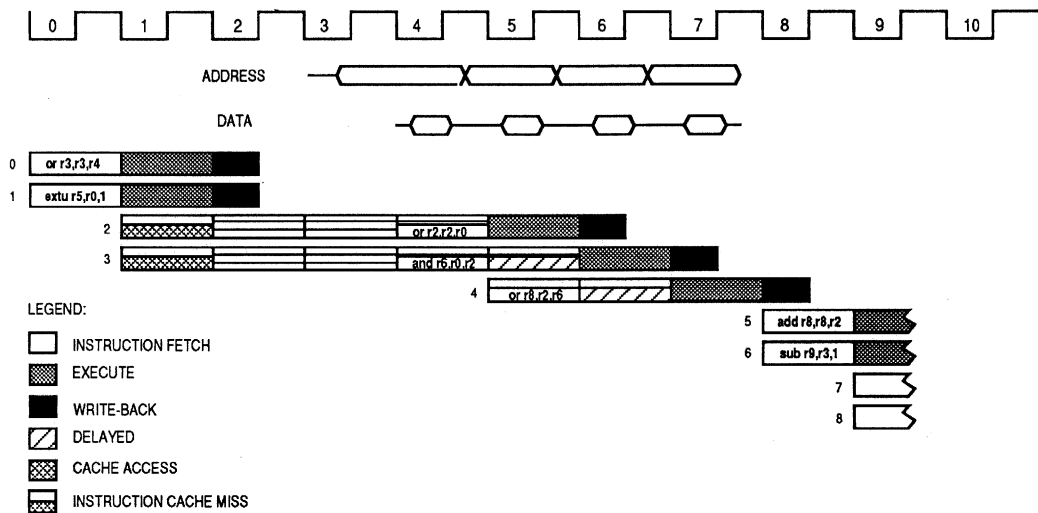
Figure 9-9. Missing the Stride of Arriving Information

9.2.1.2 SOURCE DATA CONSIDERATIONS. If an instruction attempts to use a source operand which is still being computed by a previous instruction, a data dependency exists. When a data dependency exists, instruction issue is stalled until all of the necessary source data is available (except for branch and store operations). The MC88110 employs a register scoreboard mechanism to keep track of which registers are and are not available for use.

Feed forwarding allows data to be simultaneously written to a register file and forwarded to a waiting instruction. The register scoreboard is used as an efficient method of stalling instruction issue when a data dependency exists, and feed forwarding is an efficient method for minimizing that stall time.

9.2.1.2.1 Scoreboard Checks. The scoreboard is used to keep track of operand availability. Conceptually, the scoreboard is a bit vector, and each bit in the vector corresponds to a register in the register files.

Whenever an instruction is issued, the scoreboard bit corresponding to the instruction's destination register is set, thus marking the register as busy. When the instruction completes execution and writes back its result to the destination register, the scoreboard bit corresponding to the destination register is cleared, thus marking the data in that register as available for use.

The scoreboard bits for all of an instruction's source and destination registers (except store and branch operations) must be clear (or will be cleared during the issue clock cycle) for that instruction to be issued. If the corresponding scoreboard bits are set, instruction issue is stalled until those scoreboard bits are cleared by the instruction(s) currently using the registers.

As described earlier, the MC88110 attempts to issue two instructions simultaneously. Since the scoreboard cannot be updated instantaneously, the scoreboard mechanism cannot be used to resolve data dependencies between instructions within an issue pair. These dependencies are resolved by interdependency resolution hardware whose effect is similar to a scoreboard.

Since the contents of **r0** and **x0** are hardwired to zero, neither of the scoreboard bits for these registers are ever set since their data is always current. However, **r0** and **x0** are subject to interdependency checks. In other words, if **r0** is a destination for the instruction in issue slot one and is also used as a source for the instruction in issue slot two, the interdependency resolution hardware will prevent the second instruction from issuing during the current clock cycle.

9.2.1.2.2 Feed Forwarding. In a highly parallel microprocessor, each clock cycle is valuable. In order to minimize the overhead of data dependencies in the instruction stream, the MC88110 implements a design feature known as feed forwarding. When an instruction has stalled (or will stall on the next clock cycle) because source data is not available, feed forwarding allows the operand to be forwarded directly to the waiting instruction as soon as it is available (see Figure 9-10). This forwarding occurs in parallel with the register write-back and clearing of the scoreboard bits.
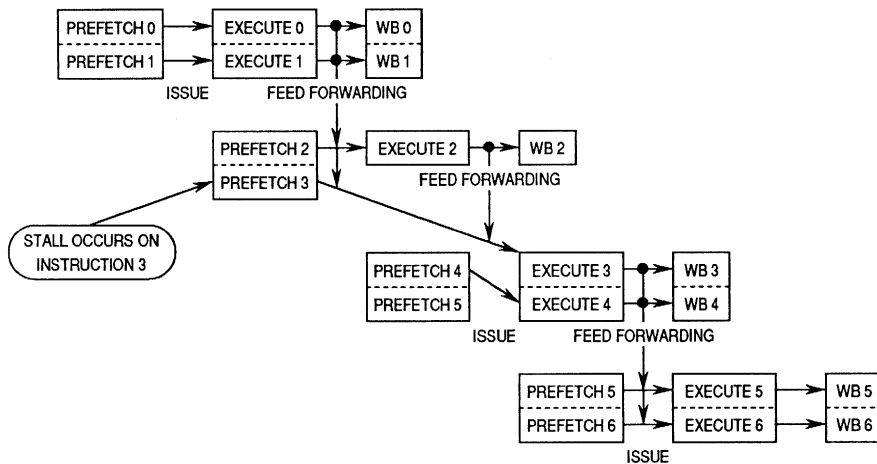
Figure 9-10. Feed Forwarding

9.2.1.3 DESTINATION REGISTER CONSIDERATIONS. The following paragraphs describe how the MC88110 prevents destination registers from being overwritten by out-of-sequence instructions and how instructions are prioritized for writing back to the register files.

9.2.1.3.1 Scoreboard Checks. In a machine that allows instructions to complete out of order, there is the potential for an instruction's result to be overwritten by an instruction which issued earlier but completed later. To preclude this possibility, the scoreboard bit corresponding to the destination register is automatically checked as a condition for instruction issue. This ensures that updates to any given register are always completed in the order specified by the program and thus no data is ever incorrectly overwritten in the register files.

The data unit maintains its own set of rules for instructions being issued and completed out of order. Only one memory access instruction can be issued per clock cycle; however, the load buffer and the store reservation station (see **9.2.2 Load Buffer and Store Reservation Station Model**) help minimize the effects of long memory latencies. Store (**st**) instructions may be issued into a reservation station before the data being stored is available. This allows continued issuance and execution of other instructions in parallel with the computation of the source data for the store operation. In other words, issue of **st** instructions is not prevented by scoreboard checks on the data being stored. Note that address operands for the **st** operation are vulnerable to scoreboard checks. Additionally, load (**ld**) instructions are allowed to bypass **st** instructions which are stalled in the reservation station as long as the address being accessed by the **ld** instruction does not match that being accessed by any waiting **st** instructions.

9.2.1.3.2 Write-back Priorities. There are two destination buses available on the MC88110. Since different execution units have different pipeline lengths, it is possible for more than two instructions to complete in a given clock cycle; therefore, execution units arbitrate for an available destination bus. Highest write-back priority is granted to single-cycle execution units (integer and graphics units). Thus, single-cycle instructions are always guaranteed the use of a destination bus while multi-cycle execution units arbitrate for their chance to use a destination bus. The priorities for each of the various execution units are as follows:

1. ALU, Bit-Field, and Graphics Units
2. Floating-Point Add Unit
3. Multiply Unit
4. Divide Unit
5. Data Unit

While waiting for a bus grant, execution units which have been denied a destination bus continue to advance their internal pipeline stages and accept new instructions until all pipeline stages are full.

9.2.1.4 EXECUTION UNIT CONSIDERATIONS. For an instruction to be issued, the required execution unit must be available. The sequencer monitors the availability of all execution units and suspends instruction issue if the required execution unit is not available. An execution unit may not be available under the following circumstances:

1. A multi-cycle, nonpipelined unit can have only one instruction in execution at a time. Such a unit becomes busy when an instruction is issued to it, and it cannot accept another instruction until the previous one completes. The divide unit is the only such unit on the MC88110.

2. An execution unit may become unavailable for additional instructions if its pipeline becomes full. This situation may occur if execution takes more clock cycles than the number of pipeline stages in the unit and enough additional instructions are issued to that unit to fill the remaining pipeline stages. This situation can only occur in the data unit. In addition, if the execution unit cannot get access to a write-back slot while additional instructions continue to fill its pipeline, the pipeline may become full.

3. Execution units can accept only one instruction per clock. Attempting to issue two instructions to the same unit on the same clock will cause a stall.

Figure 9-11 illustrates which instruction pairs can and cannot be issued simultaneously due to the one instruction per execution unit per clock restriction. For example, if the first instruction in an issue pair is an **add**, then the top row of the grid in Figure 9-11 shows that any type of instruction can be issued concurrently with the **add** (all the boxes on the top row are shaded), provided there are no data dependencies. On the other hand, if the first instruction in an issue pair were a **muls**, then the fourth row of the grid in Figure 9-11 shows that another **muls**, a **pmul** or an **fmul** (the three white boxes on row four) cannot be issued along with a **muls** instruction.

Notice that if Figure 9-11 were divided from the top-left corner to the bottom-right corner, each side of the figure would be a mirror image of the other side. This phenomenon occurs because the MC88110 is a symmetric superscalar implementation. Each instruction pair that can be issued together, can also issue together if the order of the instructions were reversed. This provides a more flexible environment for instruction scheduling and optimization.

There are several important points that should accompany a discussion of dual instruction issue with respect to execution unit availability. First, if the serial mode bit (SER) in the processor status register (PSR) is set, then simultaneous instruction issue is disabled and, at most, only one instruction can be issued per clock cycle. Second, when instructions which affect the carry flag (**add**, **sub**, **addu**, or **subu** with .co or .cio suffix) are issued as the first instruction in an issue pair, they prevent issue of any other instruction in the same clock. Third, any instruction which specifies **r0** or **r1** as a source or destination and is in the delay slot of a **bsr.n** instruction will not issue in the same clock as the **bsr.n**.

9

Figure 9-11. Simultaneous Instruction Issue Restrictions
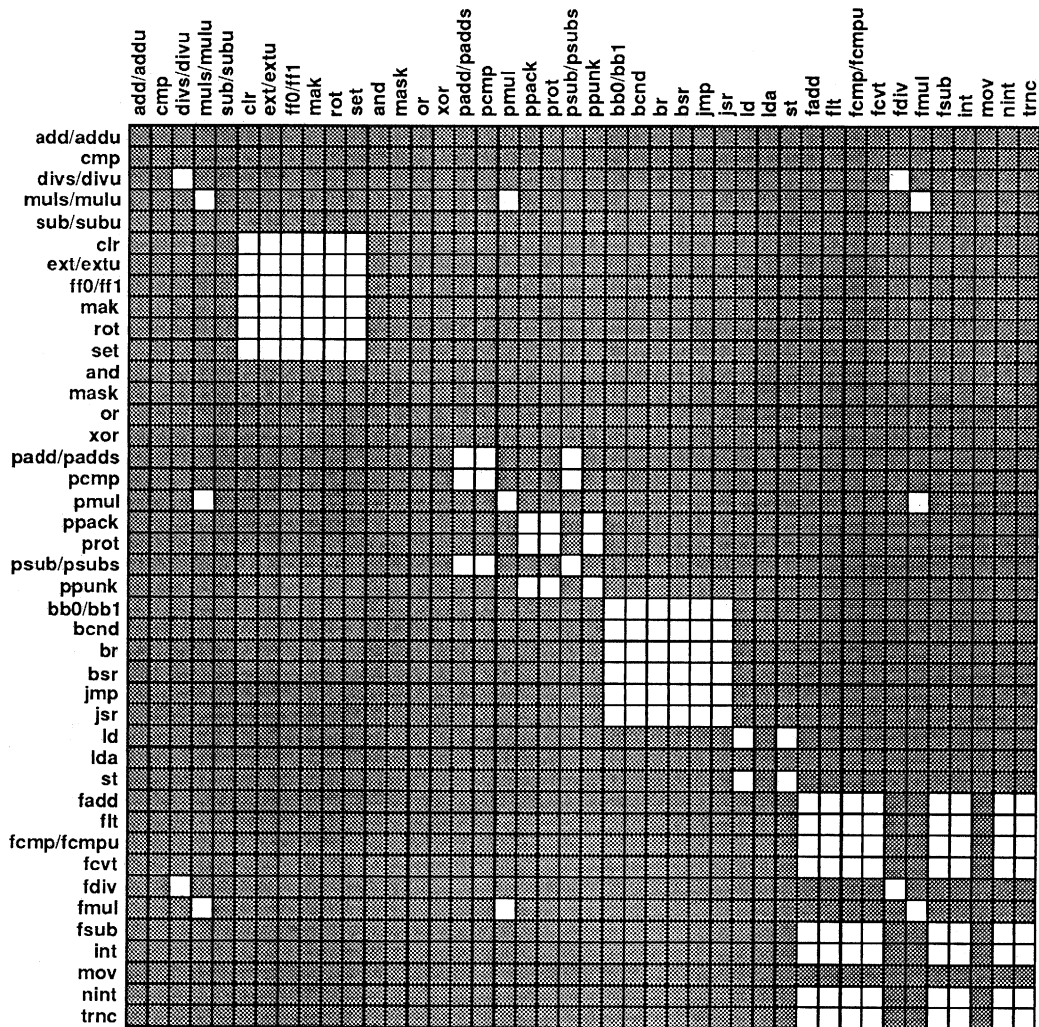
LEGEND:

▨ CAN BE ISSUED SIMULTANEOUSLY

☐ CANNOT BE ISSUED SIMULTANEOUSLY

NOTE

All instructions which cause the machine to serialize (**xmem, tb0, tb1, tcnd, rte, ldcr, xcr, stcr, fldcr, fxcr,** and **fstcr**) cannot be issued in the same clock with any other instruction.

9.2.1.5 HISTORY BUFFER INDUCED STALLS. Although the MC88110 issues instructions in strict sequential order, it is possible for instructions to complete execution out of order. The MC88110 keeps an internal first-in-first-out (FIFO) queue of all instructions that are executing. This feature, called the history buffer, keeps all details of out-of-order execution internal to the processor. To user software, the processor appears to issue and execute instructions in a strict sequential fashion.

At the time of issue, an instruction is placed at the tail of the queue. The instructions move through the FIFO queue until they reach the head. An instruction reaches the head when all of the instructions in front of it have finished execution. However, since instructions can be executed out of order, it is possible for an instruction to have completed execution, but still be in the middle of the queue. An instruction is retired from the history buffer when it reaches the head and it has completed execution.

Figure 9-12 shows an example of the history buffer where an **fdiv.ddd** instruction (a multi-cycle instruction) was the first instruction issued by the MC88110 followed by a series of single-cycle instructions. Until the **fdiv.ddd** has finished execution, no subsequent instructions can be retired from the history buffer.

TOP OF HISTORY BUFFER ⟶	**fdiv.ddd**	**r8, r6, r4**
	and	r15, r16, r17
	or	r2, r11, r13
	•	
	•	
	•	
	addu	r3, r11, r12
BOTTOM OF HISTORY BUFFER ⟶	**subu**	**r10, r11, r10**
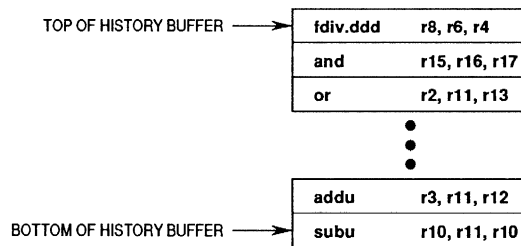
Figure 9-12. History Buffer

The history buffer contains 12 cells. It is possible to fill the history buffer to capacity. In this case, the MC88110 stalls instruction issue until the instruction at the head of the buffer completes execution and is retired from the queue.

The following algorithm is used to retire instructions from the history buffer:

```
while( history_buffer[top] == completed_execution )      /* has top instruction completed ?            */
{
    retire( history_buffer[top] )                        /* empty the top cell of the history buffer   */
    history_buffer[top] = history_buffer[top - 1]        /* shift every cell up one cell               */
    ...
    history_buffer[bottom + 1] = history_buffer[bottom]
    history_buffer[bottom] = nil                         /* clear bottom of history buffer             */
}
```

As long as the head of the history buffer contains an instruction that has completed execution, that instruction is retired from the history buffer and the remaining cells are shifted up. This algorithm is run to completion after every clock cycle; therefore, the history buffer can go from being completely full to completely empty in a single clock cycle.

9.2.2 Load Buffer and Store Reservation Station Model

The data unit contains a load address buffer and a store address/data reservation station, which operate as two independent FIFO queues. After being issued, all **ld** and **st** instructions pass through the appropriate FIFO queue. See Figure 9-13 for an illustration of the load buffer and store reservation station model. Notice that there are four slots in the load buffer and three slots in the store reservation station. Provided availability, the data cache can service either a **ld** or **st** operation every clock cycle.
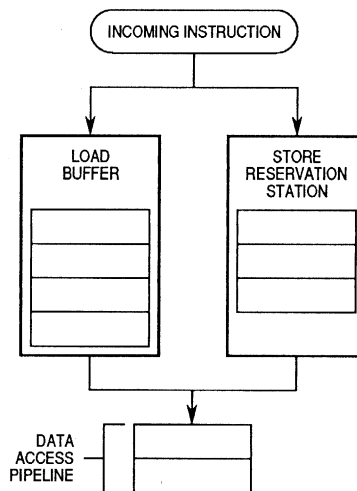
Figure 9-13. Load/Store FIFO Queue Model

The store reservation station allows **st** instructions to be issued before source data is available from a previous computation (e.g., a scoreboard hold on the data being stored does not delay the issue of **st** instructions). However, a scoreboard check is performed on the source register(s) whose contents make up the address for the store. If unavailable at the time of issue, the data being stored is forwarded directly to the pending **st** instruction in the store reservation station as soon as it becomes available.

There is an exception to a **st** operation bypassing a scoreboard check. If the data which will be stored is not available and will not be transmitted on the destination bus in the same format as the waiting **st** operation is expecting it, instruction issue will stall until the data becomes available. For example, if a single-word **st** operation will write the data in **r7** to memory, and the data in **r7** is not yet available, another check is made before

allowing the store operation to issue to the store reservation station. If **r7** is the destination register of a single-word operation, the **st** instruction will issue to the store reservation station and wait for **r7** to be forwarded. However, if **r7** is the destination register for part of a double-precision operation, instruction issue will stall until **r7** has been written back into the register file and read back out again for the waiting **st** operation.

The load buffer allows multiple **ld** instructions to be issued while a previous **ld** instruction is stalled in the load buffer. The **ld** or **st** instructions only wait in their respective FIFO queues for the following reasons:

- A **st** instruction must wait in its reservation station if the source data being stored is not available at the time of instruction issue.
- A **st** instruction must wait in its reservation station until it reaches the top of the history buffer.
- A **st** instruction must wait in its reservation station at least one clock cycle for its data to be aligned.
- A **ld** instruction must wait in its buffer if its effective memory address matches the page index of the destination address of a pending **st** instruction.
- A **ld** instruction must wait in its buffer if there are previous **ld** instructions pending.
- A **ld** or **st** instruction must wait in its respective FIFO queue if the memory system (cache) is busy.

The feature described in the second bullet of the previous paragraph is used to ensure the precise exception model on the MC88110. For more information on the exception model on the MC88110, refer to **Section 7 Exceptions**. In the case of an interrupt or exception, the MC88110 must be able to back out of any instructions that may have been issued after the excepting instruction. Since the destination of a **st** operation is cache (and possibly main) memory, it is necessary to hold that **st** operation in the store reservation station until the MC88110 is positive that it will not need to be reversed. The only way to ensure that no instruction that was issued before the **st** operation will cause an exception is to wait until the **st** has reached the top of the history buffer.

The data unit executes the queued **ld/st** instructions as data from the cache or memory becomes available. The data unit always executes **ld** instructions in program order with respect to other **ld** instructions. Likewise, **st** instructions are also always executed in program order with respect to other **st** instructions. However, **ld** instructions are allowed to execute out of order with respect to **st** instructions. While a **st** instruction is waiting in its reservation station, subsequent **ld** instructions can bypass the pending **st** and can get access to the cache. To prevent memory conflicts, load addresses are compared to store addresses and **ld** instructions are prevented from running ahead of **st** instructions for which there is an address match. Since there is only one data path out of the data unit, if the load buffer and store reservation station each have pending instructions which need access to the cache, priority is given to a **st** instruction which is ready.

9

When a **ld** or **st** instruction is encountered, the instruction unit checks the scoreboard to determine if the operands are available and, for **ld** instructions only, makes sure that there is no destination register conflict. The sequencer then checks the data unit to verify that there is an available slot in the appropriate queue. If a slot is not available, instruction issue stalls until a slot becomes available. When a slot is available, the instruction is issued to the appropriate queue on the next clock cycle. One of the following courses of action is then taken:

- For **ld** instructions—If there are no prior instructions waiting in the load buffer, and the data cache is not busy servicing a prior request, then the **ld** instruction falls through the load buffer directly to the MMU and cache. If the data cache is busy or if there are already instructions pending in the load buffer or if there is an address match between the **ld** instruction and a pending **st** instruction, then the **ld** instruction waits in the load buffer.

- For **st** instructions—The store instruction waits in the reservation station while the data (if available) is properly aligned for the cache/external memory. If all previous instructions are complete (the **st** operation has advanced to the top of the history buffer) and the data cache is available, the **st** is issued to the data cache on the next clock cycle.

Once **ld** and **st** instructions have been issued to the appropriate queue, the sequencer is free to continue issuing other instructions. Note that the **xmem** instruction is a special case. Before an **xmem** instruction issues, the MC88110 serializes (all pending instructions complete execution). This ensures that the load buffer and store reservation station are empty and the load and store operations which make up the **xmem** operation can issue unimpeded.

9

9.2.2.1 LOAD BUFFER AND STORE RESERVATION STATION EXAMPLE.
The following paragraphs and figures show an example of instructions moving through the load buffer and store reservation station:

Clock Cycle One
> During the first clock cycle (see Figure 9-14), the first two instructions in the sequence are issued. Recall that the MC88110 is capable of issuing both of these instructions in a single clock cycle. At this point, both the load and store queues are empty. The **fdiv.sss** is a multi-cycle instruction; thus, the contents of **r8** will not be ready for approximately 13 clock cycles.

fdiv.sss	r8, r7, r6
add	f11, r10, r9
st	r8, r10, r9
fadd.sss	r15, r14, r13
st	r15, r16, r0
add	r4, r3, r2
ld	r12, r11, r0
or	r2, r5, r0

LOAD BUFFER

STORE RESERVATION STATION
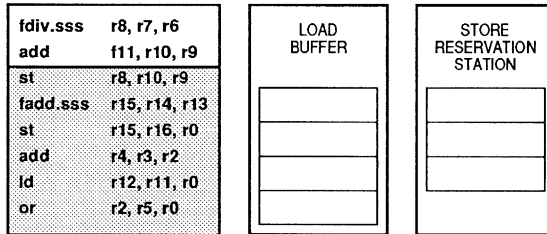
Figure 9-14. Clock Cycle One—Load/Store Example

Clock Cycle Two
> During the second clock cycle (see Figure 9-15), the contents of **r8** are needed by a **st** instruction. However, the **fdiv.sss** will not be finished with **r8** for 12 more clock cycles. The **st** instruction is issued anyway (thereby not stalling the instruction pipeline), and it waits in the store reservation station for the contents of **r8** to become available. Instruction issue continues as normal. The **fadd.sss** instruction is also a multi-cycle instruction; thus, the contents of **r15** will not be ready for another three clock cycles.

fdiv.sss	r8, r7, r6
add	f11, r10, r9
st	r8, r10, r9
fadd.sss	r15, r14, r13
st	r15, r16, r0
add	r4, r3, r2
ld	r12, r11, r0
or	r2, r5, r0

LOAD BUFFER

STORE RESERVATION STATION

st r8
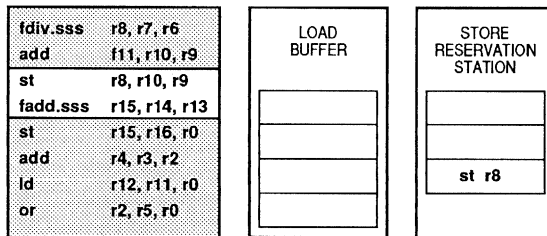
Figure 9-15. Clock Cycle Two—Load/Store Example

9

Clock Cycle Three

During the third clock cycle (see Figure 9-16), the contents of **r15** are needed by a **st** instruction; however, the **fadd.sss** will not be finished with **r15** for two more clock cycles. The **st** instruction is issued anyway (thereby not stalling the instruction pipeline), and it waits in the store reservation station for the contents of **r15** to become available. Even if the contents of **r15** had been available, the second **st** instruction would still enter and remain in the store reservation station since **st** instructions must execute in strict program sequence relative to other **ld** and **st** operations.

Instruction issue continues as normal.

Although the contents of **r15** will become available before the contents of **r8**, the **st r15** will continue to wait since it is behind another instruction in the store reservation station and the store reservation station is a strict FIFO queue.
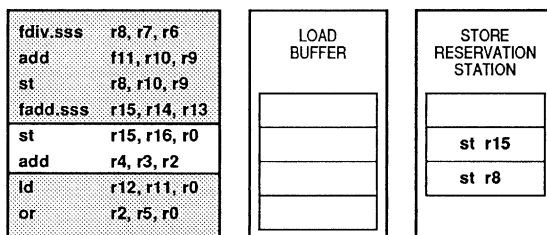
Figure 9-16. Clock Cycle Three—Load/Store Example

Clock Cycle Four

During the fourth clock cycle of this sequence (see Figure 9-17), the **ld** instruction is requesting the contents of the memory location whose address is located in **r11**. The address in **r11** resulted from adding the contents of **r9** and **r10** (the second instruction in the sequence). The address for the first **st** instruction was also formed by adding the contents of **r9** and **r10**.

When the **ld** instruction is issued, the address in **r11** is checked against all addresses in the store reservation station. Since there is a match with the first **st** instruction in the store reservation station, the **ld** instruction cannot execute. The **ld** instruction is issued anyway (thereby not stalling the instruction pipeline), and it waits in the load buffer until the pending **st** instruction has completed execution. This allows instructions to continue issuing while the **ld** instruction is pending.

If the address used in the **ld** instruction had not matched any of the addresses in the store reservation station, the **ld** instruction would have fallen through the load buffer and begun execution out of order with respect to the two pending **st** instructions in the store reservation station.

9

Figure 9-17. Clock Cycle Four—Load/Store Example

9.2.2.2 LOAD/STORE REORDERING EXAMPLE.

This example illustrates the run-time reordering of **ld** and **st** instructions (see Figure 9-18). In this example, a floating-point operation is followed immediately by a **st** of the result. The **st** instruction is issued to the store reservation station while it waits for its source data.

The **st** instruction is immediately followed by three **ld** instructions. The first **ld** is given access to the data cache while the previously issued **st** instruction waits for its data. By the time the second **ld** operation is ready to access the data cache (clock cycle 5), the data that the **st** operation is waiting for has arrived. However, it will take two additional clock cycles for the store operation to properly align its data for the write, and during this time, the second **ld** (instruction 3) accesses the data cache. On clock cycle 6, the third **ld** operation accesses the data cache because the **st** operation is still aligning its data. In this example, three load operations have bypassed the pending store.

Figure 9-18. Load/Store Reordering Timing

9.3 EXECUTION UNIT TIMINGS

After instructions are prefetched, they are either executed by the instruction unit (in the case of flow-control instructions) or dispatched to another on-chip execution unit. The following paragraphs describe the execution of instructions within the seven categories of execution units (integer, bit-field, logical, data, floating-point, instruction, and graphics units).

The clock counts presented in the following tables represent the latency induced by an instruction. Latency is the number of clock cycles necessary to execute an instruction and make ready the results of that execution. For the majority of instructions in the MC88110, this can be simplified to include only the execute phase for a particular instruction; however, data instructions will require additional clock cycles between the execute phase and the write-back phase due to memory latencies. The latencies in this section represent execution under ideal conditions. The latency of an instruction does not represent the average execution time since execution timing depends on the dynamic state of the pipelines in the MC88110.

9.3.1 Integer/Bit-Field Unit Execution Timing

There are three integer units in the MC88110—two identical arithmetic logic units (ALUs) and one bit-field unit (BFU). Each integer unit has a one clock execution phase and can process instructions at a rate of one instruction per clock; however, accesses to control registers serialize the machine causing longer latencies. Since the maximum issue rate of the MC88110 is two instructions per clock and there are two ALUs, instructions are never delayed due to an unavailable ALU. However, since there is only one BFU, only one bit-field instruction can be issued per clock.

Table 9-1 lists the latencies for instructions executed by the integer/logical/bit-field units. The Instruction Timing: time for updating the destination register is not included in the latencies listed because the write-back occurs in parallel with the execution of other instructions and does not cause an additional delay.

9

Table 9-1. Integer, Logical, and Bit-Field Execution Timings in Clock Cycles

Instruction	Latency
Integer (ALU)	
add	1
addu	1
sub	1
subu	1
cmp	1
lda	1
add.cio	1
addu.cio	1
sub.cio	1
subu.cio	1

Instruction	Latency
Logical	
and	1
mask	1
or	1
xor	1
Bit-Field (BFU)	
clr	1
ext	1
extu	1
ff0	1
ff1	1
mak	1
rot	1
set	1

Figure 9-19 illustrates an example of integer unit operation timing. During the first clock, instructions 0 and 1 are fetched from the instruction cache. During the second clock, instructions 0 and 1 are issued to the two ALUs and complete execution. Meanwhile, instructions 2 and 3 are being fetched. During the third clock, instruction 2 is issued to an execution unit but instruction 3 is delayed due to a scoreboard hold placed on **r2** by instruction 2. With instruction 2 issued, instruction 3 moves to the first issue slot and instruction 4 is fetched and placed in the second issue slot. During clock 4, instruction 2 forwards its result so that instruction 3, which was waiting for the data, can execute. Instruction 4 attempts to execute but is delayed by a destination register conflict with instruction 3. Instruction 4 then moves into the first issue slot and instruction 5 is fetched and placed in the second issue slot. During clock 5, instruction 3 releases its destination register (**r5**), instructions 4 and 5 are issued together, and instructions 6 and 7 are fetched. Instruction 6 is a bit-field instruction and is issued to the BFU in the same clock as instruction 7 is issued to an ALU. The next instruction pair, 8 and 9, however, are both bit-field instructions. Since there is only a single BFU, instruction 8 is issued while instruction 9 is delayed until the BFU is free to accept another instruction. Bit-field instruction 10 is fetched during clock 7 and attempts to be issued in clock 8 but is delayed until the BFU is free to accept another instruction.

9
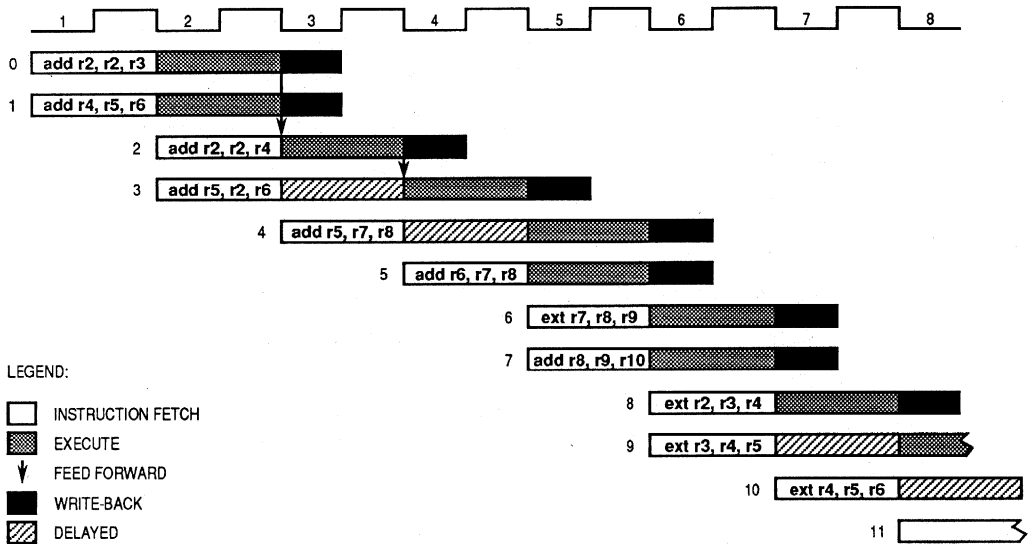
Figure 9-19. Integer and Bit-Field Instruction Sequence Timing

LEGEND:

- ☐ INSTRUCTION FETCH
- ▨ EXECUTE
- ↓ FEED FORWARD
- ■ WRITE-BACK
- ▨ DELAYED

9.3.2 Data Unit Execution Timing

The data unit is implemented as an independent execution unit. Stalls in this unit do not cause stalls in instruction issue (except in the case of a data dependency, or if the load/store queues are full and the instruction unit needs to issue an additional data access instruction). Only one data access instruction (either **ld, st** or **xmem**) can be issued to the data unit per clock cycle.

Single-word and double-word data require one clock to be accessed from the data cache while double-extended-precision data requires two clocks per access. All data transfers between the data unit and the register files occur in a single clock cycle since the internal data paths are 80-bits wide.

Table 9-2 shows the latencies for the instructions executed by the data unit. Notice that the **xmem** instruction causes the machine to serialize; therefore, all pending instructions in the execution unit pipelines and buffers will be executed before the **xmem** instruction begins execution. In addition to the time it takes for the machine to complete its serialization, the **xmem** instruction takes 12 clock cycles to execute, assuming a zero wait state memory access time.

As shown in Table 9-2, a load operation that hits in the data cache has a latency of two clocks (3 for double-extended-precision data). The latency for a load operation which misses in the cache, assuming zero wait state memory references, is five clocks (six for double-extended-precision data).

Table 9-2. Data Unit Execution Timings in Clock Cycles

Instruction	Transfer Size (Instruction suffix)				Latency
	8/16 (.b/.h)	3 2	64 (.d)	80 (.x)	
ld	•	•	•		2*
				•	3*
st	•	•	•	•	1
xmem	•	•			Serialize + 10

*Add three more clocks for cache miss assuming zero wait state memory

9.3.2.1 DECOUPLED CACHE ACCESSES. It is possible for a data instruction to access the on-chip data cache while a previous data instruction is accessing main memory. These accesses are known as decoupled cache accesses. Both load and store operations provide opportunities for decoupled cache accesses. A store operation may only access the decoupled cache during a load which has missed the on-chip data cache and has bypassed the store. Load operations may access the decoupled cache during either a touch load (see **9.3.2.2.2 Touch Load**) or a store operation which has missed the on-chip data cache. Decoupled access to the cache is inhibited during:

- the first clock cycle after a cache miss
- copyback
- the cycle during which the first data of the line fill is received
- the duration of the line-fill operation

A 2/1/1/1 external memory model will provide one clock cycle of opportunity for a decoupled cache access. Similarly, a 4/1/1/1 external memory model will provide three clock cycles of opportunity for a decoupled cache access. Refer to Section **9.3.2.3.10 Touch Load Operation Timing Example** for an example of a decoupled cache access.

9.3.2.2 USER MODE CACHE CONTROL FEATURES. Four features are implemented in the MC88110 which provide explicit control over caching behavior in user mode. These features allow performance to be improved in cases where the programmer has some specific knowledge about how or when data will be used. These new features include:

- Cache Bypassing on Stores (Store-Through)
- Cache Preloading (Touch Load)
- Forced Dirty Line Flush (Flush Load)
- Line Allocation Without Line Fill (Allocate Load)

Three of the special cache control features, touch, flush, and allocate load, are specified by performing loads of various sizes into **r0**. The touch, flush, and allocate load accesses are visible on the external bus through the transfer code pins (TC3–TC0). If the processor is in user mode during one of these cache control accesses, these pins are encoded as 0010. If the processor is in supervisor mode during one of these cache control accesses, these pins are encoded as 0110. In addition, the transfer size pins

(TSIZ1–TSIZ0) determine which cache control access is being executed. These pins will indicate 01 if the data size is a word (flush load), 10 if the data size is a half-word (allocate load), or 11 if the data size is a byte (touch load).

Past and future implementations which do not support these three cache control features are compatible with code employing these features because they do not affect the functionality of the user program. Whether or not the memory references specified by these features are actually performed is irrelevant to the program; however, performance may be affected.

9.3.2.2.1 Store-Through. The store-through feature allows a user to specify that a given store operation will be forced to update main memory. This option is provided with the triadic register addressing forms of **st** instructions. The store-through option serves two purposes. First, it provides a mechanism to force a particular data to write-through the cache and into memory even if the access is to a write-back page. This can be useful in cases such as writing to a display screen (frame buffer). Second, it provides a way to prevent data that the program knows will not be reused from allocating a new cache line on a cache miss and possibly replacing a potentially more useful line in the cache. This not only avoids the wasted time of moving a line out of the cache and back in again, but also improves the hit rate of subsequent operations to that cache line.

When specified, the store-through option unconditionally causes the store operation to write-through the cache directly into memory. If a store-through access hits the cache on its way out to memory, the cache is updated but the line is not marked as dirty unless it is already dirty (dirty implies modified). It is important to note that if a store-through access hits a dirty line in the cache on its way out to memory, the entire dirty line is not written to memory. When the store-through misses the cache on its way to memory, no line is allocated in the cache (i.e., no dirty line copyback is forced, no new line is brought into the cache, no existing line is replaced, and no data is written into the cache). In this case, the access simply goes directly to memory, bypassing the cache completely.

Store-through is specified by a write-through extension (**.wt**) on any triadic register addressing form of the **st** instruction. All operand sizes and both register files are supported as shown in Table 9-3.

Table 9-3. Store-Through Format for st Instructions

Instruction	Operand Syntax	sz/xsz options
st.sz.wt	rD,rS1,rS2 rD,rS1[rS2]	.b (byte), .h (half), {blank} (word), or .d (double)
st.xsz.wt	xD,rS1,rS2 xD,rS1[rS2]	{blank} (single), .d (double), or .x (double-extended)

9.3.2.2.2 Touch Load. The touch load feature allows data to be loaded into the cache under user program control. Normally, data is brought into the cache only when it is needed. This can lead to instruction execution stalls due to dependencies on data which must be read from main memory. In many cases, however, the need for data can be predicted. By requesting data to be read into the cache before its actual use, the latency of the memory system can be overlapped with useful work, and stalls due to long latency cache misses can be minimized.

A touch load is specified by a signed byte load to **r0** as shown in the following **ld** instructions:

ld.b	**r0,rS1,rS2**
ld.b	**r0,rS1[rS2]**
ld.b	**r0,rS1,SIMM16**

If the data specified by the effective address of the touch load operation is not already in the cache, then it is brought into the cache and replaces an existing line if necessary (just as a normal load miss would).

A touch load differs from normal loads in two ways. First, a touch load never generates an exception, and, therefore, the machine never needs to recover from one. This means that a touch load can be retired from the history buffer as soon as it enters the data unit, rather than waiting until the load completes execution. Second, although a touch load operation may bring data into the cache, it does not write a result to the register files. Thus, load operations executing during a touch load do not need to run in program sequence with the touch load and can be allowed access to the cache while waiting for the touch load operation's line fill to begin.

9.3.2.2.3 Flush Load. The flush load feature forces a dirty cache line to be written out to memory. Normally, dirty cache lines are copied back to memory only as a side effect of needing to allocate a new cache line. However, it is sometimes convenient to be able to flush data in the cache to immediately update the memory image. For example, the user may store several data words to memory which get filtered by the cache and never actually update memory. In this case, the flush load option could be used to flush the data words from the cache out to memory

The flush load option allows the programmer to perform multiple store operations to a line in the cache then write the data to memory in a single burst transaction, all from user mode code; thus, the flush load option provides performance advantages over other methods of keeping memory coherent with the cache. Placing a memory page in write-through mode or using the store-through option may have an undesirable performance impact because of the multiple individual bus transactions which would occur. Also, the time required to flush a line from supervisor mode may be prohibitive.

9

A flush load is specified by a word load to **r0** as shown in the following **ld** instructions:

    ld r0,rS1,rS2

    ld r0,rS1[rS2]

    ld r0,rS1,SIMM16

When a flush load operation hits a dirty line in the data cache, the line is flushed out to memory and the modified bit for the line is cleared. On a cache miss, the flush load is treated as a NOP. A flush load can generate an exception like other data access operations.

9.3.2.2.4 Allocate Load. It is sometimes known in advance that an entire cache line is going to be overwritten. In these cases, performance could be improved if the overhead of fetching a new line from memory that is going to be overwritten could be avoided. The allocate load option provides this capability. Allocate load allows the user to allocate a line in the cache for a series of subsequent store operations while avoiding the normal line fill from memory. This option allocates a line in the cache, as any normal load does on a cache miss, but performs only a single-beat transaction on the bus rather than a full line fill bus transaction.

The allocate load option should be used with this caution: if the sequence of stores which is overwriting the allocated line is interrupted, it is possible that the partially valid line could be pushed out to memory. However, upon returning from the interrupt, the remaining stores in the sequence will be completed and the memory state will be corrected. Thus, the invalid memory version of the line in memory will only have been a transient phenomenon.

An allocate load is specified by a signed half-word load to **r0** as shown in the following example **ld** instructions:

    ld.h r0,rS1,rS2

    ld.h r0,rS1[rS2]

    ld.h r0,rS1,SIMM16

Allocate load allocates a line in the cache on a miss but only performs a single-beat bus transaction rather than a complete line fill bus transaction. When allocate load is used on a cache inhibited access, no cache line is allocated but the single-beat bus transaction is still performed. On a data cache hit, allocate load is a NOP. An allocate load can not cause a data exception.

An allocate load never generates an exception, and therefore the machine never needs to recover from one. This means that the allocate load can be retired from the history buffer as soon as it enters the data unit, rather than waiting until the operation completes execution.

9

9.3.2.3 DATA UNIT EXECUTION TIMING EXAMPLES. The following paragraphs describe ten data unit execution timing examples.

9.3.2.3.1 Load Timing with Cache Hit Example. In this example (see Figure 9-20), during the first clock, a **ld** and **add** instruction are fetched from the instruction cache. During the second clock, both instructions are issued and begin execution. The initial phase of the **ld** execution is used to compute the effective address (the logical address for the memory access). During the third clock, the load instruction accesses the data cache and fetches the data. During the fourth clock, the load transfers data to one of the destination buses and writes data into the destination register (**r2**).

Also on the second clock, another **ld** and **add** pair is fetched from the instruction cache. This time, only the **ld** instruction is issued because the **add** has a data dependency (**r7**) on the **ld**. Execution stalls until clock 5 when the load data is received from the data cache. At this point, instructions 3 and 4, which were both dependent on the data, are issued. During clock 6, instruction 5 is issued but instruction 6 is not because it is dependent on instruction 5 for its source data (**r9**). During clock 7, a pair of **ld** instructions are fetched while instructions 6 and 7 are issued. During clock 8, the first **ld** in the pair (instruction 8) is issued but the second (instruction 9) is not because the data unit is busy accepting the first **ld** (only one **ld** or **st** instruction can be issued to the data unit per clock cycle).

Instruction 3 is issued at the beginning of clock 5, which is two clocks later than the earliest it could possibly have been issued had it not had the data dependency. Thus the load hit (instruction 2 in this example) is shown to have a latency of two clocks. Had this **ld** instruction been issued in the second issue slot, and had the next instruction been a non-data dependent instruction, the **ld** would have been issued in clock 4; if the next instruction was data dependent, it would have been issued in clock 5. Therefore, when a **ld** instruction is issued in the second issue slot, a data dependent instruction that immediately follows will experience only a single-clock cycle delay.

9

Figure 9-20. Load Hit Timing

9.3.2.3.2 Load Timing with Cache Miss Example. This example (see Figure 9-21) uses the same instruction sequence as the one starting at clock two of the previous example. However, in this example, the **ld** operation at instruction 0 misses the cache in clock 3 and begins a bus transaction in clock 4. The bus transaction shown is the best possible case; no dirty line copyback is required, the bus interface unit (BIU) is parked on the bus, and the memory returns data with no wait states (2/1/1/1). The data cache is busy beginning in clock 4 and remains busy through clock 10, until the remainder of the cache line is read into the data cache and the cache tags are updated.

9

Bus transactions always begin with the address of the missed access, regardless of its offset within a cache line; therefore, during clock 6, the double word which contains the missed data is received from the bus. During clock 7, the data is feed-forwarded to the register files and the waiting instructions. The **ld** operation at instruction 4 is not issued until clock 9, due to the data dependency on **r9**. Once issued, it waits in the load buffer until it may access the data cache in clock 10.
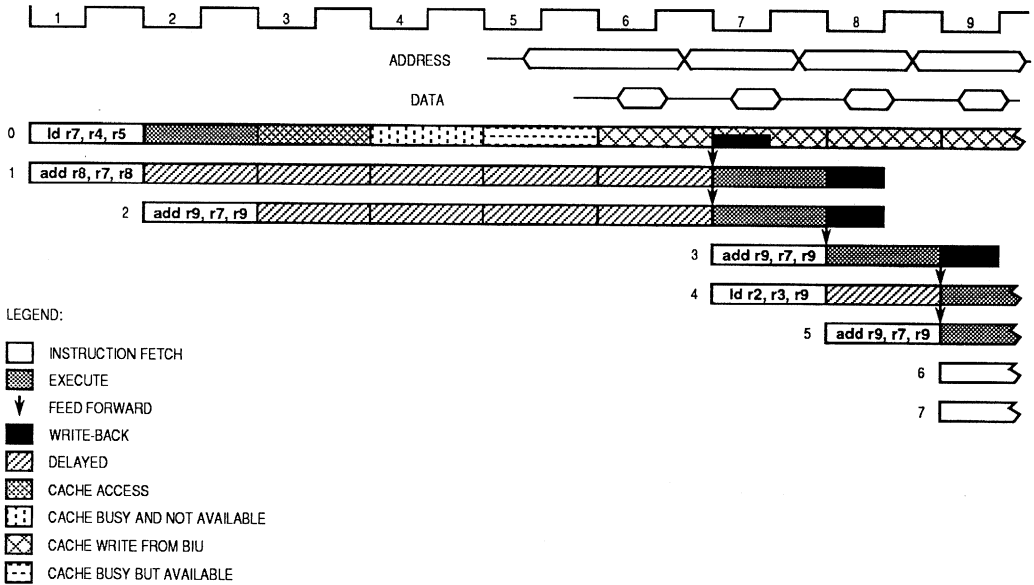
Figure 9-21. Load Miss Timing

9.3.2.3.3 Load Miss with Dirty Line Copyback Example.

This example (see Figure 9-22) also uses the same instruction sequence as the one starting in clock 2 of the example in **9.3.2.2.1 Load Timing with Cache Hit** (see Figure 9-20). However, in this example, the load operation misses the data cache and is forced to replace a dirty line in the cache. The copyback to memory begins on clock 4 and is completed on clock 9. The line fill begins on clock 11 and is completed on clock 14.
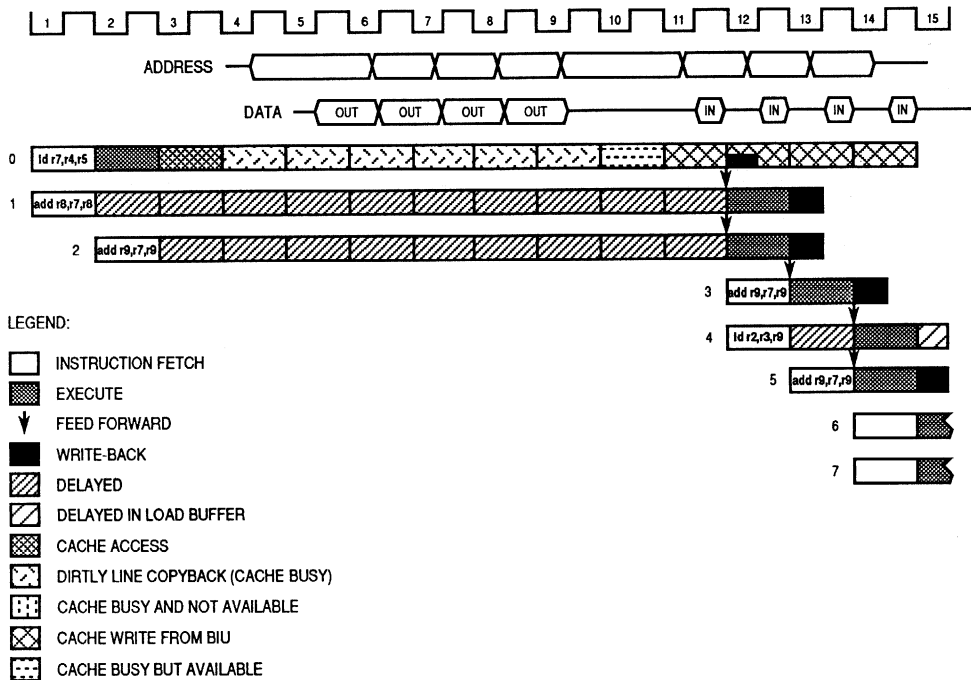
Figure 9-22. Load Miss with Copyback Timing

9.3.2.3.4 Load Miss with Instruction Overlap Example.

Figure 9-23 illustrates the execution of a code sequence which has been scheduled to avoid data dependencies resulting from the data cache miss by instruction 0. Notice that additional **ld** instructions are issued during the cache fill latency caused by the first **ld** instruction. This is possible due to the load buffer in the data unit. As a result, instruction issue continues with no stalls.

When the first **ld** operation (instruction 0) executes, its data is not in the data cache (cache miss). During clocks 2, 3, 4, and 5, additional **ld** operations are issued into the load buffer. These pending operations will remain in the load buffer until the data cache is available for access at which time the pending **ld** operations will be executed out of the load buffer in the sequence that they were issued.

Notice that five load operations are issued to the load buffer on consecutive clock cycles. This is possible because instruction 0 is retired from the load buffer just in time to make room for instruction 8.
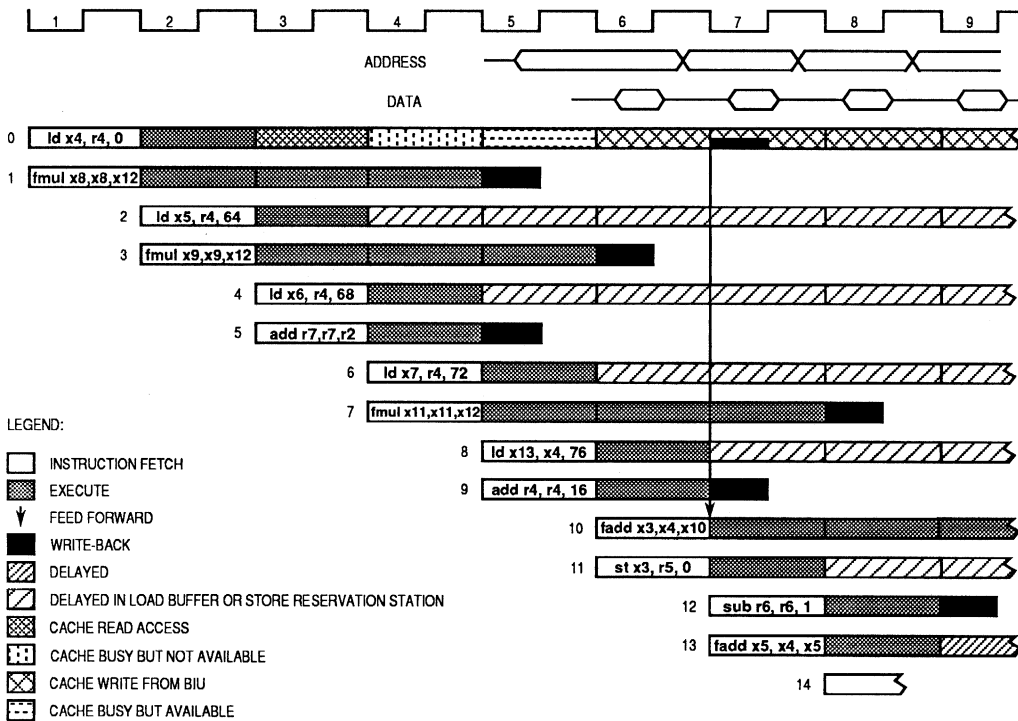
Figure 9-23. Load Miss with Instruction Overlap Timing

9.3.2.3.5 Load Miss with Data Streaming Example. The MC88110 supports streaming of data from the bus to the data unit as soon as the data is received. Instruction 0 in the Figure 9-24 is a **ld** instruction which misses the cache and begins a bus transaction in clock 4. Data is received from the bus in clock 6 and is forwarded to instruction 8 during clock 7. Meanwhile, a second **ld** instruction is issued in clock 3 and waits in the load buffer. Since the second **ld** operation is from the next memory address (relative to the first **ld**), when the next double word is received from the bus, it is forwarded to instruction 10.
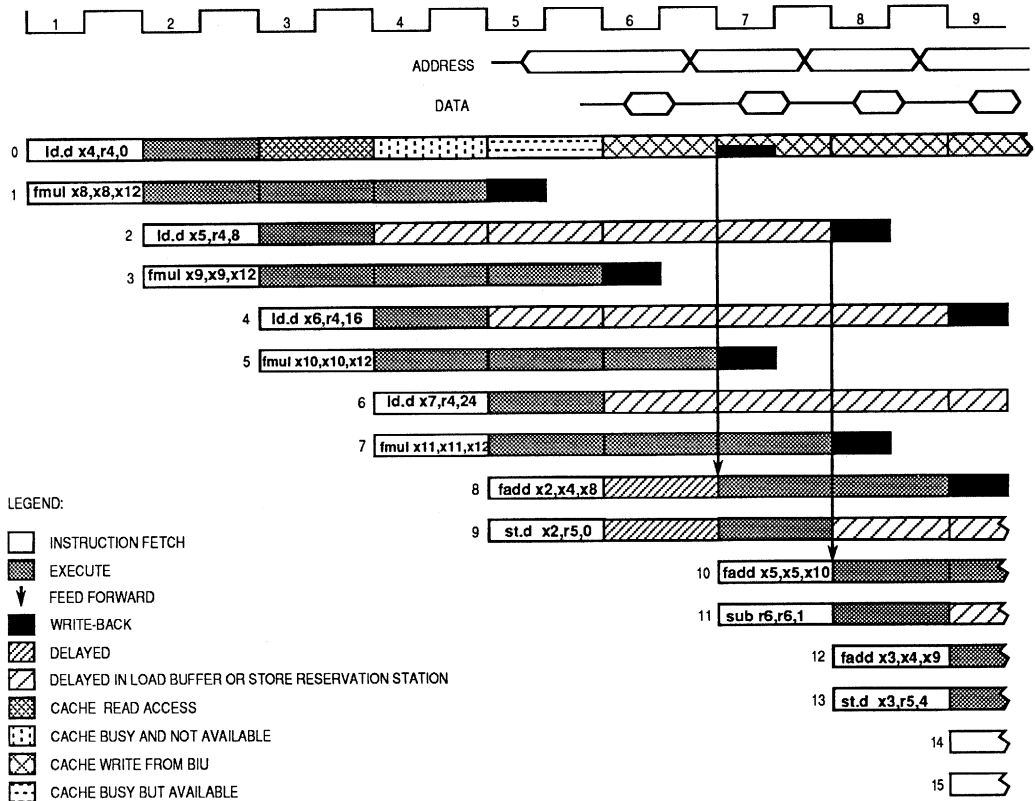
Figure 9-24. Load Miss with Data Streaming Timing

9.3.2.3.6 Store Example. Figure 9-25 shows the timing for a sequence of **st** instructions. The data unit is pipelined so that one cache access can occur every clock cycle. In this example, store operations begin accessing the on-chip data cache on clock 5.

Notice that there are clock cycles in the store pipeline labeled "data alignment." Two things occur during these two clock cycles which precede the cache access phase of the **st** operation. First, the data which is being stored is properly aligned on the 80-bit internal bus so that the correct word is written to the correct memory location. Second, if necessary, internal steps are taken which ensure the precise exception model. The instruction in the history buffer preceding the **st** determines the need for these internal steps. If the preceding instruction is another **st**, then these steps are not necessary and only one clock is needed for data alignment. If the preceding instruction is not another st, then the steps are necessary and two clocks are needed for data alignment.

The first clock cycle of data alignment for a store operation can occur during the second clock cycle of data alignment for a previous store operation. This is illustrated in Figure 9-25 during clocks 7 and 8.

0 add r2, r2, r3

1 st r3, r4, 0

2 sub r2, r2,r3

3 st x1, r4, 4

4 add r5, r4, 16

5 st x2, r4, 0

6 add r3, r3, r3

7 st r2, r4, 4

8 add r2, r2, r2

9 st r2, r4, 8

10 st r3, r4, $C

11 add r5, r2, r3

LEGEND:

☐ INSTRUCTION FETCH

▨ EXECUTE

▼ FEED FORWARD

■ WRITE-BACK

▨ DELAYED

▨ DELAYED IN LOAD BUFFER OR STORE RESERVATION STATION

▨ CACHE ACCESS

▨ DATA ALIGNMENT
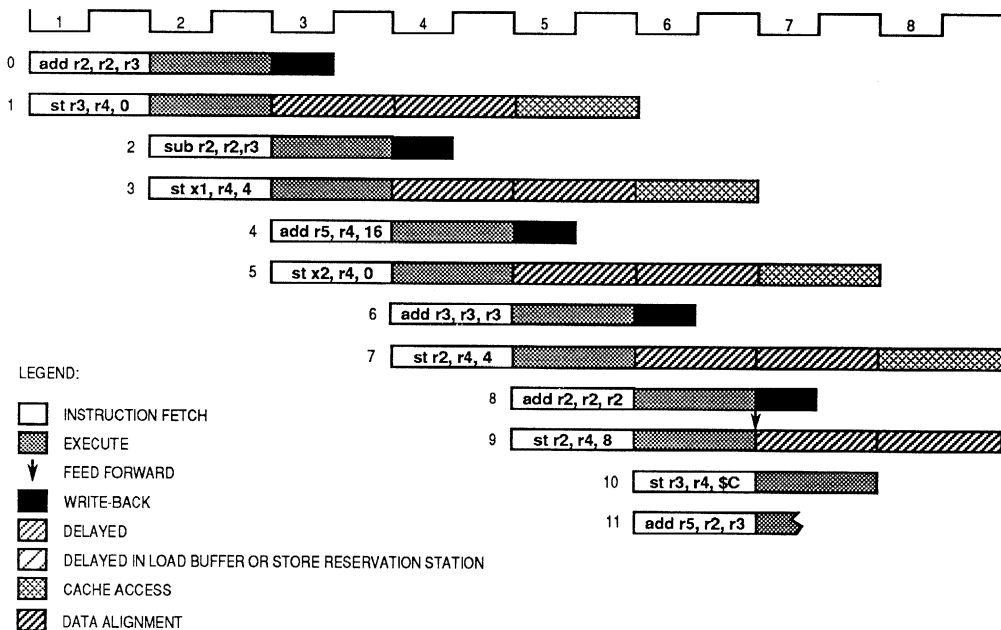
Figure 9-25. Store Timing

9.3.2.3.7 Write-Back Arbitration Example. In Figure 9-26, a floating-point operation (instruction 0) is issued in clock 2 and a **ld** operation (instruction 2) is issued in clock 2. Both of these instructions complete and need to write-back their results during clock 5; however, two integer instructions are issued in clock 4 and these integer instructions (instructions 4 and 5) have priority for both destination buses. Thus, the write-back for both the floating-point operation and the **ld** is delayed one clock until clock 6 when both operations attempt to write-back. During clock 5, two new integer instructions attempt to be issued. The first one (instruction 6) is issued and uses one of the write-back slots in clock 6. The second integer instruction (instruction 7) has a data dependency (**r2**) on the floating-point operation and therefore fails to be issued in clock 5. Since instruction 7 is not issued, one write-back slot is available in clock 6, and both the **ld** and the floating-point operation contend for it. Priority is given to the floating-point operation which writes back, and the **ld** is delayed another clock. Another data dependency (**r8**) delays instruction 8 from being issued. As a result, in clock 7, the **ld** finally gets a write-back slot.
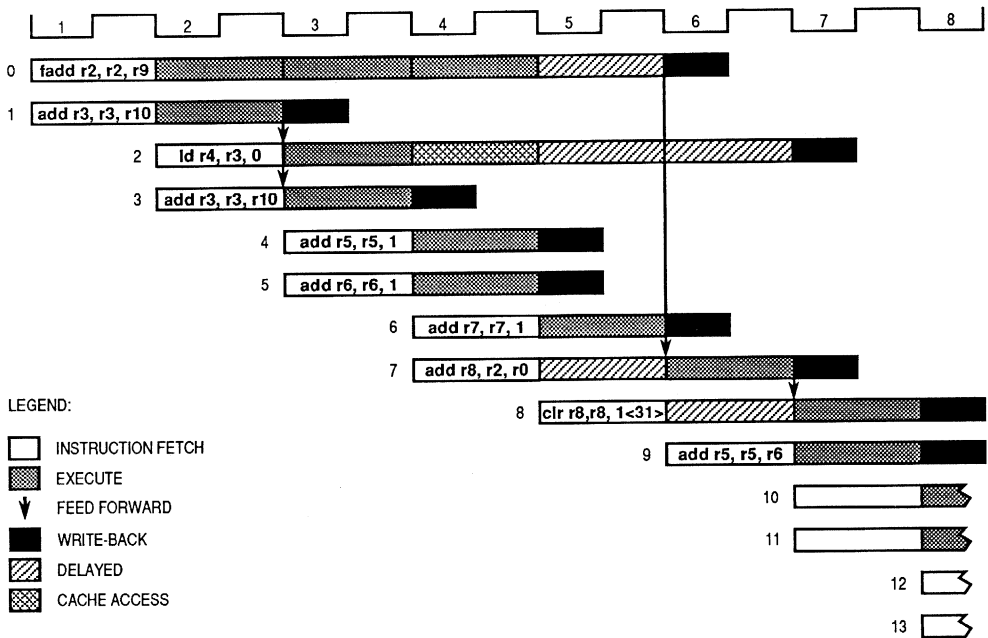
Figure 9-26. Write-Back Arbitration Timing

9.3.2.3.8 Load/Store with Extended Operands Example.

The width of the data path to/from the cache is 64 bits. Therefore, operations with 80-bit double-extended-precision operands require two clock cycles for the data unit to perform cache accesses. The data unit accepts and delivers double-extended-precision operands to the extended register file in a single clock. In Figure 9-27, a **st.x** operation (instruction 1) is issued during the same clock cycle as an **fadd.x** operation (instruction 0), even though the **st.x** is dependent on the results of the **fadd.x**. After being issued, the **st.x** instruction waits in the store reservation station for the results of the **fadd.x** instruction. The **st.x** operation is delayed in the store reservation station until clock 5, when the result of the **fadd.x** instruction is available.

The first **ld.x** operation (instruction 2) is fetched and immediately executed since it has no data dependencies and no address conflicts with the pending **st.x** operation in the store reservation station. The second **ld.x** operation (instruction 3) is issued, but it is delayed in the load buffer. It is allowed to execute when the first **ld.x** instruction completes because the pending **st.x** is still aligning its data for the write. When the **st.x** operation has finished aligning its data and is ready to write to the data cache, it is delayed by instruction 3, which is accessing the cache.

MC88110 USER'S MANUAL   **MOTOROLA**

The second **fadd.x** operation (instruction 4) is delayed until clock 6 because it has a data dependency (**x2**) on instruction 2. Notice that the cache access periods for the double-extended-precision memory operations are 2 clock cycles as explained previously. The extra clock cycle of cache access is the only difference between the timing for double-extended-precision data operations and the timing for single- and double-precision data operations.
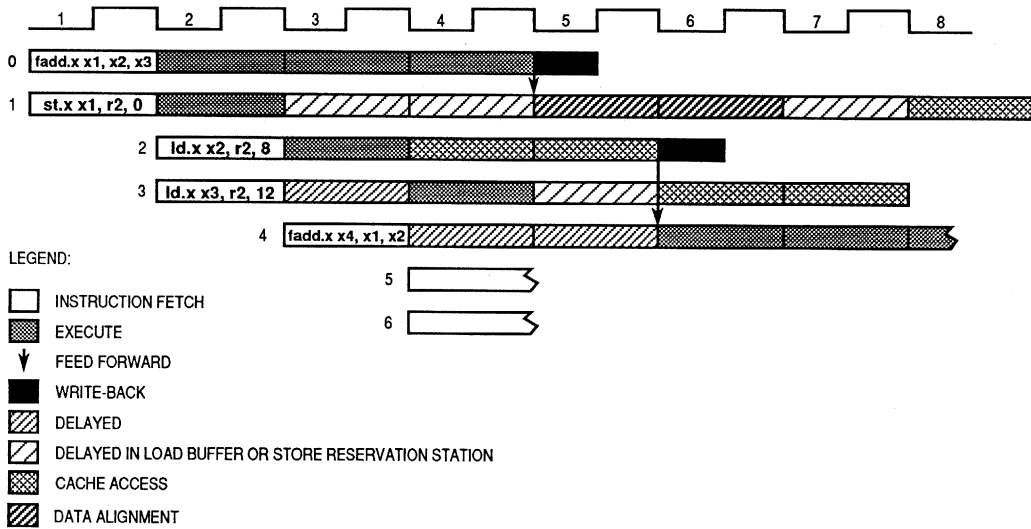
Figure 9-27. Load/Store with Extended Operands Timing

9.3.2.3.9 I/O Serialization Example. This example illustrates using trap instructions to force the MC88110 to serialize bus transactions, which can be useful in systems requiring I/O bus transactions to occur in strict program order. Trap instructions will not be executed until all previously issued instructions have completed; therefore, a trap instruction can be inserted before a load or store instruction to guarantee that the load or store will not execute out of order with respect to other loads or stores. The **tb1** 0, **r0**, 0 instruction is recommended for this use, because it will force the machine to serialize without causing any other side effects.

In this example (see Figure 9-28), a **st** instruction is issued but must wait in the store reservation station for data from a previous instruction (instruction 0) which is already in progress. Normally, the **ld** operation (instruction 3) would be issued to the data unit and be granted access to the data cache during clock 4, ahead of the **st** at instruction 1. If the **ld** and **st** were both to miss the data cache, which would be the case for an address to an I/O device declared uncacheable, then the specified bus transactions would execute out of order. But as shown, the trap instruction forces the machine to serialize before any other instructions are issued. Note that the MC88110 requires one clock cycle before and one clock cycle after the execution of the trap instruction to serialize the machine. Therefore, the **ld** at instruction 3 is delayed until clock 11, when the store and trap operations have completed and the machine is synchronized.

The use of the trap instruction could be avoided by setting the serialize memory bit (SRM) in the PSR. When this is done, the execution of the **ld** at instruction 3 would have the same effect as the trap instruction (i.e., the machine would fully serialize before the **ld** was allowed to issue). The use of the trap instruction is recommended when possible because only the **ld** and **st** instructions must run in strict program sequence. Operation of other **ld** and **st** operations can then proceed normally.

LEGEND:
- ☐ INSTRUCTION FETCH
- ▨ EXECUTE
- ▼ FEED FORWARD
- ■ WRITE-BACK
- ▨ DELAYED
- ▨ DELAYED IN STORE BUFFER
- ▨ CACHE ACCESS
- ▨ DATA ALIGNMENT
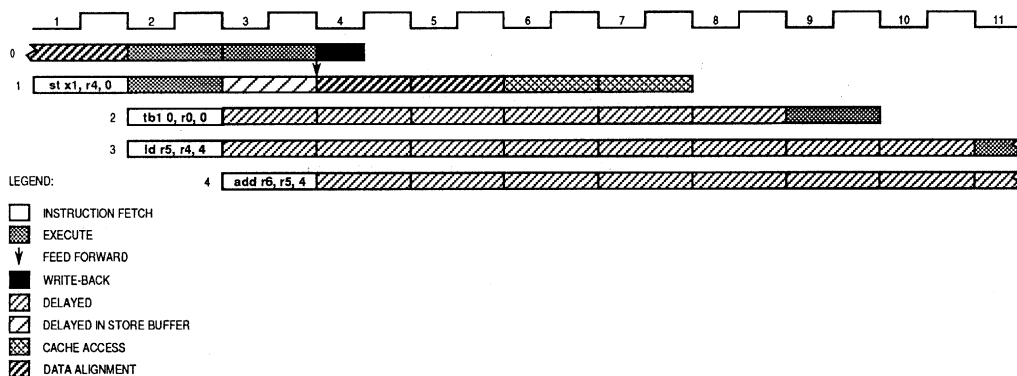
Figure 9-28. I/O Serialization Timing

9.3.2.3.10 Touch Load Operation Timing Example. Touch load operations provide the opportunity for other data instructions to steal cycles from the data cache after the copyback (caused by the touch load) is complete (if a copyback is required) and before the data is received from the bus to load the cache. Accesses to the data cache during a touch load operation are called decoupled cache accesses. Refer to section **9.3.2.1 Decoupled Cache Accesses** for a detailed description of the decoupled cache access feature of the MC88110.

A 2/1/1/1 memory transaction without a copyback will present a one-cycle opportunity for cache access under a touch operation while a 4/1/1/1 allows three. Data cache access is not permitted once the actual data transfer begins so, for example, a 4/2/2/2 transaction provides the same number of cycles of decoupled access as a 4/1/1/1.

Figure 9-29 shows an example of a 4/2/2/2 bus transaction with no touch-induced copyback. The first **ld** (instruction 0) is a touch load executed to bring data into the data cache for future use. The **ld** at instruction 2 is able to access the cache in clock 5, whereas if instruction 0 had been a regular **ld** then it would not have been granted access to the data cache until clock 15. Additionally, instruction issue would have halted on clock 6 due to the data dependency that instruction 8 has on the **ld** (instruction 2). Use of the touch load provides a mechanism for doing useful data-access work during cache misses, thus avoiding the performance degradation typically associated with long latency memory systems.
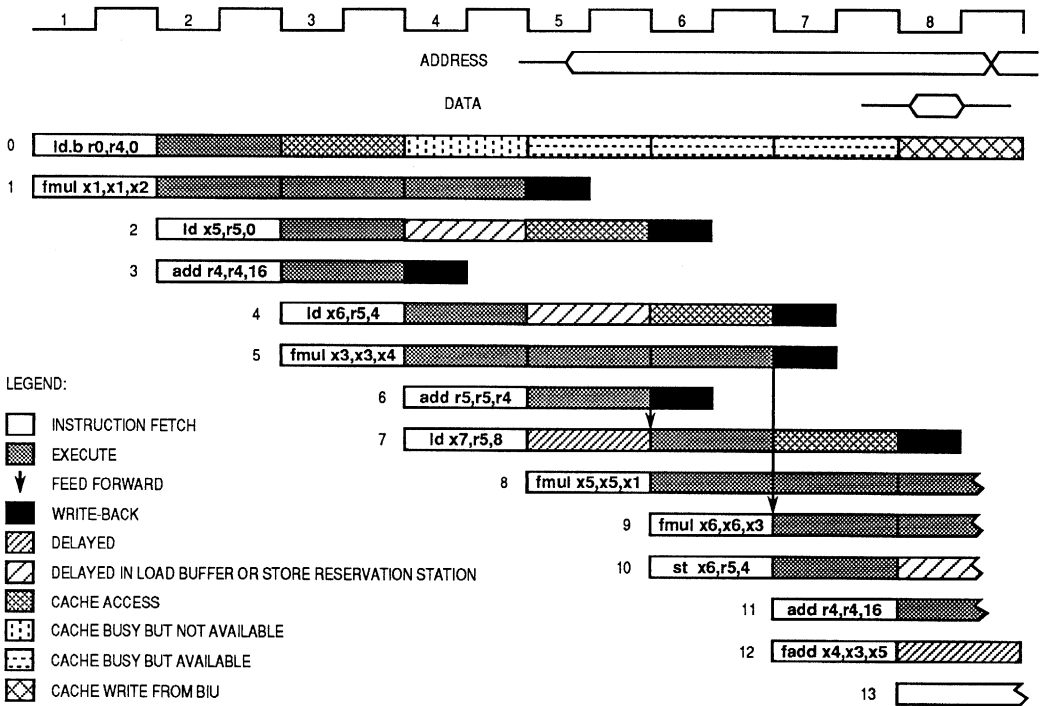
Figure 9-29. Touch Load Operation Timing

LEGEND:

☐	INSTRUCTION FETCH
▨	EXECUTE
↓	FEED FORWARD
■	WRITE-BACK
▨	DELAYED
▨	DELAYED IN LOAD BUFFER OR STORE RESERVATION STATION
▨	CACHE ACCESS
⊡	CACHE BUSY BUT NOT AVAILABLE
⊡	CACHE BUSY BUT AVAILABLE
⊠	CACHE WRITE FROM BIU

9.3.3 Multi-Cycle Execution Unit Timing

There are three multi-cycle execution units in the MC88110: the floating-point add unit, the multiply unit, and the divide unit.

The floating-point adder is a three-stage, fully pipelined design, capable of accepting one single-, double-, or double-extended-precision addition operation every clock. The floating-point adder requires 3 clocks to complete execution. The instructions executed by the floating-point adder are **fadd**, **fsub**, **fcmp**, **fcmpu**, **fcvt**, **int**, **nint**, **trnc**, and **flt**.

The multiplier is also a three-stage, fully pipelined design, capable of accepting one single-, double-, or double-extended-precision multiplication operation every clock with a latency of 3 clocks. The multiplier is shared between floating-point, integer, and graphics operations. The instructions executed by the multiplier are **fmul**, **mulu**, **muls**, and **pmul**.

The divider is a nonpipelined, iterative design which produces exact IEEE results that require no software modifications. The divider is shared between floating-point and integer operations. The instructions executed by the divider are **fdiv**, **divs**, **divu** and **divu.d**. The performance of the divider is dependent on the precision and type of the operands. The MC88110 executes signed integer divide instructions with negative operand(s) directly in hardware.

Table 9-4 shows the latencies for MC88110 floating-point operations.

**Table 9-4. Floating-Point Execution Timings
in Clock Cycles**

Instruction	Size			Execution Unit	Latency
	32 .s	64 .d	80 .x		
fadd,fsub	•	•	•	Floating-Point Add	3
fcmp, fcmpu	•	•	•	Floating-Point Add	1
fmul	•	•	•	Multiply	3
fcvt	•	•	•	Floating-Point Add	3
flt	•	•	•	Floating-Point Add	3
int,nint,trnc	•	•	•	Floating-Point Add	3
mov g <-> x	•	•		Instruction	1
mov x <- x			•	Instruction	1
fldcr, fstcr, fxcr				Instruction	Serialize + 2
fdiv	•			Divide	13 (3*)
fdiv		•		Divide	23 (3*)
fdiv			•	Divide	26 (3*)
divs, divu				Divide	18 (5*)
muls, mulu, mulu.d				Multiply	3
fsqrt	•	•	•	Trap	N/A

* If either operand is 0

9.3.3.1 FLOATING-POINT ADD AND MULTIPLY TIMING EXAMPLE.

In this example (see Figure 9-30), two floating-point instructions issue in clock 2—one to the floating-point adder and the other to the multiplier. During clock 3, two **fadd** instructions attempt to be issued but since each execution unit can only accept one instruction per clock, the second **fadd** is delayed until the next clock cycle. Instruction 4 can not be issued in clock 4 because of a data dependency on instructions 0 and 1 until clock 5. Instructions 0, 1, 6, and 7 show that **fadd** and **fmul** instructions may be issued simultaneously and in either order. Since both floating-point multiply instructions and integer multiply instructions use the multiply unit, instruction 9 is delayed by one clock while the **fmul** at instruction 8 is issued.

MC88110 USER'S MANUAL
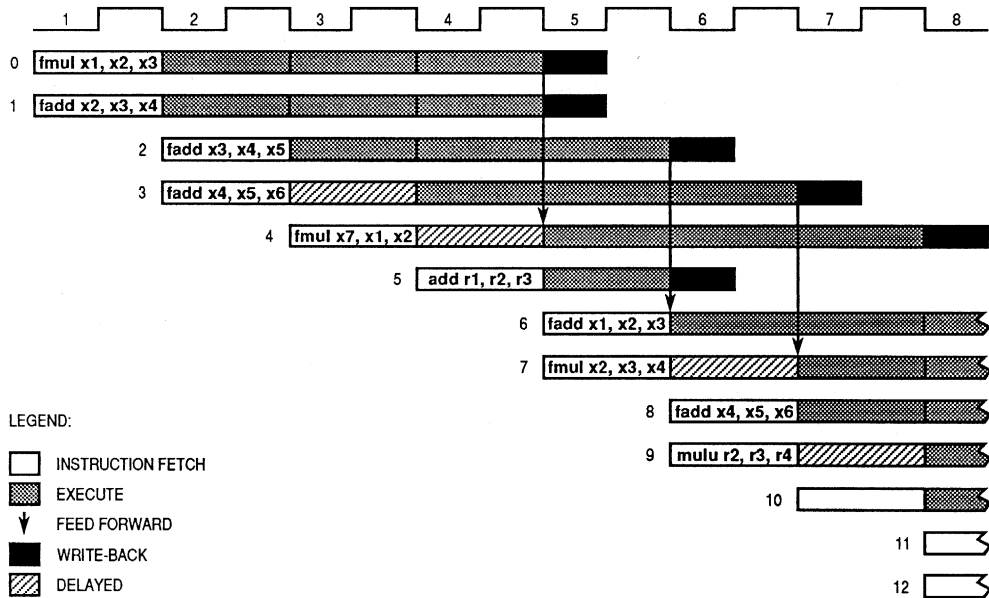
Figure 9-30. Floating-Point Add and Multiply Timing

9.3.3.2 DIVIDE TIMING EXAMPLE. In this example (see Figure 9-31), during clock 2, an **fdiv** instruction is issued and begins execution. Other instructions continue to execute simultaneously with the divide until clock 13 when an integer **div** instruction attempts to be issued. At this point, since the divider is nonpipelined, the sequencer finds the divider busy and stalls issue until the previous divide finishes on clock 15. This is also the first clock in which an instruction with a data dependency on the **fdiv** can issue, as demonstrated by instruction 13.

9
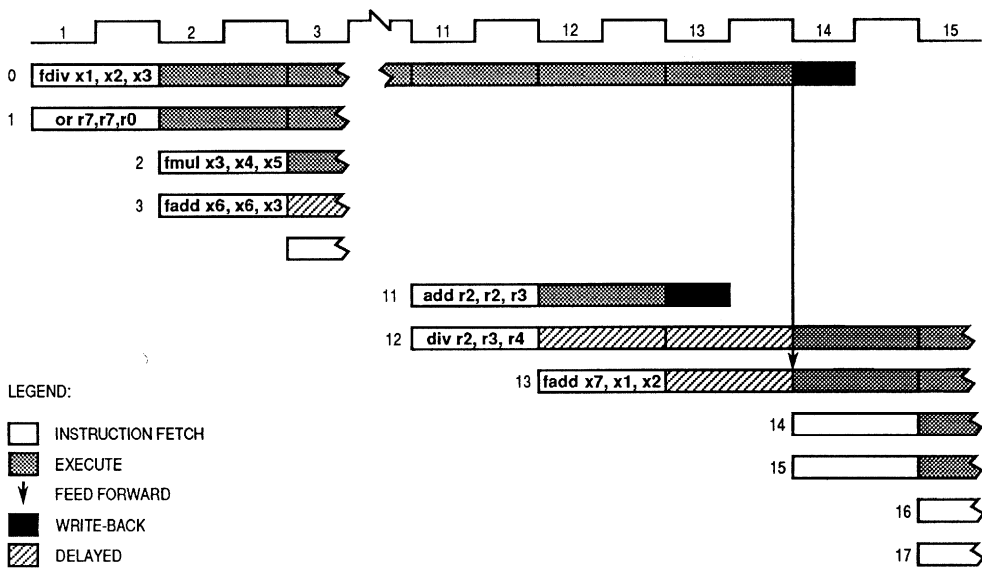
Figure 9-31. Divide Timing

9.3.4 Instruction Unit (Flow Control) Execution Timing

Flow control operations (jumps, branches and traps) are typically expensive to execute in most machines because they disrupt normal flow in the instruction stream. When a change in program flow occurs, the instruction pipeline must be reloaded with the target instruction stream. During this time, bubbles can be introduced into the instruction stream. However, since all of the execution units on the MC88110 operate independently, previously issued instructions will continue to execute while the new instruction stream makes its way to the issue stage of the instruction pipeline.

Design strategies such as delayed branching, the target instruction cache (TIC), and static branch prediction help minimize the penalties associated with branch instructions in the MC88110; therefore, the timing for branch instruction execution is determined by many factors including the following:

- Whether or not the branch is taken.
- Whether or not the delayed branch option (**.n**) is specified.
- Whether the branch issues from the first or second issue slot.
- Whether or not the first two instructions of the target instruction stream are in the TIC (TIC hit).
- Whether or not the target instruction stream is in the instruction cache.
- Whether the branch is predicted or unpredicted.
- Whether the prediction was correct or incorrect.

Table 9-5 lists the flow control instructions and the penalties associated with the execution of these instructions. The causes of these penalties are explained in the following paragraphs.

Table 9-5. Flow Control Instruction Execution Penalties

Instruction	Position In Issue Pair		Not Taken	Taken	
	1st	2nd		TIC Hit	TIC Miss
Branch and Jump Instructions	1st	2nd	Number of bubbles introduced into instruction stream.		
jmp, jsr	•		—	—	3
		•	—	—	2
jmp.n, jsr.n	•		—	—	2
		•	—	—	1
br, bsr	•		—	1	3
		•	—	0	2
br.n, bsr.n	•		—	0	2
		•	—	1	1
bb0, bb1, bcnd	•		1	1	3
		•	0	0	2
bb0.n, bb1.n, bcnd.n	•		0	0	2
		•	0	1	1
Trap Instructions			Latency in clock cycles.		
tb0, tb1, tcnd	•		Serialize+1	—	Serialize + 3
tbnd	•		1	—	Serialize + 3
rte	•		—	—	Serialize + 3
Control Register Instructions			Latency in clock cycles.		
ldcr	•		—	—	Serialize + ≥2
stcr	•		—	—	Serialize + ≥2
xcr	•		—	—	Serialize + ≥2

9

9.3.4.1 DELAYED BRANCHING. The instruction issue opportunity immediately following a flow control instruction is called a delay slot (see Figure 9-32). If the flow control instruction is the first instruction in an issue pair, then the second slot in the issue pair would be the delay slot. If the flow control instruction is the second in an issue pair, then the first issue slot of the next clock cycle is the delay slot.
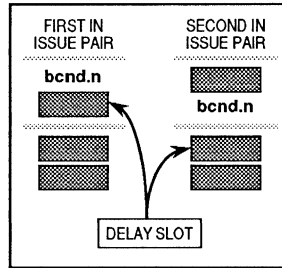
LEGEND:

.... CLOCK BOUNDARIES

▨▨▨ INSTRUCTION-ISSUE OPPORTUNITY

Figure 9-32. Branch Delay Slot

When a branch instruction is encountered, the clock cycle following the branch is only used for refilling the instruction pipeline (i.e., no instruction is issued). However, it is possible to issue an instruction during the delay slot by using the delayed branching option. The delayed branching option (.n) can be specified for all branch and jump instructions. Delayed branching allows a useful instruction immediately following a branch or jump instruction to be unconditionally executed during the penalty time incurred by the disruption in program flow.

An instruction that might normally reside before the branch can be placed in the delay slot. For example, the results from a loop can be stored during the delay slot since this operation should occur whether or not the loop is executed again. Another option is to place the first instruction of the target instruction stream in the delay slot, provided that executing this instruction does not affect the program if the branch is not taken.

Although the MC88110 is capable of issuing two instructions per clock cycle, the delayed branch option only allows a single instruction to be issued during the penalty incurred from a flow control instruction. Since only one instruction can be issued during the one clock cycle of penalty, a single bubble may still be introduced into the instruction pipeline.

NOTE

Delayed branching was developed for the MC88100 to help reduce penalties associated with changes in program flow; however, in future machines, delayed branching may be implemented in software and may actually reduce performance. Therefore, it is recommended that new software (e.g., compilers) avoid delayed branching.

9.3.4.2 TARGET INSTRUCTION CACHE. The target instruction cache (TIC) allows the first two instructions at the target address of a branch instruction to be executed while the instruction pipeline is being refilled. The TIC can be used in place of, or in conjunction with, the delayed branching option.

As shown in Figure 9-33, the TIC is a fully associative, 32-entry, logically addressed cache which must be flushed on a context switch. Each entry in the cache can maintain the first two instructions of a branch target instruction stream and a 31-bit logical address tag. The 31-bit logical address tag holds the 30-bit address of the branch instruction and a user/supervisor bit.
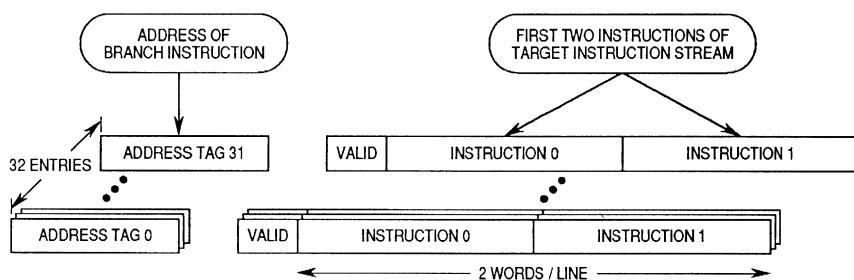
Figure 9-33. The Target Instruction Cache (TIC)

One entry in the TIC is automatically filled when a branch is taken (assuming all conditions are met). Each time the branch instruction at that address is prefetched, the TIC is accessed (i.e, a TIC hit occurs) in parallel with the decode of the branch instruction.

When a TIC hit occurs and the branch is taken, the two instructions in the TIC entry are ready to execute on the next clock cycle. The first instruction fetched from the TIC must be placed in the first-issue slot of the clock cycle. If the branch is not taken, the TIC entry remains valid.

If a TIC miss (a branch instruction is encountered that is not already in the TIC) occurs, the branch is taken, and there are no empty (invalid) entries in the TIC to accept a new entry, then one of the valid entries is chosen for replacement using a FIFO replacement policy. If the branch is not taken, no entry is allocated in the TIC.

There are several conditions when the TIC is not used to accelerate a flow control operation. First, jump instructions are not accelerated by the TIC. In other words, the first two instructions at the target instruction stream of a jump operation are not entered in the TIC. Second, when a delayed branch and the instruction in its delay slot are not issued during the same clock, the first two instructions at the target instruction stream are not entered in the TIC. Third, when two instructions at the target instruction stream can not be fetched together, those two instructions will not be entered in the TIC. For example, if the target address of a branch operation points to the last word on a cache line, only one instruction can be fetched, thus no instructions at the target instruction stream will be entered in the TIC. It is important to note that this does not depend on whether the two instructions can actually issue together. For more information on instruction fetch timing, refer to section **9.2.1.1 Instruction Cache Timing**.

The advantages of the TIC depend on whether a branch instruction resides in the first or second issue slot as well as whether the delayed branching option was used. The following examples show how instruction issue is affected by the TIC when a branch instruction is in the first and second issue positions for both delayed and nondelayed branches.

9.3.4.2.1 Delayed Branching Example. When the delayed branching option is used and the branch instruction resides in the first issue slot, the instruction following the branch is placed in the second issue slot. This is possible since delayed branching causes the instruction in the second issue slot to be issued whether or not the branch is taken. The MC88110 requires an additional clock cycle to refill the instruction pipeline with the branch target instruction stream. During this clock cycle, if there is a TIC miss, two bubbles are created (see Figure 9-34). However, when there is a TIC hit for this branch instruction, these two bubbles are filled by the two instructions stored in the TIC. Thus, when delayed branching is used in the first issue slot and a TIC hit occurs for the branch, there are no interruptions in the execution of instructions.

A different situation occurs when a branch instruction using delayed branching resides in the second issue slot. The instruction following the branch is issued alone during the next clock cycle (clock cycle 1); thus, only one opportunity to issue an instruction is lost (see Figure 9-34). When there is a TIC miss the instruction pipeline is refilled in the clock cycle after the branch (clock 1). Execution continues in the second clock cycle after the branch (clock cycle 2). When there is a TIC hit for this branch, the two instructions at the target address are ready to execute during clock cycle 1; however, the instruction which was placed in the delay slot must be placed in the first issue slot of clock cycle 1. Since the first instruction fetched from the TIC must also be placed in the first issue slot of a clock cycle, the instructions from the TIC cannot be issued until clock cycle 2 after the delay slot instruction has been issued. This results in a bubble after the delay slot instruction. Therefore, in the specific case where delayed branching is used and the branch instruction falls in the second issue slot, the TIC provides no advantage. Since the TIC provides no advantage in this specific case, if a delayed branch and the instruction in its delay slot are not issued during the same clock cycle, then that branch instruction will not be entered in the TIC.
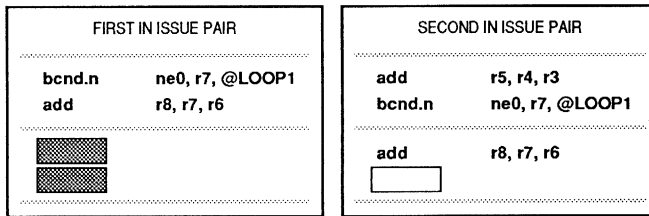
FIRST IN ISSUE PAIR	SECOND IN ISSUE PAIR

```
bcnd.n    ne0, r7, @LOOP1      add       r5, r4, r3
add       r8, r7, r6           bcnd.n    ne0, r7, @LOOP1

                               add       r8, r7, r6
```

LEGEND:

········· CLOCK BOUNDARIES

▒▒▒▒ BUBBLES WHICH OCCUR AS THE RESULT OF A TIC MISS

☐ BUBBLES WHICH OCCUR REGARDLESS OF A TIC HIT OR MISS

Figure 9-34. Effect of the TIC When Delayed Branching Is Used

9.3.4.2.2 Nondelayed Branching Example. When the delayed branching option is not used and the branch instruction resides in the first issue slot, no instruction is issued from the second issue slot. Additionally, the MC88110 uses another clock cycle to refill the instruction pipeline with the target instruction stream. During this clock cycle, if a TIC miss occurs, two more opportunities to issue instructions are lost. Thus, a total of three bubbles occur when a nondelayed branch is prefetched into the first-issue slot and a TIC miss occurs (see Figure 9-35). When there is a TIC hit for this branch case, the second issue slot remains vacant (the first issue slot contains the branch instruction), resulting in a loss of one opportunity to issue an instruction. However, the instructions fetched from the TIC are ready to execute on the next clock cycle. Thus, when there is a TIC hit, and a nondelayed branch is prefetched into the first issue slot, only one opportunity to issue an instruction is lost.

When a nondelayed branch instruction is prefetched into the second issue slot, and a TIC miss occurs, the MC88110 uses one additional clock cycle to refill the instruction pipeline. This delay introduces two bubbles into the instruction pipeline (see Figure 9-35). When there is a TIC hit for this branch case, the two instructions from the TIC will be ready to execute during the next clock, thus no bubbles occur. This case (branch instruction in second issue slot with no delayed branching) is important because it shows that the TIC provides the opportunity to execute a nondelayed branch instruction without incurring a penalty.

9

LEGEND:

······· CLOCK BOUNDARIES

▓▓▓ BUBBLES WHICH OCCUR AS THE RESULT OF A TIC MISS

☐ BUBBLES WHICH OCCUR REGARDLESS OF A TIC HIT OR MISS
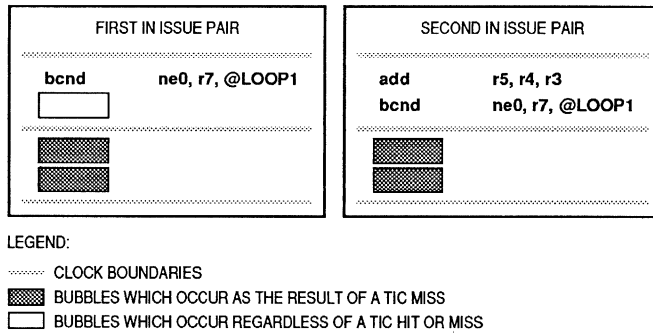
Figure 9-35. Effect of the TIC When Nondelayed Branching Is Used

Table 9-6 summarizes the penalties incurred by executing branch instructions when the branch is taken. It is important to note how the penalties vary with respect to TIC hits versus TIC misses, as well as with respect to whether or not delayed branching is used.

Table 9-6. Penalties Incurred by Branch Instructions When the Branch Is Taken

TIC Hit/Miss	Delayed Branch	Nondelayed Branch
Branch Instruction in First Issue Slot		
TIC Miss	2 Bubbles	3 Bubbles
TIC Hit	0 Bubbles	1 Bubble
Branch Instruction in Second Issue Slot		
TIC Miss	1 Bubble	2 Bubbles
TIC Hit	1 Bubble	0 Bubbles

9.3.4.3 STATIC BRANCH PREDICTION. Static (compiler-directed) branch prediction is a mechanism by which software (e.g. compilers) can give a hint to the machine hardware on which direction the branch is likely to go. When a branch instruction encounters a data dependency, the branch instruction is issued to the branch reservation station where it waits for the required source operand to become available. Rather than stalling instruction issue until the source operand is ready, the MC88110 predicts which path the branch instruction is likely to take, and instructions are fetched and executed along that path. When the branch operand becomes available, it is forwarded to the instruction unit and the branch is evaluated. If the predicted path was correct, program flow continues along that path; otherwise, the processor backs up using the history buffer, and program flow resumes along the correct path.

There is a scenario where a conditional branch, whose source data is not available, will not be predicted on the MC88110. If the data which is being tested by the branch operation is not available and will not be transmitted on the destination bus in the same format as the waiting branch needs it, instruction issue will stall until the data becomes

available. For example, if a conditional branch is testing **r6** and the data in **r6** is not yet available, another check is made before allowing the branch operation to issue to the branch reservation station. If **r6** is the destination of a double-precision operation, instruction issue will stall until **r6** has been written back into the register file and read back out again for the waiting branch operation.

The MC88110 has three conditional branch instructions: **bb0** (branch on bit clear), **bb1** (branch on bit set), and **bcnd** (conditional branch). The static branch prediction mechanism is defined in the MC88110 to maximize performance of conditional branches. The implementation of branch prediction is not a change from the MC88100 instruction set but is simply a convention which the compiler can use to optimize branch performance on the MC88110.

The preferred direction of program flow (i.e., taken or not taken) for each branch instruction is predicted based on hints provided by the software. Table 9-7 shows how the MC88110 interprets the **bb0**, **bb1**, and **bcnd** instructions for static branch prediction purposes. When the MC88110 encounters a **bb1** instruction, the branch is predicted to be taken. When a **bb0** is encountered, the branch is predicted not to be taken. How the MC88110 interprets the **bcnd** instruction depends on which instruction encoding variation is used: if the condition being tested for is either greater-than-zero, greater-than-or-equal-to-zero, or not-equal-to-zero (i.e., bit 21 in the instruction encoding is set), the **bcnd** instruction is predicted to be taken. Conversely, if bit 21 is clear, the branch is predicted to not be taken. This convention is consistent with the common use of **bcnd** as the loop test-and-branch or null-check operation.

Table 9-7. Branch Predictions for Conditional Branch Instructions

Instruction			Prediction
bcnd	rS1 Condition	Bit 21	
	=0	0	Not Taken
	≠0	1	Taken
	>0	1	Taken
	<0	0	Not Taken
	≥0	1	Taken
	≤0	0	Not Taken
bb1			Taken
bb0			Not Taken

Branch instructions whose source data is not available and therefore must be issued to the branch reservation station are said to be predicted. When the MC88110 takes a predicted branch which later turns out to have been incorrect (i.e., the processor conditionally executed the wrong path), that branch instruction is said to be mispredicted. Branch instructions which are issued with source data already available (and thus do not have to wait in the branch reservation station) are said to be

unpredicted. Instructions issued as a result of a predicted branch are said to be issued conditionally, and are tagged as such until the branch is resolved.

When a branch is resolved and it has been correctly predicted, the conditional tag on all instructions issued conditionally is cleared, and instruction execution continues without interruption along the predicted path. In the event that a branch is mispredicted, the instruction unit causes all execution units to flush all instructions in their respective pipelines that are tagged as conditional. In addition, the instruction unit then reverses the effects of any conditionally issued instructions that have completed execution, thereby returning the machine to its state at the time the branch was issued. Execution then resumes down the correct path. The mechanism used to reverse the effects of mispredicted branches is the history buffer. A detailed description of the history buffer can be found in **Section 7 Exceptions**.

If an instruction fetch is attempted conditionally and misses both the TIC and instruction cache then a bus transaction to read the instruction cache line from memory is initiated.

The MC88110 places the following restrictions on the execution of conditionally issued instructions:

- The MC88110 conditionally issues instructions down only one predicted path at a time; instruction issue will stall if an attempt is made to issue a predicted branch instruction while instructions are being issued conditionally. Unpredicted branches are allowed to be issued while instructions are being issued conditionally.

- **st** instructions are issued conditionally to the store reservation station but are not granted access to the data cache or the external bus until the branch is favorably resolved.

- **ld** instructions are issued conditionally to the load buffer and execute normally if they hit the address translation cache (ATC) and data cache. If a conditionally issued **ld** instruction causes a miss in the ATC or data cache, then further execution of that instruction is stalled until the branch is favorably resolved.

- The number of instructions which can be executed conditionally after the issue of a predicted branch instruction is limited by the depth of the history buffer (12 instructions).

The following is an example of static branch prediction:

1. The MC88110 encounters a **bb1** instruction which cannot be executed because of a scoreboard hold on its source register.

2. The MC88110 always predicts **bb1** to be taken, so the instruction is issued to the branch reservation station while the processor continues instruction execution at the branch target address.

3. When the source data for the branch instruction becomes available, the initial **bb1** instruction is executed. If it was correctly predicted, then the **bb1** has been successfully executed without stalling the instruction pipeline while source data becomes available. If the branch was mispredicted, the MC88110 reverses its state to when the **bb1** instruction was issued.

The MC88110 is able to reverse its machine state at a rate of two instructions per clock cycle; therefore, if 8 instructions were executed conditionally, and the branch was mispredicted, the MC88110 will require 4 clock cycles to return to the correct state.

9.3.4.4 UNPREDICTED BRANCH TIMING EXAMPLES. The following notation is observed in the timing diagrams for the examples in the following paragraphs:

Nx: An instruction labeled with an "N" is the next sequential instruction following a branch in the program.

Tx: An instruction labeled with a "T" is the target instruction of a branch—i.e., the instruction stream to which control will be transferred if the branch condition is evaluated to be true.

bxx: A generic label for **br, bb0, bb1,** or **bcnd**.

bxx.n: A generic label for a branch instruction with the delayed branching option (i.e., **br.n, bb0.n, bb1.n,** or **bcnd.n**).

9.3.4.4.1 Unpredicted Branch Not Taken Example. This example assumes that the conditions have been resolved for all branch instructions by the time they are issued—i.e., the branches are unpredicted. In general, branches which are not taken require only the instruction issue slot they occupy and do not introduce additional bubbles into the instruction pipeline. One exception to this is when a nondelayed branch is issued as the first instruction in an issue pair. This exception is illustrated by instruction 0 in Figure 9-36. Notice that instruction 1 cannot begin execution on the same clock as the branch (instruction 0). This is because the MC88110 has no way of knowing if the branch will be taken or not and must resolve the branch before any additional instructions can be issued. As a result, one opportunity to issue an instruction is lost.

Instruction 7 is a delayed branch; therefore, instruction 8 is executed whether or not the branch is taken and instruction 8 is issued along with instruction 7. In this case, no opportunities to issue instructions are lost.

9
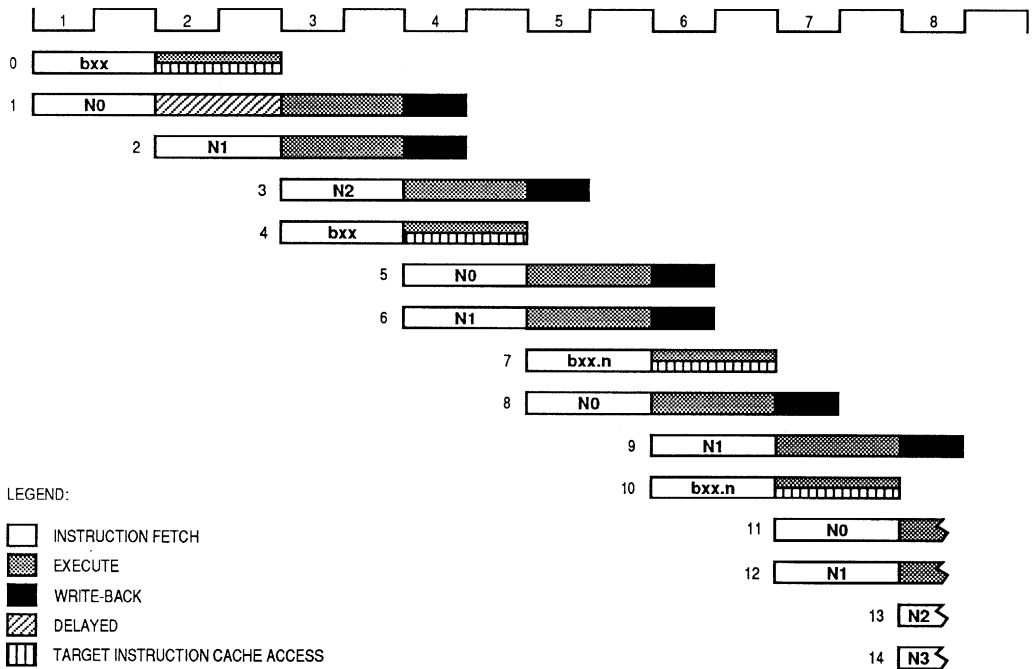
Figure 9-36. Unpredicted Branch Not Taken Timing

LEGEND:

☐ INSTRUCTION FETCH
▨ EXECUTE
■ WRITE-BACK
▧ DELAYED
▥ TARGET INSTRUCTION CACHE ACCESS

9.3.4.4.2 Unpredicted Branch Taken with TIC Miss Example. In this example (see Figure 9-37), instruction 0 is a branch instruction that is issued as the first instruction in an issue pair. In clock 2 the branch is detected in the first issue slot and instruction 1 is unconditionally delayed in case the branch is taken. The TIC is accessed during clock 2 but the target instructions are not found in the cache (TIC miss). Meanwhile the instruction unit, in preparation for the issue of the next instruction pair, has already fetched the next two sequential instructions, N1 and N2. By clock 3 the processor has determined that the branch is to be taken, so N0, N1 and N2 are discarded. At the same time, the branch target address computed during clock 2 is used to fetch the first two instructions of the target instruction stream from the instruction cache. During clock 4, normal execution of the target instruction stream begins. As a result of the branch taken at instruction 0, three bubbles have been introduced into the instruction pipeline.

The second branch in the example (instruction 7) is an example of a taken branch issued as the second instruction in an issue pair. The execution of this branch is similar to that of the first branch but this time there is no post-branch instruction to be issued in the same clock; thus, only two bubbles are introduced into the instruction pipeline.

bxx

0 bxx

1 N0

2 N1

3 N2

4 T0

5 T1

6 T2

7 bxx

8 N0

9 N1

10 T0

11 T1

12

13

LEGEND:

☐ INSTRUCTION FETCH
▨ EXECUTE
■ WRITE-BACK
▨ DELAYED
▨ INSTRUCTION CACHE ACCESS
▥ TARGET INSTRUCTION CACHE ACCESS (MISS)
▨ ABORT
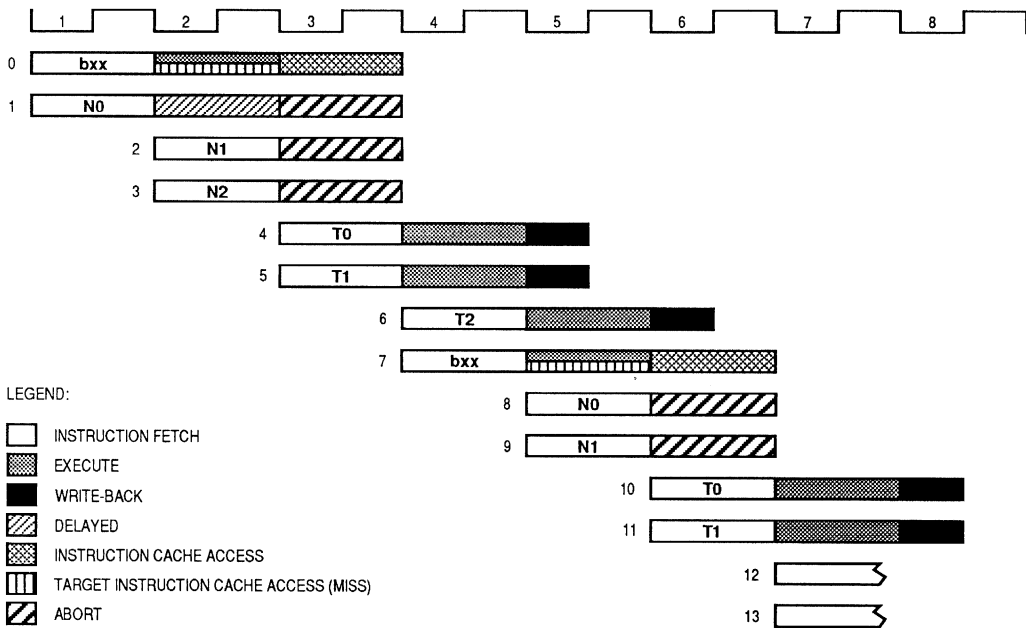
Figure 9-37. Unpredicted Branch Taken with TIC Miss Timing

9.3.4.4.3 Unpredicted Delayed Branch Taken with TIC Miss Example. This example uses the same code sequence as the previous example, except the branch instructions in this example use the delayed branching option (see Figure 9-38).

The first branch operation (instruction 0) is issued in the first issue slot and a TIC miss occurs. Instruction 1 (N0) is executed along with the branch since the delayed branch option (**.n**) has been used. The processor determines that the branch is to be taken, so the next two instructions in the current instruction stream are discarded during clock 2. Also during clock 2, the target instruction stream is computed. The MC88110 begins executing the target instruction stream during clock 3 (instructions 4 and 5). Because of the TIC miss, two opportunities to issue instructions are lost during clock 2.

Another delayed branch occurs during clock 4. This time, the branch is fetched into the second issue slot. Again, there are no entries in the TIC corresponding to this branch instruction (probably because this is the first time this branch instruction has been executed). Since the delayed branch option (**.n**) is used, the next instruction in the stream is issued during the next clock cycle; however, since the delayed branch option only allows the next instruction after a branch to be issued, instruction 9 is aborted during clock 5.

Because of the delayed branch (**.n**) feature, the number of instruction bubbles introduced by each branch in this example have been reduced by one from the nondelayed branches in the previous example. It can be seen from Figure 9-38 that because of the delay slot, the instruction following the branch is always executed and

therefore never has to be delayed or aborted. Delayed branching allows a bubble to be replaced by the issue of an instruction.

It is usually possible to rearrange the code sequence "**br.n**, N0" to be "N0, **br**". It is also possible that both sequences may have the same performance and functionality; thus, in the MC88110, little benefit may result from the use of the delayed branching option. In future implementations, delayed branching may give worse performance than nondelayed branching; therefore, while the 88000 architecture and the MC88110 continue to fully support the delayed branch option, it is recommended that new compilers not use this option and that the use of delayed branching be phased out completely over time. For a more in-depth discussion of the delayed branch option, refer to **9.3.4.1 Delayed Branching** and **9.3.4.2 Target Instruction Cache**.
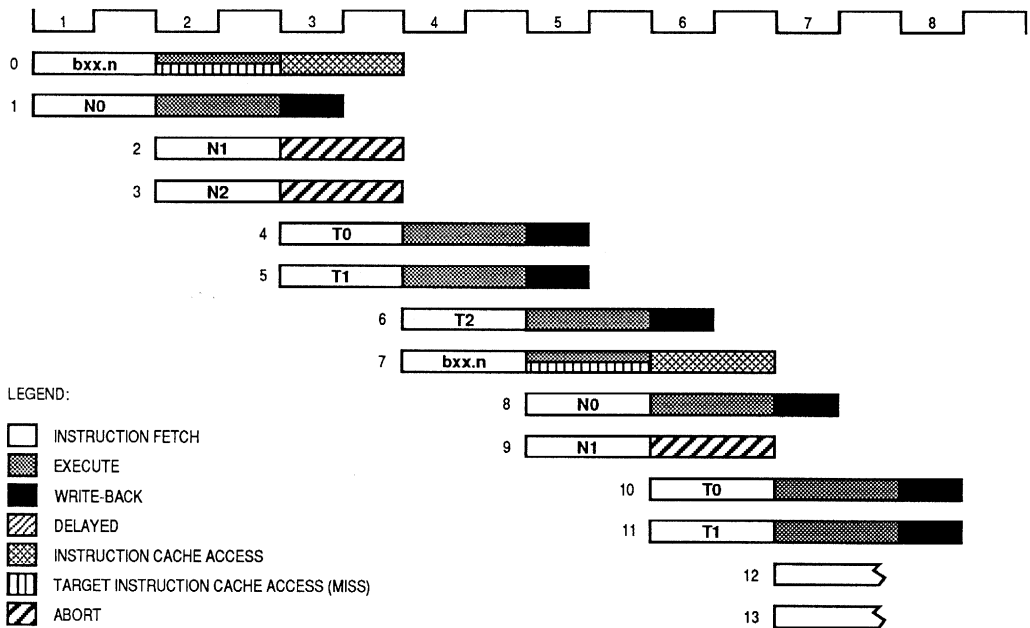
**Figure 9-38. Unpredicted Delayed Branch
Taken with TIC Miss Timing**

9.3.4.4.4 Unpredicted Branch Taken with TIC Hit Example. This example illustrates taken branches which hit in the TIC (see Figure 9-39). A branch instruction is issued on clock 2, and the TIC is accessed using the address of the branch instruction. Since instruction 1 is in the second issue slot and has already been fetched, it is aborted, resulting in one bubble in the pipeline. Thus, the TIC has reduced the latency of the branch by a full clock and has kept two potential bubbles from being introduced into the instruction pipeline. Notice that the target instructions are available a clock earlier when a TIC hit occurs than they are when a TIC miss occurs since the target instructions have to be fetched from the instruction cache in the case of a TIC miss.

The branch at instruction 5, which is issued as the second instruction in an issue pair, does not result in any wasted fetches and thus introduces no bubbles into the pipeline. There are no wasted fetches because the TIC has the first two instructions from the target instruction stream ready in clock 4.

The **bsr** (instruction 8) has the same timing as any other taken branch. This example shows that the write-back of the return address to **r1** occurs in time for the first target instruction to use it without a stall.
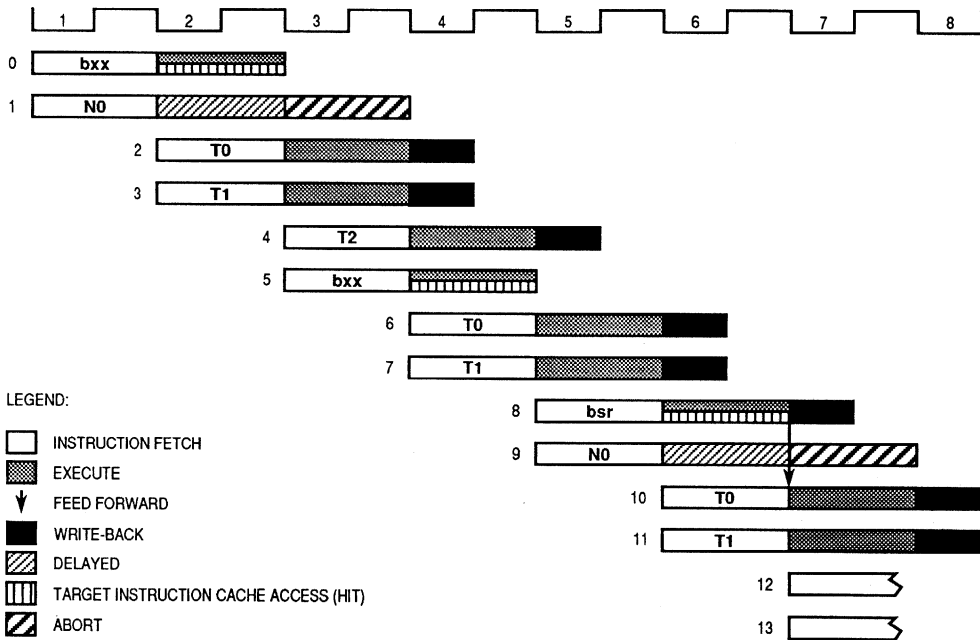
Figure 9-39. Unpredicted Branch Taken with TIC Hit Timing

9.3.4.4.5 Unpredicted Delayed Branch Taken with TIC Hit Example. When a delayed branch is issued as the first instruction in an issue pair, the fetch of the next instruction is not wasted and no pipeline bubbles occur. When a delayed branch is issued as the second instruction in a pair (see instruction 5 in Figure 9-40) the next pair of instructions are fetched from the instruction cache even though the second instruction in the pair will not be issued. For this reason, instruction 7 in Figure 9-40 is not issued, thus introducing one bubble into the instruction pipeline. A nondelayed branch introduces one bubble when issued in the first slot and zero bubbles when issued in the second slot; a delayed branch introduces zero bubbles when issued from the first slot and one from the second.

The timing of a delayed branch to subroutine (**bsr.n**) is the same as other conditional delayed branches when there are no data dependencies on its result. However, when there is a data dependency on the result, one bubble is introduced into the pipeline. For

example, the **bsr.n** at instruction 10 introduces a data dependency (**r1**). When the **bsr.n** is issued, **r1** is marked "busy" in the scoreboard. When the **add** at instruction 11 tries to be issued, it finds **r1** busy and is delayed one clock until **r1** is ready. Thus, an instruction in the delay slot of a **bsr.n** that has a data dependency on the result of the **bsr.n**, will introduce one bubble into the pipeline before receiving the required data. If the **bsr.n** had been issued as the second instruction in an issue pair, no bubble would have occurred.
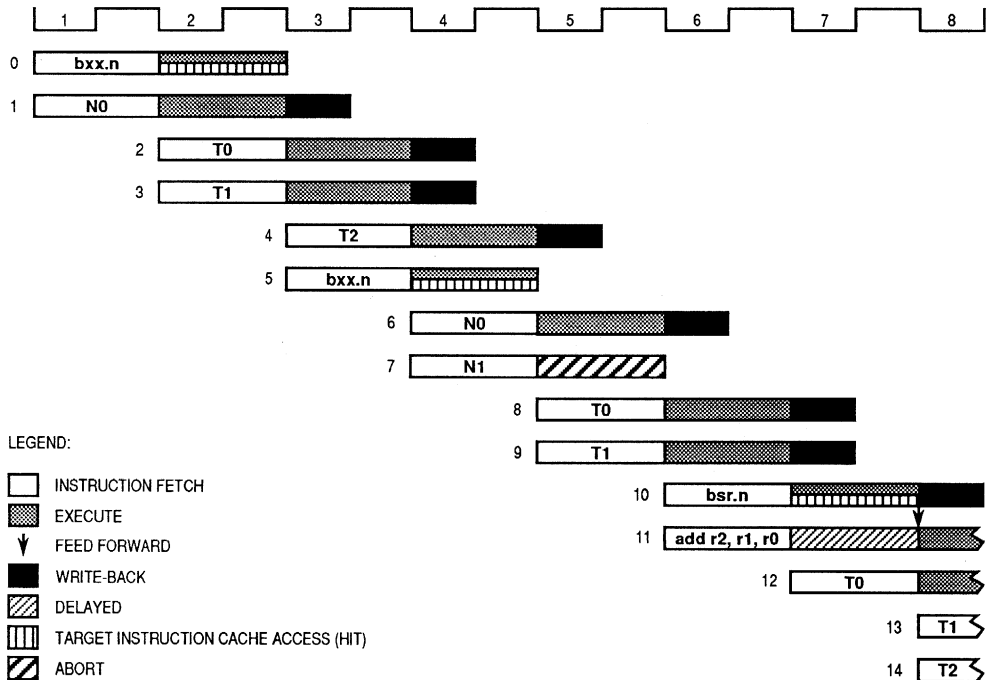
Figure 9-40. Unpredicted Delayed Branches Taken with TIC Hit Timing

9.3.4.5 PREDICTED BRANCH TIMING EXAMPLES. The following paragraphs illustrate various cases of predicted branch timing.

9.3.4.5.1 Predicted Branch Example. Branch prediction does not affect the latency of flow control instructions. Instead, branch prediction is used to accelerate the issue of conditional branches by allowing branch instructions to issue even when the branch condition cannot be evaluated due to a data dependency. The timing of predicted branches after being issued is the same as shown for the unpredicted branches in the preceding pipeline diagrams. The effect of prediction on branch issue is illustrated in the pipeline diagrams which follow.

The prediction of the branch evaluation is based on which branch instruction is used for the operation (refer to **9.3.4.3 Static Branch Prediction** for a detailed explanation of how branch predictions are made and carried out). While the branch waits in the branch reservation station for its source data, instructions along the predicted path are conditionally fetched and executed. If the branch turns out to have been mispredicted, the instruction unit causes all execution units to flush all instructions in their respective pipelines which are tagged as conditional, and the instruction unit then reverses the effects of any conditionally issued instructions which have completed execution and might have erroneously updated the machine state. Execution then resumes down the correct path.

If the MC88110 did not implement branch prediction, execution would proceed as illustrated in the first instruction sequence shown in Figure 9-41. The branch issue would be delayed until clock 3, when the results of the compare instruction are available, and the subsequent issue of instructions down the new instruction stream would not have been able to start until clock 4. With branch prediction, however, the MC88110 can begin execution down the predicted path one clock earlier, thus eliminating two instruction bubbles from the instruction stream.

If the **cmp** instruction was issued as the second instruction in a pair and the dependent branch was issued as the first instruction of the next issue pair, then evaluation of the branch would not have been delayed and branch prediction would not have improved performance.

9
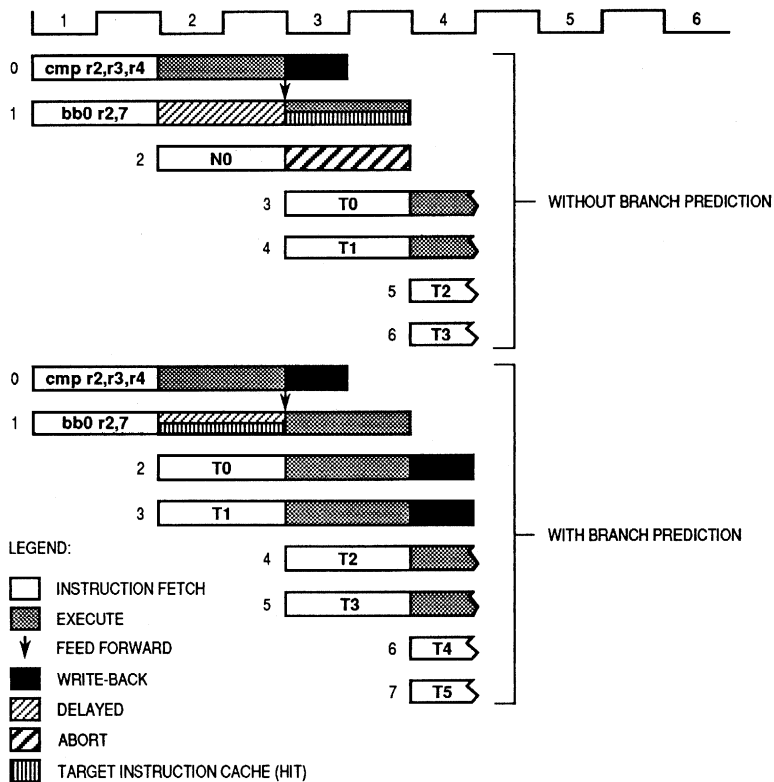
Figure 9-41. Branch Prediction Effect Timing

9.3.4.5.2 Predicted Branch Taken with TIC Hit Example.

Figure 9-42 illustrates the operation of branches which are predicted to be taken. The first branch at instruction 1 shows a correctly predicted branch. The second branch at instruction 7 shows a misprediction. Compared to not having branch prediction at all, the correct prediction saves two instruction bubbles, and the misprediction adds two instruction bubbles. Thus branch prediction provides the same performance as not having branch prediction if the prediction is completely random. If the prediction is more than 50% accurate, prediction provides a net increase in performance.

The prediction mechanism was designed such that flow control will be correctly predicted more than 50% of the time. For example, the majority of branch operations are used for looping. The majority of loops use a counter which is decremented to zero; thus, the loop branch instruction is taken most of the time. Since the **bcnd gt0** instruction is widely used to test if a counter has been decremented to zero, it follows that this instruction is used to predict a change in instruction flow.
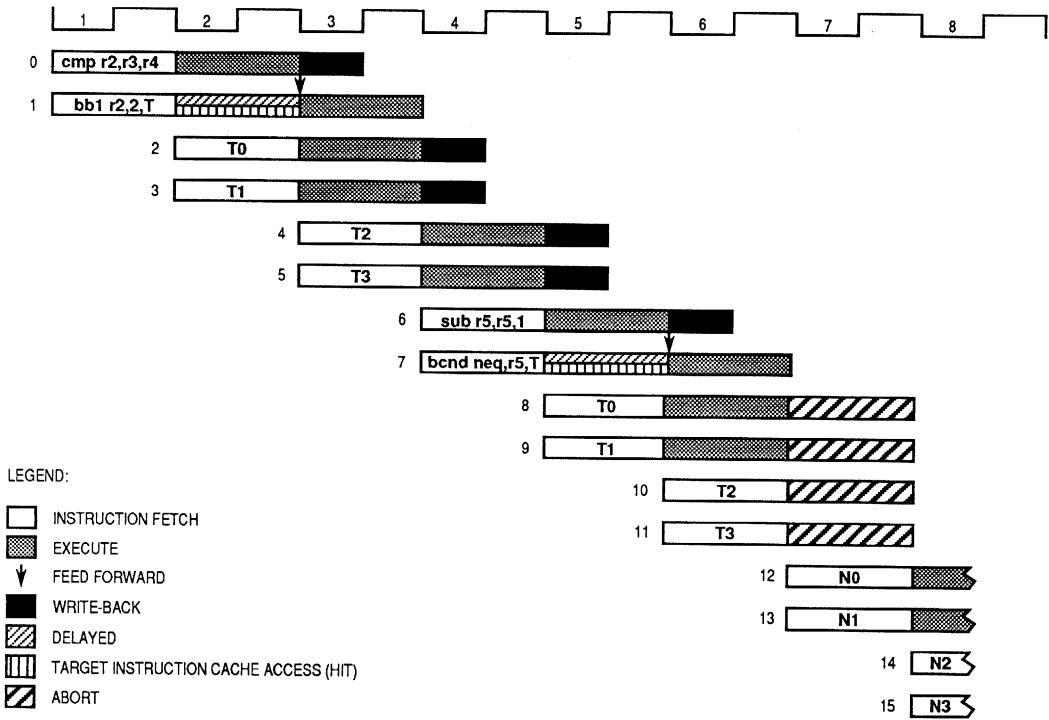
Figure 9-42. Predicted Branch Taken Timing

LEGEND:

- ☐ INSTRUCTION FETCH
- ▨ EXECUTE
- ▼ FEED FORWARD
- ■ WRITE-BACK
- ▨ DELAYED
- ▥ TARGET INSTRUCTION CACHE ACCESS (HIT)
- ▨ ABORT

9.2.4.5.3. Predicted Branch Not Taken with TIC Hit Example. Figure 9-43 shows the case of branches which are predicted not to be taken. The first branch is correctly predicted and the second is mispredicted. The relative performance cost and benefits are the same as for the previous example.
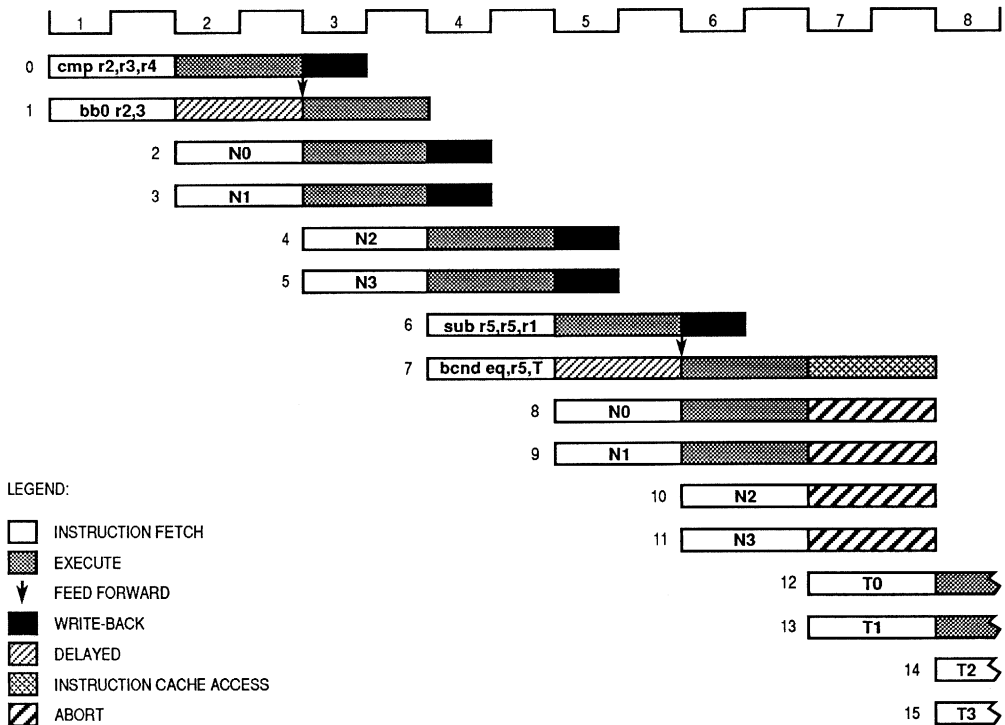
Figure 9-43. Predicted Branch Not Taken Timing

LEGEND:

☐ INSTRUCTION FETCH
▨ EXECUTE
↓ FEED FORWARD
■ WRITE-BACK
▨ DELAYED
▨ INSTRUCTION CACHE ACCESS
▨ ABORT

9.3.4.5.4 Long Latency with Misprediction Example. This example (see Figure 9-44) shows the effect of branch prediction when the availability of the branch source register is delayed for several clocks. The branch at instruction 1 is dependent on data (**r2**) from the **ld** at instruction 0. The **ld** is issued in clock 2 but is delayed in a load buffer (because of a previous **ld** instruction that is not shown). After the branch is issued in clock 2, the branch waits in the branch reservation station for the load to complete. Because the branch is a **bcnd ne0**, the branch is predicted to be taken, so on clock 3 instruction issue begins along the branch target instruction stream. Four instructions, T0–T3, are issued conditionally; the issue of T4 and T5 is delayed because of a data dependency. On clock 5, the data from the **ld** at instruction 0 becomes available to the instruction unit, and the branch condition is evaluated during that same clock. During clock 6, it is known that the branch was mispredicted, so the first two instructions from the correct instruction stream, N0 and N1, are fetched from the instruction cache. At the same time, the issue of instructions T4 and T5 is canceled, and the machine state is restored to the state before the issue of instructions T2 and T3. Instruction issue is delayed one more clock while the effects of instructions T0 and T1 are reversed during clock 7. In clock 8, instruction issue is resumed down the correct path.

LEGEND:

- ☐ INSTRUCTION FETCH
- ▨ EXECUTE
- ↓ FEED FORWARD
- ■ WRITE-BACK
- ▨ DELAYED
- ▨ DELAYED IN LOAD BUFFER
- ▨ ABORTED (BUBBLE)
- ▥ TARGET INSTRUCTION CACHE ACCESS (HIT)
- ▨ DATA CACHE ACCESS
- ▧ INSTRUCTION CACHE ACCESS
- ▨ UNDO
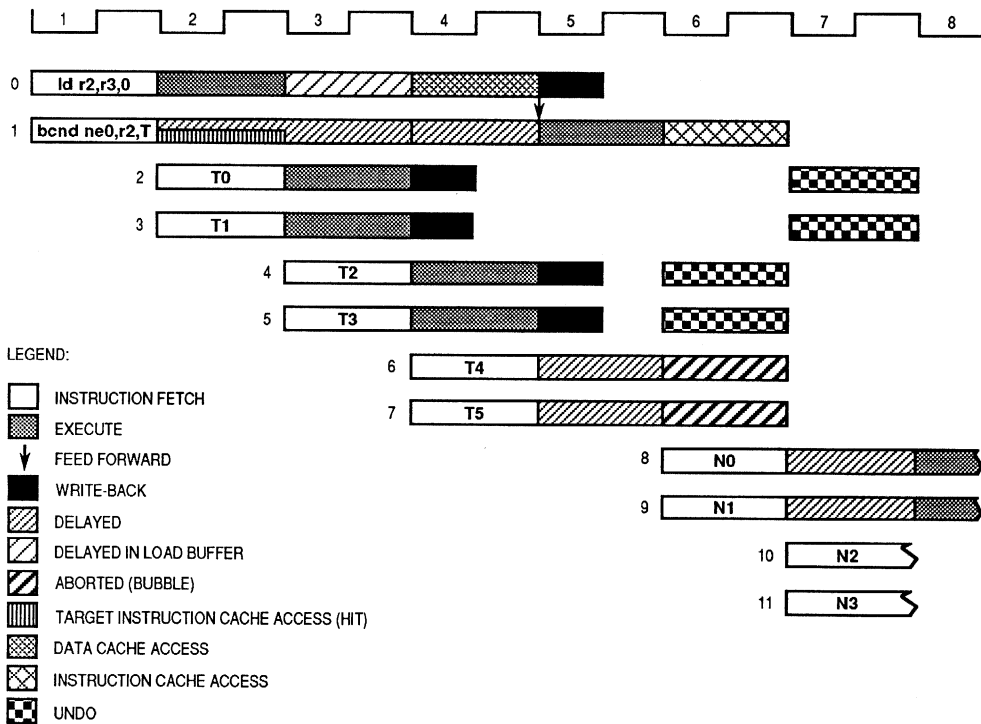
Figure 9-44. Long Latency with Misprediction Timing

9.3.5 Graphics Unit Execution Timing

The process of rendering realistic animated 3D images in real time is computationally intensive; therefore, the MC88110 has a dedicated set of instructions for accelerating 3D graphics rendering algorithms. All graphics instructions in the MC88110, except pixel multiplication, execute in a single clock cycle. Like all other MC88110 instructions, graphics instructions are capable of issuing two at a time. They can be intermixed freely with other integer and floating-point instructions with no restrictions on whether they are in the first or second slot in an issue pair. Also, as with the other execution units, instruction pipelines in the graphics unit are not exposed to the programmer. This means that NOPs are not required to schedule pipeline delay slots. Data dependencies are automatically detected and interlocked by the same hardware scoreboard mechanism used for all other instructions.

There are two independent graphics units in the MC88110: a pixel adder and a pixel packing/unpacking unit. Instructions executed by the pixel adder include **padd**, **padds**, **psub**, **psubs**, and **pcmp**. The pixel packing/unpacking unit is a specialized bit-field unit for packing, unpacking, and shifting pixel or fixed-point data. Instructions executed by the pixel packing/unpacking unit include **ppack**, **punpk** and **prot**. Both of the graphics units execute instructions in a single clock.

Table 9-8 shows the execution timings for the MC88110 graphics instructions. Note that the **pmul** instruction is executed by the multiply unit, which also executes floating-point and integer multiplication operations.

Table 9-8. Graphics Instruction Execution Timings in Clock Cycles

Instruction	Execution Unit	Latency
padd, psub	Pixel Add	1
padds, psubs	Pixel Add	1
pcmp	Pixel Add	1
ppack	Pixel Pack	1
punpk	Pixel Pack	1
prot	Pixel Pack	1
pmul	Multiply	3

In the example shown in Figure 9-45, a **padd** and a **punpk** are issued during clock 2— each to their respective graphics execution units. In clock 3, an attempt is made to issue a **padd** and a **psub**; however, both of these instructions use the graphics adder. As a result, the **psub** is delayed one clock, and is issued in clock 4. Also in clock 4, a **pmul** instruction is kept from issuing because of a data dependency (**r4**) on instruction 3. On clock 5 the **pmul** and an **add** are issued. On clock 8, the **ppack** receives the required data from the multiplier and is issued. The **punpk** fetched in clock 5 cannot be issued to the packing unit during the same clock as the **ppack** instruction and so its issue is delayed until clock 9.
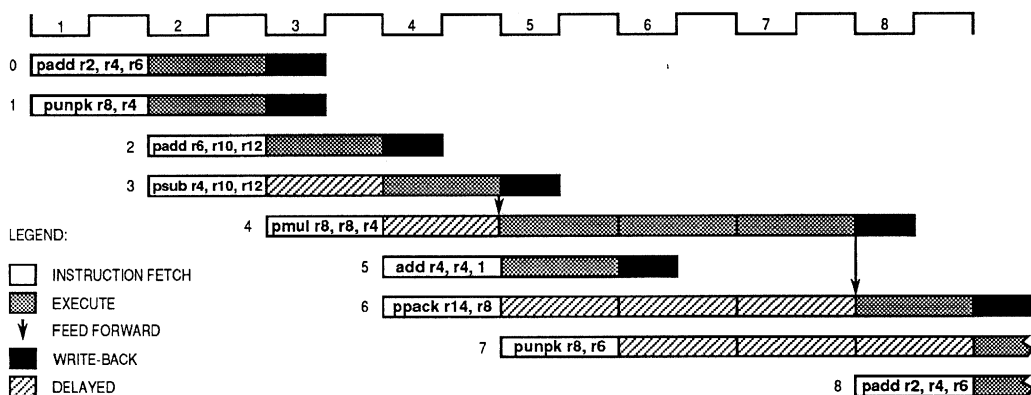
Figure 9-45. Example Graphics Pipelines

9.3.6 Instruction Execution Example

This example demonstrates the instruction sequencing capability of the MC88110 for a highly computational, floating-point intensive process—a 3D transformation involving matrix multiplication. For this example, matrix **R**, a vertex, is multiplied by the transform, matrix **K**, and the resulting matrix is **R'** (where **R' = R * K**) as shown in the following illustration:

$$
\begin{array}{ccc}
\mathbf{R'} & \mathbf{R} & \mathbf{K} \\
\end{array}
$$

$$
[X',Y',Z',H'] \;=\; [X,Y,Z,H] \;*\;
\begin{vmatrix}
A_0 & B_0 & C_0 & D_0 \\
A_1 & B_1 & C_1 & D_1 \\
A_2 & B_2 & C_2 & D_2 \\
A_3 & B_3 & C_3 & D_3
\end{vmatrix}
$$

To perform this matrix multiplication, the code sequence shown in Figure 9-46 performs 32 floating-point multiplications, 24 floating-point additions, 8 loads, and 8 stores. The 80 instructions in the code sequence are executed in 41 clock cycles, resulting in an average rate of 1.95 instructions per clock cycle. In addition, at 50 MHz, this sequence results in 2.4 mega-points per second, 68 double-precision MFLOPS, and 97 MIPS.

9

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
|-|-|-|  fmul.ddd T2 20 A02          |-|-|-|  fadd.ddd T2 T2 T1
|-|-|    ld.d Y1 IPTR YNDX           |-|-|-|  fmul.ddd T1 X1 A10
 |-|-|-|  fmul.ddd T3 H0 A03          |-|-|-|  fadd.ddd T0 T0 T6
|-|-|    ld.d 21 IPTR ZNDX           |-|-|-|  fmul.ddd T5 Y1 A11
  |-|-|-|  fadd.ddd T0 T6 T1          |-|-|-|  fadd.ddd T3 T3 T4
  |-|-|-|  fmul.ddd T1 X0 A10         |-|-|-|  fmul.ddd T4 21 A12
   |-|-|-|  ld.d H1 IPTR HNDX         |-|-|    st.d T0 TEMP HNDX
   |-|-|-|  fmul.ddd T4 Y0 A11        |-|-|-|  fmul.ddd T6 H1 A13
    |-|-|-|  fadd.ddd T2 T2 T3         |-|-|-|  fadd.ddd T1 T1 T5
    |-|-|-|  fmul.ddd T3 20 A12        |-|-|-|  fmul.ddd T5 X1 A20
     |-|-|-|  fmul.ddd T5 H0 A13        |-|-|-|  fadd.ddd T2 T2 T3
      |-|-|-|  fadd.ddd T1 T1 T4         |-|-|-|  fmul.ddd T0 Y1 A21
      |-|-|-|  fmul.ddd T4 X0 A20        |-|-|-|  fadd.ddd T4 T4 T6
       |-|-|-|  fadd.ddd T0 T0 T2         |-|-|-|  fmul.ddd T6 21 A22
       |-|-|-|  fmul.ddd T2 Y0 A21        |-|-|    st.d T2 TPTR XNDX
        |-|-|-|  fadd.ddd T3 T3 T5          |-|-|-|  fmul.ddd T3 H1 A23
        |-|-|-|  fmul.ddd T5 20 A22         |-|-|-|  fadd.ddd T5 T5 T0
         |-|-|    st.d T0 TPTR XNDX          |-|-|-|  fmul.ddd T0 X1 A30
         |-|-|-|  fmul.ddd T6 H0 A23          |-|-|-|  fadd.ddd T1 T1 T4
          |-|-|-|  fadd.ddd T4 T4 T2           |-|-|-|  fmul.ddd T2 Y1 A31
          |-|-|-|  fmul.ddd T2 X0 A30          |-|-|-|  fadd.ddd T6 T6 T3
           |-|-|-|  fadd.ddd T1 T1 T3           |-|-|-|-|  fmul.ddd T3 21 A32
           |-|-|-|  fmul.ddd T0 Y0 A31          |-|-|    st.d T1 TPTR YNDX
            |-|-|-|  fadd.ddd T5 T5 T6            |-|-|-|  fmul.ddd T4 H1 A33
            |-|-|-|-|  fmul.ddd T6 20 A32          |-|-|-|  fadd.ddd T0 T0 T2
             |-|-|    st.d T1 TPTR YNDX            |-|    add IPTR IPTR 16
             |-|-|-|  fmul.ddd T3 H0 A33            |-|-|-|  fadd.ddd T5 T5 T6
              |-|-|-|  fadd.ddd T0 T0 T2            |-|-|-|  ld.d X1 IPTR XNDX
              |-|    add IPTR IPTR 16                |-|-|-|  fadd.ddd T3 T3 T4
               |-|-|-|  fadd.ddd T4 T4 T5            |-|    add TEMP TPTR 0
               |-|-|-|-|  ld.d X0 IPTR XNDX           |-|-|-|  fmul.ddd T6 X0 A00
                |-|-|-|-|  fadd.ddd T6 T6 T3           |-|-|    st.d T5 TEMP ZNDX
                |-|    add TEMP TPTR 0                  |-|-|-|  fmul.ddd T1 Y0 A01
                 |-|    add TPTR TPTR 16                |-|-|-|  fadd.ddd T4 T0 T3
                 |-|-|    st.d T4 TEMP ZNDX             |-|-|-|-|  st.d T4 TEMP HNDX
                  |-|-|-|  fmul.ddd T2 X1 A00            |-|    bcnd.n /= N loop
                  |-|    sub N N 2                       |-|    add TPTR TPTR 16
                   |-|-|-|  fmul.ddd T1 Y1 A01
                   |-|-|    ld.d Y0 IPTR YNDX
                    |-|-|-|  fmul.ddd T3 21 A02
                    |-|-|    ld.d 20 IPTR ZNDX
                     |-|-|-|  fmul.ddd T4 H1 A03
                     |-|-|    ld.d H0 IPTR HNDX
```

Figure 9-46. Example Matrix Multiplication Code Sequence

9.4 MEMORY PERFORMANCE CONSIDERATIONS

When instruction throughput approaches two instructions per clock cycle, lack of data bandwidth can become a performance bottleneck. In order for the MC88110 to approach its potential performance levels, it must be able to read and store data quickly. If there are many processors in a system environment, one processor may experience long memory latencies while another bus master (e.g., another processor or a direct memory access controller) is using the external bus.

In order to alleviate this possible contention, the MC88110 provides three memory update modes: write-back, write-through, and cache inhibit. Each page of memory is specified to be in one of these modes. If a page is in write-back mode, data being stored to that page is written only to the data cache. If a page is in write-through mode, writes to that page update the data cache on hits and always update main memory. If a page is cache inhibited, data in that page will never be stored in the data cache. All three of

these modes of operation have advantages and disadvantages. Which mode to use depends on the system environment as well as the application.

This section describes how performance is impacted by each memory mode. For details on the operation of the data cache and the memory update modes, refer to **Section 6 Instruction and Data Caches**.

9.4.1 Write-Back Mode

When storing data while in write-back mode, store operations for cacheable data do not necessarily cause an external bus cycle to update memory. Instead, memory updates only occur on line replacements, cache flushes, or when another processor attempts to access a specific address for which there is a corresponding dirty cache entry. For this reason, write-back mode may be preferred when external bus bandwidth is a potential bottleneck—e.g., in a multiprocessor environment. Write-back mode is also well suited for data that is closely coupled to a processor, such as local variables.

If more than one processor uses data stored in a page which is in write-back mode, snooping must be enabled to allow write-back operations and cache invalidations of modified data. The MC88110 implements snooping hardware to prevent other devices from accessing invalid data. When bus snooping is enabled, the MC88110 monitors the transactions of the other devices. For example, if another device accesses a memory location, the MC88110 data cache has a modified value for that address, and the global (G) bit corresponding to that page is set, the MC88110 preempts the bus transaction, and updates memory with the cache data. The other device is then free to attempt an access to the updated memory address. See **Section 11 System Hardware Design** for complete information on bus snooping.

Write-back mode provides complete cache/memory coherency as well as maximizing available external bus bandwidth.

9.4.2 Write-Through Mode

Store operations to memory in write-through mode always update memory as well as the data cache (on data cache hits). Write-through mode is used when the data in the cache must always agree with external memory (e.g., video memory) or when there is shared (global) data that may be used frequently or when allocation of a cache line on a cache miss is undesirable. Automatic write-back of cached data is not performed if that data is from a memory page marked as write-through mode since valid cache data always agrees with memory.

It is important to note that although store operations do not cause any scoreboard bits to be set (i.e., store operations never cause data dependencies), stores to memory that is in write-through mode may cause a decrease in performance. Each time a store is performed to memory in write-through mode, the bus will be busy for the extra clock cycles required to perform the memory update; therefore, pending load operations which miss the data cache must wait while the external store operation completes.

9

9.4.3 Cache Inhibit

If a memory page is specified to be cache inhibited, data from this page will not be stored in the data cache.

Areas of the memory map can be cache inhibited by the operating system software; however, the **xmem** instruction always performs as if cache inhibition is in effect. If a cache inhibited access hits in the data cache, the corresponding cache line is invalidated. If the line is marked as modified, it is copied back to memory before being invalidated.

The cache inhibited mode is most detrimental to performance since every memory access must bypass the data cache and incur the latencies of a bus transaction with memory.

9.5 SUPERSCALAR OPTIMIZATION TECHNIQUES

The MC88110 instruction set allows software to break large tasks into smaller ones that execute very rapidly and, if possible, in parallel. Performance can be hindered by lower instruction throughput caused by poor instruction scheduling. Good instruction scheduling techniques can greatly increase the performance levels of the MC88110. The MC88110 has many design features, such as multiple independent execution units, two independent ALUs, a store reservation station, and static branch prediction which simplify the task of scheduling code. The MC88110 can use many of the scheduling algorithms that were appropriate for the MC88100; however, the dual instruction issue capability of the MC88110 slightly changes effectiveness of these algorithms.

The following paragraphs address some of the issues involved with code scheduling for the MC88110. In addition, some insights are provided into how the code scheduling algorithms for an MC88110 code scheduler differ from existing MC88100 code scheduling algorithms. Finally, a brief example of code scheduling for the MC88110 is given.

9.5.1 The Impact of Superscalar Processing on Schedulers

The ability to issue more than one instruction per clock cycle adds a new dimension to the code scheduling algorithms used for the MC88100. Not only must the programmer schedule each instruction, but must also look at each potential pair of instructions as a single unit. The programmer should try to maximize the instances in which an instruction pair can issue together. Since there are no address boundaries dictating whether an instruction will be placed in the first or second issue slot, the programmer usually doesn't know if an instruction will be paired with the instruction above or the instruction below.

When scheduling small segments of code, the programmer only needs to avoid execution unit and register contentions within instruction issue pairs (except in the case of the divide and data execution units, which may not be able to accept new instructions every clock cycle). An example of this might be if a code sequence called for 5 **fadd** and 5 **fmul** instructions to be executed. Rather than issue the 5 **fadds** and then the 5 **fmuls**

(taking 9 clock cycles to issue and 12 clock cycles to execute the sequence), the sequence can be combined as **fadd fmul, fadd fmul, fadd fmul, fadd fmul, fadd fmul**. Since the floating-point adder and multiplier are both fully pipelined (they can receive a new instruction each clock cycle), and both execution units are independent, each **fadd-fmul** pair can issue together on each clock cycle. The new sequence will issue in 5 clocks and execute in 8 clocks. To further simplify scheduling, two ALU execution units prevent execution unit contention within ALU instruction pairs.

When scheduling larger segments of code, execution unit contentions, write-back contentions, and execution latencies must all be given additional attention. For example, all single-cycle instructions are guaranteed a slot on the destination bus while multi-cycle instructions must arbitrate. Therefore, it is possible to issue a multi-cycle instruction followed by a stream of single-cycle instructions which use all available write-back slots. This may lead to bubbles in the instruction stream because a multi-cycle instruction will not be able to write its results to the register file before another instruction needs those results.

An example of this case is shown in Figure 9-47. The code sequence begins with a **ld** operation into **r7**. During the execute and cache access phases of the **ld** operation, additional instructions are being issued and executed. To hide any latency which might be incurred, the scheduler has placed 12 instructions between the **ld** operation and the instruction which uses **r7**. The execution of 12 instructions would seem to provide enough time for the **ld** operation to complete; however, all 12 instructions are single-cycle operations with no register or execution unit contentions. This combination guarantees that every write-back slot will be used.

The **ld** operation is ready to place its results on the destination bus on clock 3, however instructions 3 and 4 are using the write-back slot during clock 3. During clock 4 the **ld** operation is again denied a write-back slot because instructions 5 and 6 are completing. This continues until clock 8. During clock 6, instructions 13 and 14 are prefetched. During clock 7, instruction 13 is executed, but instruction 14 is stalled because of a data dependency on **r7**. This bubble propagates through the instruction pipeline and provides an open slot on the destination bus during clock 8. On clock 8, the **ld** operation writes its results to the register file and simultaneously forwards its results to instruction 14. In this example, the write-back opportunity for the **ld** operation is a result of a bubble which is produced because the **ld** operation has not yet written its results.
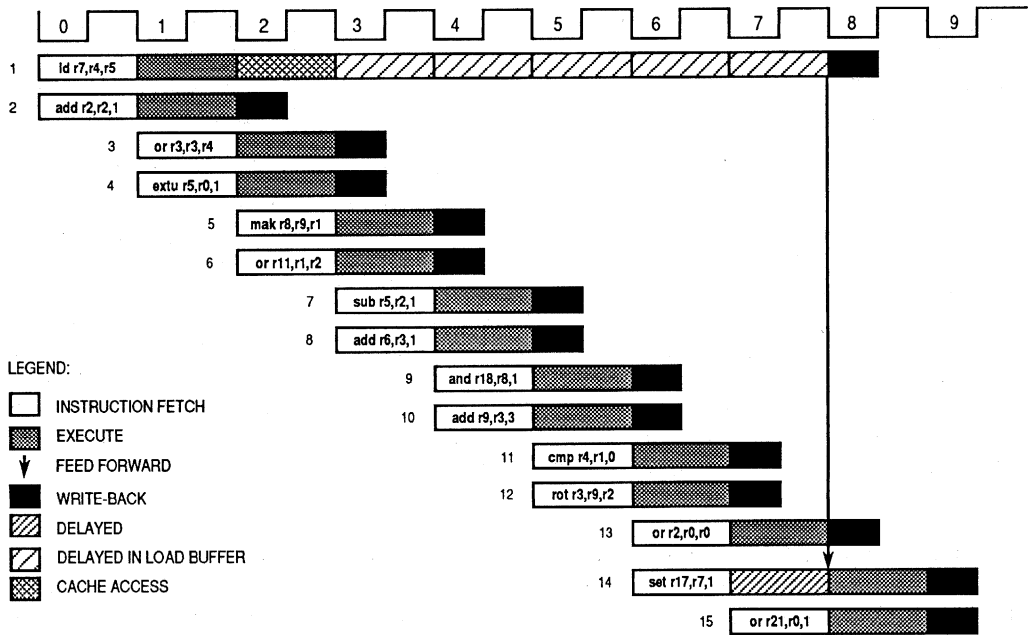
Figure 9-47. Instruction Stall Due to Write-Back Arbitration

9.5.2 Upgrading from an MC88100 Scheduler to an MC88110 Scheduler

The following paragraphs describe some of the guidelines used to schedule instructions for the MC88100 and discuss how these algorithms can be be adapted to produce efficient code for the MC88110.

9.5.2.1 OVERLAPPING LATENCIES WITH USEFUL WORK. One technique of scheduling instructions for the MC88100 is to overlap unavoidable latencies with useful work. Figure 9-48 shows an example of this technique.
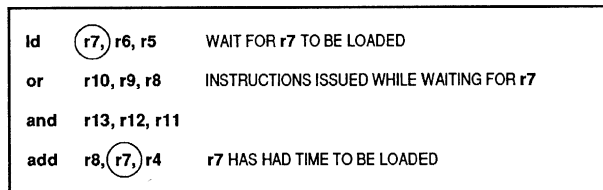
Figure 9-48. Example of the MC88100 Technique of Overlapping Latencies with Useful Work

Notice that when a **ld** instruction is issued in an MC88100, the data is not ready for at least three clock cycles. This number has been reduced to two clocks on the MC88110 for a data cache hit. While the data is being read into the register file, other instructions can be issued.

This scheme is still effective on the MC88110, but the dual instruction issue characteristic must also be taken into account. Suppose the code above is run through an MC88110. There are four instructions in the sequence and it is not possible to determine which instructions will be paired together. The first **ld** instruction may be paired with the instruction above it or it may be paired with the **or** instruction.

Consider the case in Figure 9-48 where the **ld** instruction is executed together with an instruction that might appear above it. In the next clock cycle, the **and** and **or** instructions will execute together. In the third clock cycle, the **add** instruction can execute because the first **ld** instruction has had time to complete (assuming a data cache hit). In this instruction sequence, no stalls occur (assuming that none of the instructions shown depend on the instructions preceding the **ld** operation).

Now consider the case in Figure 9-48 where the **ld** instruction is executed together with the first **or** instruction. In the next clock cycle, it would have been possible to execute both the **and** and **add** instructions together. Unfortunately not enough time has passed for the **ld** instruction to complete, thus a stall occurs. The **add** instruction must wait an additional clock cycle to execute.

9.5.2.2 NO GROUPING VS. GROUPING OF LIKE INSTRUCTIONS. When scheduling assembly code for the MC88100, a common technique is to group like instructions. Benefits of this technique include making the code more readable as well as overlapping unavoidable latencies with other useful work. Figure 9-49 illustrates this technique.

9

```
ld.d        r24, r4, 0      ; dx(i)
ld.d        r22, r4, 8      ; dx(i+1)
ld.d        r20, r4, 16     ; dx(i+2)
ld.d        r18, r4, 24     ; dx(i+3)
fmul.ddd    r24, r24, r8    ; da*dx(i)
fmul.ddd    r22, r22, r8    ; da*dx(i+1)
fmul.ddd    r20, r20, r8    ; da*dx(i+2)
fmul.ddd    r18, r18, r8    ; da*dx(i+3)
ld.d        r16, r6, 0      ; dy(i)
ld.d        r14, r6, 8      ; dy(i+1)
ld.d        r12, r6, 16     ; dy(i+2)
ld.d        r10, r6, 24     ; dy(i+3)
fadd.ddd    r16, r16, r24   ; dy(i)
fadd.ddd    r14, r14, r22   ; dy(i+1)
fadd.ddd    r12, r12, r20   ; dy(i+2)
fadd.ddd    r10, r10, r18   ; dy(i+3)
addu        r4, r4, 32      ; dx
subu        r2, r2, 4       ; loop count
st.d        r16, r6, 0      ; store dy(i)
st.d        r14, r6, 8      ; store dy(i+1)
st.d        r12, r6, 16     ; store dy(i+2)
st.d        r10, r6, 24     ; store dy(i+3)
```

**Figure 9-49. Example of the MC88100
Technique of Grouping Like Instructions**

The code sequence in Figure 9-49 begins by loading four elements from the array dx. By the time data from these load operations is needed, it is ready (assuming cache hits). Next, four **fmul.ddd** instructions are issued. Again, these operations have completed by the time their results are needed (eight instructions later). Furthermore, the corresponding **fadd.ddd** instructions are complete by the time the results from the loop begin to be stored.

Grouping like instructions improves throughput of code on the MC88100. This technique can also be used with the MC88110; however, the ability to issue two instructions per clock combined with the limitations imposed by the execution units must be taken into account when using this technique.

Recall that each execution unit on the MC88110 can only accept one instruction per clock cycle. Since there are two ALUs, two arithmetic/logic instructions can be executed per clock. Suppose the code sequence in Figure 9-49 was executed by an MC88110. There are three segments of this code that contain a series of back-to-back **ld** or **st** instructions. Since the data unit can accept only one instruction per clock, and **ld** or **st** instructions are both executed by the data unit, dual instruction issue would not occur during these segments.

Two floating-point instructions can be issued during the same clock cycle if they are issued to two different execution units. There are two segments of the code in Figure 9-49 that contain a series of back-to-back floating-point instructions which would be issued

to the same execution unit. Thus, dual instruction issue would not occur during these segments.

Because grouping like instructions often wastes the opportunity to issue two instructions per clock cycle, this technique must be used carefully when scheduling instructions for the MC88110. Note however, that there are no execution unit considerations when grouping ALU instructions since two ALU instructions can be issued and executed simultaneously.

9.5.2.3 REGISTER USAGE. Since the register scoreboard cannot be updated instantaneously, the scoreboard mechanism cannot be used to resolve data dependencies between instructions within an issue pair. These dependencies are resolved in the MC88110 by Instruction Timing: which is similar to the register scoreboard mechanism. When there is a register conflict between instructions in an issue pair, the interdependency resolution hardware stalls the second instruction until the register becomes available.

When scheduling code for the MC88100, register allocation is not a concern when ordering single-cycle instructions; since the MC88100 can only issue one instruction per clock cycle, single-cycle instructions do not cause scoreboard holds. However, for the MC88110, register allocation is an important performance factor which must be considered when scheduling single- or multi-cycle instructions. Even though a sequence of instructions may execute on an MC88100 with no scoreboard holds or stalls, it is quite possible that when the same sequence is run on the MC88110, the interdependency resolution hardware will prevent two instructions from being issued during the same clock cycle. Although the performance for the same code on the MC88110 can not be any worse than its relative performance on the MC88100, opportunities for dual instruction issue may be lost.

9

The interdependency resolution hardware uses the following rules to determine whether both instructions in an issue pair will be issued during the same clock cycle (see Figure 9-50):

1. Two instructions will not be issued on the same clock cycle if the destination register for the first instruction is a source register for the second instruction. However, if the first or second instruction is a **st** operation, and the data which is being stored is dependent on the results of the instruction in issue slot one, this rule is not applied by the interdependency resolution hardware. Similarly, if the second instruction is a predicted branch, this rule is not applied. For more information on branch prediction and store instructions, refer to **9.3.4.3 Static Branch Prediction** and **9.2.2 Load Buffer and Store Reservation Station Model**.

2. Two instructions will not be issued during the same clock cycle if the destination register is the same for both instructions. This situation should be very rare since the compiler could simply remove the first instruction without any logical effect on the program results.

3. Two instructions may be issued in the same cycle if the source register for the first instruction is the destination register for the second instruction. The pair can be issued together because the data is forwarded to the appropriate execution units in the order that it appears in the instruction pair. In other words, the data in **r3** (see Figure 9-50) is forwarded into the first ALU before the second instruction can modify it. It is as if the instructions in the issue pair are executed sequentially, but performance is improved because instructions are executed simultaneously.
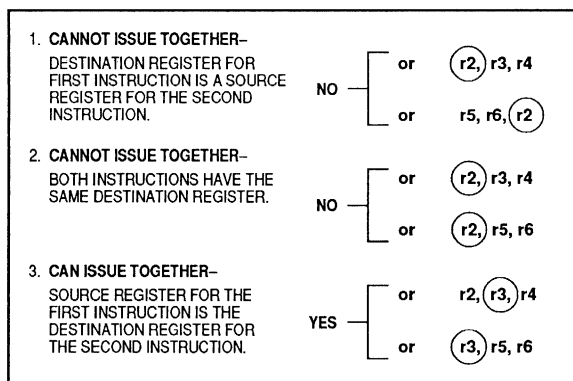
Figure 9-50. Interdependency Resolution Hardware Rules

9.5.3 Code Optimization Example

The following paragraphs provide a brief overview of some code scheduling techniques which are applicable to the MC88110. The code segments shown are examples and do not represent the best possible code scheduling.

Figure 9-51 shows a tight loop of double-precision operations that has been compiled into MC88110 assembly language. No instruction scheduling was performed during the conversion. If this code sequence is run on an MC88110, the first two instructions in the code sequence will not be issued in the same clock cycle because only one memory access instruction can be issued during each clock cycle. The next two instructions can not issue during the same clock cycle for two reasons: both instructions have the same destination register and the **fmul.ddd** instruction requires the data in **r24** (which will not be ready until the completion of the **ld.d** operation). Because of this data dependency, instruction issue will stall until the data in **r24** becomes available.

```
do 50i = 1, n
    dy(i) = dy(i)+da*dx(i)
50 continue
```

```
@L800:
    ld.d        r8, r0,_da      ; r8, r9 ◀ da
    ld.d        r24, r4, 0      ; r24, r25 ◀ dx(i)
    fmul.ddd    r24, r24, r8    ; r24, r25 ◀ da*dx(i)
    ld.d        r16, r6, 0      ; r16, r17 ◀ dy(i)
    fadd.ddd    r16, r16, r24   ; r16, r17 ◀ dy(i)+da*dx(i)
    st.d        r16, r6, 0      ; dy(i) ◀ r16, r17
    addu        r4, r4, 8       ; r4 ◀ &dx(i)
    addu        r6, r6, 8       ; r6 ◀ &dy(i)
    subu        r2, r2, 1       ; r2 ◀ n-4
    bcnd        gt, r2, @L800   ; branch if n > 0
```
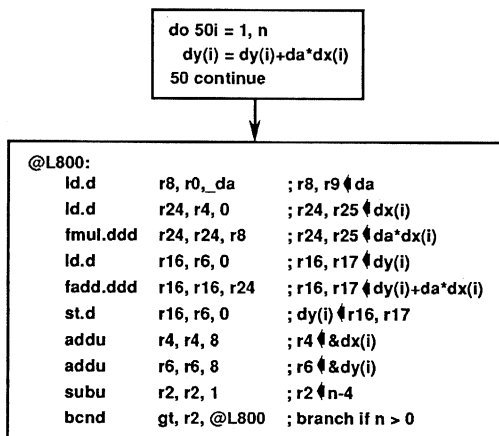
Figure 9-51. Example Source Code Which Has Been Converted into Assembly Language

Once the second **ld.d** operation has completed, the **fmul.ddd** and the third **ld.d** will be issued during the next clock cycle. Unfortunately, another stall will occur in the following clock cycle because the **fadd.ddd** instruction must wait for the data in **r16** to become available.

Once the load into **r16** has completed, the processor can issue the **fadd.ddd** and the **st.d**. These two instructions are issued in the same clock cycle even though the **st.d** has a data dependency (**r16**) on the **fadd.ddd** instruction because the **st.d** instruction waits in the store reservation station for the data in **r16** to become available. While the **st.d** instruction is waiting, instruction issue will continue.

Since there are no data dependencies or register contentions between the two **addu** instructions, they will be issued during the same clock cycle. The next two instructions are the **subu** and the **bcnd**. Although the **bcnd** depends on the result of the **subu** instruction, the use of branch prediction allows these two instructions to be issued together. Since the **bcnd** instruction is testing for a greater than (gt) condition, the branch will be predicted to be taken. The **bcnd** instruction will be issued to the branch reservation station. When the **subu** instruction has completed, it will forward its results to the pending **bcnd** instruction. In the meantime, execution will continue at the top of the loop.

Since there are so few instructions in this loop, simply rearranging the instructions provides few options for improving performance. Figure 9-52 illustrates a technique called Instruction Timing:loop unrolling which increases the number of instructions in the loop. With more instructions in the loop, multi-cycle instructions can be overlapped to achieve maximum throughput.

In Figure 9-52, four iterations of the original loop (see Figure 9-51) are executed within each loop and the loop counter is decremented by four after each pass. Little rescheduling has been done in this example; where there was a single **ld.d** instruction in the first example, four **ld.d** instructions now appear. Although the arrangement of these instructions (i.e., similar instructions grouped together) is effective for achieving a throughput of one instruction per clock cycle, this algorithm can be detrimental to achieving a throughput rate of two instructions per clock cycle (see **9.5.2.2 No Grouping vs. Grouping of Like Instructions**); however, it is possible to rearrange the instructions in the loop to help improve the throughput rate.

9

```
do 50i = 1, n, 4
    dy(i) = dy(i)+da*dx(i)
    dy(i+1) = dy(i+1)+da*dx(i+1)
    dy(i+2) = dy(i+2)+da*dx(i+2)
    dy(i+3) = dy(i+3)+da*dx(i+3)
50 continue
```

```
@L800:
    ld.d        r8, r0,_da      ; r8, r9 ← da
    ld.d        r24, r4, 0      ; r24, r25 ← dx(i)
    ld.d        r22, r4, 8      ; r22, r23 ← dx(i+1)
    ld.d        r20, r4, 16     ; r20, r21 ← dx(i+2)
    ld.d        r18, r4, 24     ; r18, r19 ← dx(i+3)
    fmul.ddd    r24, r24, r8    ; r24, r25 ← da*dx(i)
    fmul.ddd    r22, r22, r8    ; r22, r23 ← da*dx(i+1)
    fmul.ddd    r20, r20, r8    ; r20, r21 ← da*dx(i+2)
    fmul.ddd    r18, r18, r8    ; r18, r19 ← da*dx(i+3)
    ld.d        r16, r6, 0      ; r16, r17 ← dy(i)
    ld.d        r14, r6, 8      ; r14, r15 ← dy(i+1)
    ld.d        r12, r6, 16     ; r12, r13 ← dy(i+2)
    ld.d        r10, r6, 24     ; r10, r11 ← dy(i+3)
    fadd.ddd    r16, r16, r24   ; r16, r17 ← dy(i)+da*dx(i)
    fadd.ddd    r14, r14, r22   ; r14, r15 ← dy(i+1)+da*dx(i+1)
    fadd.ddd    r12, r12, r20   ; r12, r13 ← dy(i+2)+da*dx(i+2)
    fadd.ddd    r10, r10, r18   ; r10, r11 ← dy(i+3)+da*dx(i+3)
    subu        r2, r2, 4       ; r2 ← n-4
    st.d        r16, r6, 0      ; dy(i) ← r16, r17
    st.d        r14, r6, 8      ; dy(i+1) ← r14, r15
    st.d        r12, r6, 16     ; dy(i+2) ← r12, r13
    st.d        r10, r6, 24     ; dy(i+3) ← r10, r11
    addu        r4, r4, 32      ; r4 ← &dx(i+4)
    bcnd.n      gt, r2, @L800   ; branch if n≥4, &
    addu        r6, r6, 32      ; r6 ← &dy(i+4)
```

Figure 9-52. First Pass Loop Unrolling

Figure 9-53 shows a rescheduled version of the instructions in Figure 9-52. The shaded lines represent clock cycles. Since it is usually not possible to determine if a certain instruction will correspond to the first or second issue slot of a clock cycle, the clock divisions shown may not be valid during the first pass through the loop. However, the **bcnd.n** instruction puts the loop into a steady state after the first pass, thus making the indicated clock divisions valid. The clock boundaries shown assume cache hits on every memory access.

```
@L800:
        fmul.ddd    r24, r24, r8     ; r24, r25 ◀ da*dx(i)
        ld.d        r16, r6, r0      ; r16, r17 ◀ dy(i)
        fmul.ddd    r22, r22, r8     ; r22, r23 ◀ da*dx(i+1)
        ld.d        r14, r6, 8       ; r14, r15 ◀ dy(i+1)
        fmul.ddd    r20, r20, r8     ; r20, r21 ◀ da*dx(i+2)
        ld.d        r12, r6, 16      ; r12, r13 ◀ dy(i+2)
        fmul.ddd    r18, r18, r8     ; r18, r19 ◀ da*dx(i+3)
        ld.d        r10, r6, 24      ; r10, r11 ◀ dy(i+3)
        fadd.ddd    r16, r16, r24    ; r16, r17 ◀ dy(i)+da*dx(i)
        ld.d        r24, r4, 0       ; r24, r25 ◀ dx(i)
        fadd.ddd    r14, r14, r22    ; r14, r15 ◀ dy(i+1)+da*dx(i+1)
        ld.d        r22, r4, 8       ; r22, r23 ◀ dx(i+1)
        fadd.ddd    r12, r12, r20    ; r12, r13 ◀ dy(i+2)+da*dx(i+2)
        ld.d        r20, r4, 16      ; r20, r21 ◀ dx(i+2)
        fadd.ddd    r10, r10, r18    ; r10, r11 ◀ dy(i+3)+da*dx(i+3)
        ld.d        r18, r4, 24      ; r18, r19 ◀ dx(i+3)
        st.d        r16, r6, 0       ; dy(i) ◀ r16, r17
        subu        r2, r2, 4        ; r2 ◀ n-4
        st.d        r14, r6, 8       ; dy(i+1) ◀ r14, r15
        st.d        r12, r6, 16      ; dy(i+2) ◀ r12, r13
        st.d        r10, r6, 24      ; dy(i+3) ◀ r10, r11
        addu        r4, r4, 32       ; r4 ◀ &dx(i+4)
        addu        r6, r6, 32       ; r6 ◀ &dy(i+4)
        bcnd        gt, r2, @L800    ; branch if n>4
```

LEGEND:
-------- CLOCK BOUNDARIES

Figure 9-53. Unrolled Loop with Scheduling

For this code sequence, the first four elements of **dx** must be loaded into registers **r24**, **r22**, **r20**, and **r18** before the loop is entered. In addition, it will be necessary to initialize registers **r4** and **r6** to point to the appropriate data structures, and initialize **r2** and **r8** to contain appropriate values.

Provided cache hits occur on every memory access and registers have been initialized before the loop is entered, the loop in Figure 9-53 will execute in 13 clock cycles (at 50 MHz, that is 92.3 MIPS and 30.7 MFLOPS).

SECTION 10
INSTRUCTION SET

This section provides the details for each of the MC88110 instructions. A complete opcode summary is also listed.

10.1 INSTRUCTION SET DETAILS

This section provides a detailed description of each instruction in the MC88110 instruction set. The instructions are arranged in alphabetical order with the instruction mnemonic in large bold type for easy reference.

Each instruction description provides a complete discussion of the instruction operation, the assembler syntax, and the instruction encoding. The assembler syntax is supported by the Motorola MC88110 assembler. Figure 10-1 illustrates how the information is presented for each instruction.

INSTRUCTION NAME

OPERATION DESCRIPTION

ASSEMBLER SYNTAX FOR THE INSTRUCTION

POTENTIAL EXCEPTIONS CAUSED BY THE INSTRUCTION

TEXT DESCRIPTION OF INSTRUCTION OPERATION

INSTRUCTION ENCODING: THE INSTRUCTION CATEGORY, THE ADDRESSING MODULES, THE BIT PATTERNS AND THE FIELDS OF THE INSTRUCTION.

EXPLANATION OF FIELDS WITHIN THE INSTRUCTION

add   Integer Add

Operation: Destination ← Source 1 + Source 2

Assembler   **add**   rD, rS1, rS2   signed add (with
Syntax: **add. ci**   rD, rS1, rS2   signed add plus
add. co   rD, rS1, rS2   signed add, pro
add. cio   rD, rS1, rS2   signed add
add   rD, rS1, SIMM16   signed add with i

Exceptions: Integer Overflow

Description: The add instruction adds the contents of a the S1 reg with either the contents of the S2 register or a 16-bit immediate ope

Instruction Encoding:

Integer Category—Register with 16-Bit Immediate

31	26 25	21 20	16 15
0 1 1 1 0 0	D	S1	SIMM

Integer Category - Triadic Register

31	26 25	21 20	16 15
1 1 1 1 0 1	D	S1	0 1 1 1 0 0 i o

D: Destination Register
S1:   Source 1 Register
SIMM16:   16-Bit Signed Immediate Operand
I: 0 - Disable Carry In
    1 - Add Carry to Results
O: 0 - Disable Carry Out
    1 - Generate carry
S2:   Source 2 Register

Figure 10-1. Instruction Description Format

add

add

Operation: Destination ← Source 1 + Source 2

Assembler Syntax:			
add	rD,rS1,rS2	signed add (without carry)
add.ci	rD,rS1,rS2	signed add plus carry
add.co	rD,rS1,rS2	signed add, propagate carry out
add.cio	rD,rS1,rS2	signed add plus carry, propagate carry out
add	rD,rS1,SIMM16	signed add with immediate (without carry)

Exceptions: Integer Overflow

Description: The **add** instruction adds the contents of the S1 register with either the contents of the S2 register or a 16-bit immediate operand. The immediate operand is zero-extended in unsigned mode or sign-extended in signed immediate mode. Binary addition is performed, and the result is placed in the D register. If the result cannot be represented as a signed 32-bit integer, an integer overflow exception occurs.

The **.ci** option causes the carry bit to be added to the result (i.e., D = S1 + S2 + carry). The **.co** option causes the generated carry bit to be written to the PSR. The **.cio** option causes the carry bit to be added to the result and also causes the generated carry bit to be written to the PSR.

Instruction Encoding:
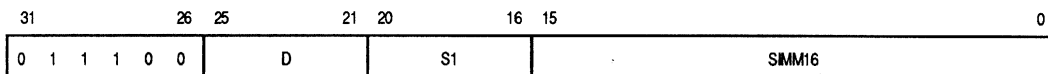
Integer Category—Register with 16-Bit Immediate

31		26	25		21	20		16	15			0
0 1 1 1 0 0			D			S1				SIMM16		

Integer Category—Triadic Register

31		26	25		21	20		16	15		10	9	8	7		5	4		0
1 1 1 1 0 1			D			S1			0 1 1 1 0 0			I	O	0 0 0			S2		
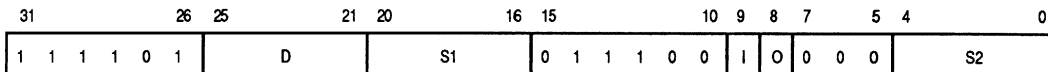
D:	Destination Register (rD)
S1:	Source 1 Register (rS1)
SIMM16:	16-Bit Signed Immediate Operand
I:	0—Disable Carry In
	1—Add Carry to Result
O:	0—Disable Carry Out
	1—Generate Carry
S2:	Source 2 Register (rS2)

addu Unsigned Integer Add addu

Operation: Destination ← Source 1 + Source 2

Assembler   **addu**   **rD,rS1,rS2**   unsigned add (without carry)
Syntax: **addu.ci**   **rD,rS1,rS2**   unsigned add plus carry
                addu.co   **rD,rS1,rS2**   unsigned add, propagate carry out
                addu.cio **rD,rS1,rS2**   unsigned add plus carry,
                                               propagate carry out

                addu   **rD,rS1,IMM16** unsigned add with immediate
                                               (without carry)

Exceptions: None

Description: The **addu** instruction adds the contents of the S1 register to either the contents of the S2 register or to a 16-bit, zero-extended immediate operand. Binary addition is performed, and the result is placed in the D register. The **.ci** option causes the carry bit to be added to the result (i.e., D = S1 + S2 + carry). The **.co** option causes the generated carry bit to be written to the PSR. The **.cio** option causes the carry bit to be added to the result and also causes the generated carry bit to be written to the PSR.

The **addu** instruction does not cause an overflow exception when the sum of the operands cannot be represented as an unsigned 32-bit integer (see the **add** instruction).

Instruction Encoding:

Integer Category—Register with 16-Bit Immediate

31		26	25		21	20		16	15		0
0 1 1 0 0 0			D			S1			IMM16		

Integer Category—Triadic Register

31		26	25		21	20		16	15		10	9	8	7		5	4		0
1 1 1 1 0 1			D			S1			0 1 1 0 0 0	I	O	0 0 0			S2		

D:	Destination Register (rD)
S1:	Source 1 Register (rS1)
IMM16:	16-Bit Unsigned Immediate Operand
I:	0—Disable Carry In
	1—Add Carry to Result
O:	0—Disable Carry Out
	1—Generate Carry
S2:	Source 2 Register (rS2)

10

and

Logical AND

and

Operation: Destination ← Source 1 ∧ Source 2

Assembler	**and**	rD,rS1,rS2
Syntax:	**and.c**	rD,rS1,rS2
	and	rD,rS1,IMM16
	and.u	rD,rS1,IMM16

Exceptions: None

Description: For triadic register addressing, the **and** instruction logically ANDs the data contained in the S1 and S2 registers. The result is stored in the D register. If the **.c** (complement) option is specified, the S2 operand is complemented before being ANDed.

For register with immediate addressing, the **and** instruction logically ANDs the lower 16 bits of the S1 register with the 16-bit unsigned immediate operand encoded in the instruction. The result is stored in the D register, and the upper 16 bits of the S1 register are copied unchanged into the D register. If the **.u** (upper word) option is specified, the upper 16 bits of the S1 operand are ANDed with the immediate operand. The result is stored in the D register, and the lower 16 bits of the S1 register are copied unchanged into the D register.

Instruction Encoding:

Logical Category—Register with 16-bit Immediate

31				27	26	25			21	20		16	15		0
0	1	0	0	0	U		D			S1				IMM16	

Logical Category—Triadic Register

31					26	25		21	20		16	15		11	10	9		5	4		0
1	1	1	1	0	1	D			S1			0 1 0 0 0			C	0 0 0 0 0			S2		

U:	0—AND IMM16 to bits 15–0 of S1
	1—AND IMM16 to bits 31–16 of S1
D:	Destination Register (rD)
S1:	Source 1 Register (rS1)
IMM16:	16-bit Unsigned Immediate Operand
C:	0—Second operand not complemented before the operation
	1—Second operand complemented before the operation
S2:	Source 2 Register (rS2)

Operation: If bit clear, transfer program flow to (D16<<2) + (address of **bb0**)

Assembler   **bb0**   B5, rS1, D16
Syntax: **bb0.n**   B5, rS1, D16

Exceptions: None

Description: The **bb0** instruction examines a bit in the S1 register specified by the B5 field. If the bit is clear, the branch is taken. To calculate the branch target address, the 16-bit displacement is sign-extended and shifted left two bits to form a word displacement, and this displacement is added to the address of the **bb0** instruction. The **.n** (delayed branch) option causes the instruction following the **bb0** instruction to be executed before the branch target instruction is executed.

To ensure future compatibility, the instruction following a **bb0.n** instruction should not be a trap, jump, branch or any other instruction that modifies the instruction pointer. Using such an instruction constitutes a programming error which is not detected.

Use of the **bb0** instruction indicates to the processor for static branch prediction purposes that the branch is not likely to be taken.

Instruction Encoding:

Flow Control Category—Register with 16-Bit Displacement

31				27	26	25		21	20		16	15		0
1	1	0	1	0	N		B5			S1			D16	
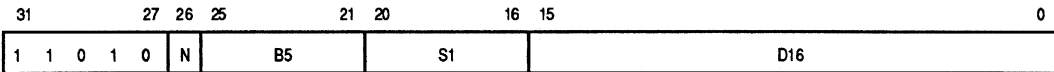
N: 0—Next sequential instruction suppressed
        1—Next sequential instruction executed before branch is taken
B5:   5-bit unsigned integer denoting a bit number in the S1 operand
S1:   Source 1 Register (rS1)
D16:   16-Bit Sign-Extended Displacement

10

Operation: If bit set, transfer program flow to (D16<<2) + (address of **bb1**)

Assembler   **bb1**   B5, rS1, D16
Syntax: **bb1.n**   B5, rS1, D16

Exceptions: None

Description: The **bb1** instruction examines a bit in the S1 register specified by the B5 field of the instruction. If the bit is clear, the branch is taken. To calculate the branch target address, the 16-bit displacement is sign-extended and shifted left two bits to form a word displacement, and this displacement is added to the address of the **bb1** instruction. The **.n** (delayed branch) option causes the instruction following the **bb1** instruction to be executed before the branch target instruction.

To ensure future compatibility, the instruction following a **bb1.n** instruction should not be a trap, jump, branch or any other instruction that modifies the instruction pointer. Using such an instruction constitutes a programming error which is not detected.

Use of the **bb1** instruction indicates to the processor for static branch prediction purposes that the branch is likely to be taken.

Instruction Encoding:

Flow Control Category—Register with 16-Bit Displacement

31				27	26	25		21	20		16	15		0
1	1	0	1	1	N		B5			S1			D16	

N: 0—Next sequential instruction suppressed.
         1—Next sequential instruction executed before branch is taken
B5:   5-bit integer denoting a bit number in the S1 operand
S1:   Source 1 Register (rS1)
D16:   16-Bit Sign-Extended Displacement

10

Operation: If condition true, transfer program flow to (D16<<2) + (address of bcnd)

Assembler	bcnd	eq0,rS1,D16	bcnd.n	eq0,rS1,D16
Syntax:	bcnd	ne0,rS1,D16	bcnd.n	ne0,rS1,D16
	bcnd	gt0,rS1,D16	bcnd.n	gt0,rS1,D16
	bcnd	lt0,rS1,D16	bcnd.n	lt0,rS1,D16
	bcnd	ge0,rS1,D16	bcnd.n	ge0,rS1,D16
	bcnd	le0,rS1,D16	bcnd.n	le0,rS1,D16
	bcnd	M5,rS1,D16	bcnd.n	M5,rS1,D16

Exceptions: None

Description: The **bcnd** instruction provides conditional branching in one instruction without requiring an explicit compare instruction. The **bcnd** instruction examines the data contained in the S1 register and branches if the value in the register meets the specified condition (**eq0** for equals zero, etc.). To form the branch target address, the 16-bit displacement is shifted left two bits and sign-extended to form a word displacement, and then this displacement is added to the address of the **bcnd** instruction. The **.n** (delayed branch) option causes the instruction following the **bcnd.n** instruction to be executed before the branch target instruction.

The MC88110 assembler provides mnemonics for commonly used comparison conditions. The following chart lists these mnemonics and their corresponding bit values for the M5 field. The M5 field may also be indicated explicitly by a literal value.

	Bit:	25	24	23	22	21	
eq0 (equals zero)		0	0	0	1	0	Not Taken
ne0 (not equal to zero)		0	1	1	0	1	Taken
gt0 (greater than zero)		0	0	0	0	1	Taken
lt0 (less than zero)		0	1	1	0	0	Not Taken
ge0 (greater than/equals zero)		0	0	0	1	1	Taken
le0 (less than/equals zero)		0	1	1	1	0	Not Taken

Static branch prediction conventions have been added such that specifying the not equal to zero, greater than zero, and greater than/equals zero conditions indicates that the branch is likely to be taken. Specifying the equals zero, less than zero, and less than/equals zero conditions indicates that the branch is not likely to be taken.

To ensure future compatibility, the instruction following a **bcnd.n** instruction should not be a trap, jump, branch or any other instruction that modifies the instruction pointer. Using such an instruction constitutes a programming error which is not detected.

10

Instruction Encoding:

Flow Control Category—Register with 16-Bit Displacement

31	27	26	25	21	20	16	15	0

1 1 1 0 1	N	M5	S1	D16
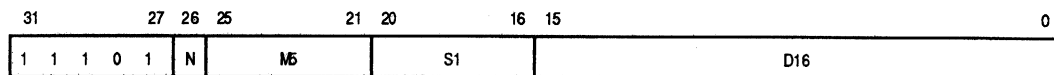
N:	0—Next sequential instruction suppressed.
	1—Next sequential instruction executed before branch is taken
M5:	5-Bit Condition Match Field:

       bit 25—reserved, unused by the branch selection logic
        (must be zero for future compatibility)
       bit 24—maximum negative number   [Sign and Zero]
       bit 23—less than zero   [Sign and (not Zero)]
       bit 22—equal to zero   [(not Sign) and Zero]
       bit 21—greater than zero   [(not Sign) and (not Zero)]

S1:	Source 1 Register (rS1)
D16:	16-Bit Sign-Extended Displacement

10

Operation: Transfer program flow to (D26<<2) + (address of **br**)

Assembler   **br**   D26
Syntax: **br.n**   D26

Exceptions: None

Description: The **br** instruction causes an unconditional transfer of program flow to the branch target address. To form the branch target address, the 26-bit displacement is sign-extended and shifted left two bits to form a word displacement, and this displacement is added to the address of the **br** instruction. The **.n** (delayed branch) option causes the instruction following the **br.n** instruction to be executed before the branch target instruction.

To ensure future compatibility, the instruction following a **br.n** instruction should not be a trap, jump, branch or any other instruction that modifies the instruction pointer. Using such an instruction constitutes a programming error which is not detected.

Instruction Encoding:

Flow Control Category—26-Bit Displacement

31			27	26	25		0
1	1	0	0	0	N	D26	

    N: 0—Next sequential instruction suppressed.

                   1—Next sequential instruction executed before branch is taken

    D26:   26-Bit Sign-Extended Displacement

10

bsr
bsr

Operation: Transfer program flow to (D26<<2) + (address of **bsr**)
r1 ← address of first instruction (second if **.n**) after **bsr**

Assembler **bsr** D26
Syntax: **bsr.n** D26

Exceptions: None

Description: The **bsr** instruction unconditionally transfers program flow to the branch target address and saves the return address in register **r1**. To form the branch target address, the 16-bit displacement is sign-extended and shifted left two bits to form a word displacement, and this displacement is added to the address of the **bsr** instruction. If the **.n** option is not specified, the return address is the address of the instruction following the **bsr** instruction. The **.n** (delayed branch) option causes the instruction following the **bsr.n** instruction to be executed before the branch target instruction.

When the **.n** option is specified, the return address written to **r1** is the address of the second instruction following the **bsr.n** instruction. If the instruction in the delay slot uses **r1** as an operand, the contents of **r1** will be the new return address. If the instruction in the delay slot modifies **r1**, its result will supersede the **bsr** return address.

To ensure future compatibility, the instruction following a **bsr.n** instruction should not be a trap, jump, branch or any other instruction that modifies the instruction pointer. Using such an instruction constitutes a programming error which is not detected.

Instruction Encoding:

Flow Control Category—26-Bit Displacement

31				27	26	25	0
1	1	0	0	1	N	D26	

N: 0—Next sequential instruction suppressed
1—Next sequential instruction executed before branch is taken
D26: 26-Bit Sign-Extended Displacement

clr

Clear Bit Field

clr

Operation: Destination ← (Source 1 ∧ (Bit Field of 0's))

Assembler   **clr**   rD,rS1, W5<O5>
Syntax: **clr**   rD,rS1,rS2

Exceptions: None

Description: The **clr** instruction reads the data from the S1 register and inserts a field of zeros into the data. The result is placed in the D register. The width of the bit field is specified by the W5 field, and the offset of the bit field from bit zero of the S1 data is specified by the O5 field. A W5 field of all zeros specifies a 32-bit wide bit field. If the specified field extends beyond bit 31 of the S1 data, those bits are ignored.

For triadic register addressing, bits 9–5 and bits 4–0 of the S2 register are used as the W5 and O5 fields, respectively, and the rest of the S2 register is ignored.

The following illustration shows the operation of the **clr** rD, rS1, 5<16> instruction. In this example, W5 contains 5 and O5 contains 16, thereby placing a field of five zeros in bits 16 through 20 of the S1 data.

Instruction Encoding:

Bit Field Category—Register with 10-Bit Immediate

31				26	25			21	20			16	15					11	10		5	4		0
1	1	1	1	0	0		D			S1			1	0	0	0	0	0		W5			O5	

Bit Field Category—Triadic Register

31				26	25			21	20			16	15									5	4		0
1	1	1	1	0	1		D			S1			1	0	0	0	0	0	0	0	0	0		S2	

D:	Destination Register (rD)
S1:	Source 1 Register (rS1)
W5:	5 bit unsigned integer denoting a bit-field width (0 denotes 32 bits)
O5:	5-bit unsigned integer denoting a bit-field offset
S2:	Source 2 Register (rS2)

10

Operation: Destination ← Source 1 :: Source 2

Assembler   **cmp**   rD,rS1,rS2
Syntax: **cmp**   rD,rS1,SIMM16

Exceptions: None

Description: The **cmp** instruction compares the data contained in the S1 register with either the data in the S2 register or with the specified 16-bit immediate operand. The immediate operand is sign-extended if the processor is in signed immediate mode or zero-extended if it is in unsigned mode. The instruction returns the evaluated conditions as a bit string in the D register. The format and interpretation of the returned bit string is as follows:

Returned String:

31															16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	nh	he	nb	be	hs	lo	ls	hi	ge	lt	le	gt	ne	eq	0	0

Bits 31–16 and 1–0 are not guaranteed to be zeros in future implementations.

eq:	true (1) if and only if $S_1 = S_2$ (equal)
ne:	true (1) if and only if $S_1 \neq S_2$ (not equal)
gt:	true (1) if and only if $(rS_1) > (rS_2)$ (signed greater than)
le:	true (1) if and only if $(rS_1) \leq (rS_2)$ (signed less than or equal)
lt:	true (1) if and only if $(rS_1) < (rS_2)$ (signed less than)
ge:	true (1) if and only if $(rS_1) \geq (rS_2)$ (signed greater than or equal)
hi:	true (1) if and only if $(rS_1)\ U > (rS_2)$ (unsigned greater than)
ls:	true (1) if and only if $(rS_1)\ U \leq (rS_2)$ (unsigned less than or equal)
lo:	true (1) if and only if $S_1\ U < S_2$ (unsigned less than)
hs:	true (1) if and only if $S_1\ U \geq S_2$ (unsigned greater than or equal)
be:	true (1) if and only if any byte equal
nb:	true (1) if and only if no byte equal
he:	true (1) if and only if any half-word equal
nh:	true (1) if and only if no half-word equal

Comparison results can be used by branch on bit instructions (**bb0** and **bb1**) to synthesize compare and branch on condition operations. The results can also be used by trap on bit instructions (**tb0** and **tb1**). Note that for out-of-bounds array access checking, it is more efficient to use the trap on bounds check instruction (**tbnd**) than to use a **cmp**/trap on bit instruction combination.

Instruction Encoding:

Integer Category—Register with 16-Bit Immediate

31					26	25			21	20			16	15				0

```
31              26 25          21 20          16 15                                      0
┌───────────────┬─────────────┬─────────────┬──────────────────────────────────────────┐
│ 0  1  1  1  1  1 │      D      │     S1      │                  SIMM16                    │
└───────────────┴─────────────┴─────────────┴──────────────────────────────────────────┘
```

Integer Category—Triadic Register

```
31              26 25          21 20          16 15                              5  4      0
┌───────────────┬─────────────┬─────────────┬──────────────────────────────┬───────────┐
│ 1  1  1  1  0  1 │      D      │     S1      │ 0  1  1  1  1  1  0  0  0  0  0 │    S2     │
└───────────────┴─────────────┴─────────────┴──────────────────────────────┴───────────┘
```

D:	Destination Register (rD)
S1:	Source 1 Register (rS1)
SIMM16:	16-Bit Signed Immediate Operand
S2:	Source 2 Register (rS2)

10

Operation: Destination ← Source 1 / Source 2

Assembler   **divs**   rD,rS1,rS2
Syntax: **divs**   rD,rS1,SIMM16

Exceptions: Integer Divide-By-Zero
               Integer Overflow

Description: The **divs** instruction divides the data contained in the S1 register by the data in the S2 register or by the specified 16-bit immediate operand. The immediate operand is zero-extended in unsigned mode or sign-extended in signed immediate mode. A 32-bit two's complement binary division is performed. The quotient is stored in the D register.

If the divisor is zero, the integer divide-by-zero exception is generated. An integer overflow exception can only be caused by dividing the largest magnitude representable (32-bit) negative integer by a negative one. If an integer overflow exception occurs, the rD is not updated.

NOTE

Unlike the MC88100, this instruction does not cause a floating-point unimplemented exception when SFU1 is disabled.

Instruction Encoding:

Integer Category—Register with 16-Bit Immediate

31		26	25		21	20		16	15		0
0 1 1 1 1 0			D			S1			SIMM16		

Integer Category—Triadic Register

31		26	25		21	20		16	15		5	4		0
1 1 1 1 0 1			D			S1			0 1 1 1 1 0 0 0 0 0 0			S2		

     D: Destination Register (rD)
     S1:   Source 1 Register (rS1)
     SIMM16:   16-Bit Signed Immediate Operand
     S2:   Source 2 Register (rS2)

10
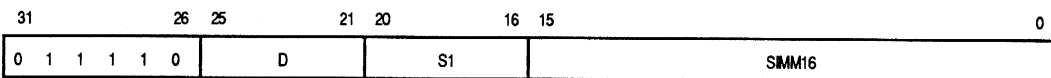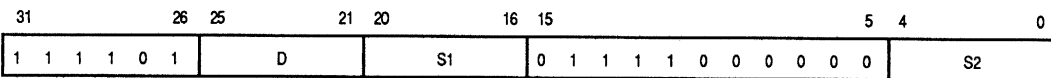
Operation: Destination ← Source 1 / Source 2

Assembler   **divu**   rD,rS1,rS2
Syntax: **divu**   rD,rS1,IMM16
                  divu.d   rD,rS1,rS2

Exceptions: Integer Divide-By-Zero

Description: The **divu** instruction divides the data contained in the S1 register by either the data in the S2 register or by the specified zero-extended, 16-bit immediate operand. A 32-bit two's complement binary division is performed. The quotient is stored in the D register.

If the **.d** option is specified, the unsigned 64-bit value in the double register S1:S1+1 is divided by the unsigned 32-bit value in the S2 register and the 64-bit unsigned quotient is placed in register pair D:D+1.

If the divisor is zero, an integer divide-by-zero exception is generated.

NOTE

Unlike the MC88100, this instruction does not cause a floating-point unimplemented exception when SFU1 is disabled.

Instruction Encoding:

Integer Category—Register with 16-Bit Immediate

31				26	25			21	20			16	15				0
0	1	1	0	1	0		D			S1					IMM16		

Integer Category—Triadic Register

31				26	25		21	20		16	15							9	8	7		5	4		0
1	1	1	1	0	1		D			S1	0	1	1	0	1	0	0	d	0	0	0		S2		
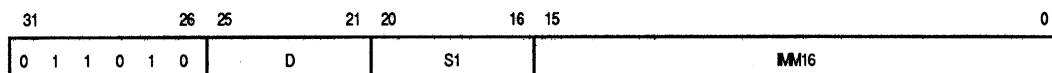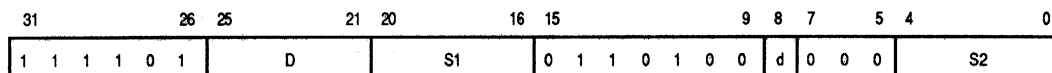
D:	Destination Register (rD)
S1:	Source 1 Register (rS1)
IMM16:	16-Bit Zero-Extended Immediate Operand
d:	0—Single-Word Divide
	1—Double-Word Divide
S2:	Source 2 Register (rS2)

10

Operation: Destination ← (sign-extended bit field) of Source 1

Assembler   **ext**   rD,rS1,W5<O5>
Syntax: **ext**   rD,rS1,rS2

Exceptions: None

Description: The **ext** instruction extracts a bit field from the S1 register. The bit-field width is specified by the W5 field, and the offset of the bit field from bit 0 of the S1 register is specified by the O5 field. The extracted bit field is sign-extended to 32 bits and placed in the D register. If the bit field extends beyond bit 31 of the S1 register, then bit 31 is used as the sign bit and is extended in the D register.

For triadic register addressing, bits 9-5 and 4-0 of the S2 register are used for the W5 and O5 fields, respectively and the rest of the S2 register is ignored.

The following illustration shows the operation of the **ext** instruction:
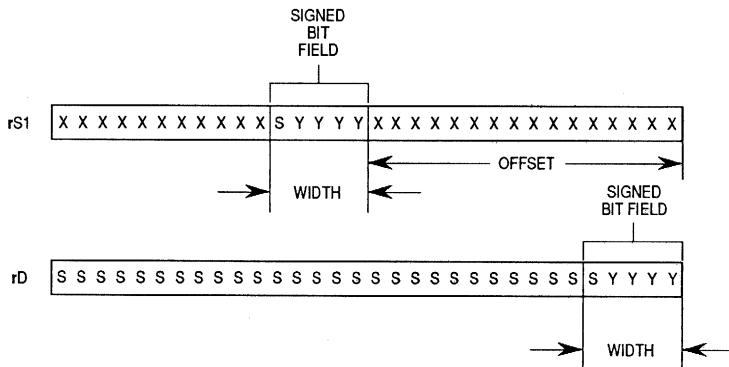
When the W5 field contains all zeros (specifying a bit field width of 32 bits), the **ext** instruction operates as an arithmetic shift right instruction. The O5 field specifies the number of positions to shift, and the high-order bits are sign filled in the D register. The following illustration shows an example of a shift operation performed by the **ext** instruction:

10

WIDTH = 32, OFFSET = 5

rS1

rD

EXTENDED
SIGN BIT

Instruction Encoding:

Bit Field Category—Register with 10-Bit Immediate

31				26	25			21	20			16	15					11	10		5	4		0
1	1	1	1	0 0		D				S1			1	0	0	1	0	0		W5			O5	

Bit Field Category—Triadic Register

31				26	25			21	20			16	15									5	4		0
1	1	1	1	0 1		D				S1			1	0	0	1	0	0	0	0	0	0	0		S2

D: Destination Register (rD)

S1:   Source 1 Register (rS1)

W5:   5-bit unsigned integer denoting a bit-field width (0 denotes 32 bits)

O5:   5-bit unsigned integer denoting a bit-field offset

S2:   Source 2 Register (rS2)

10

Operation: Destination ← (zero-extended bit field) of Source 1

Assembler   **extu**   rD,rS1,W5<O5>
Syntax: **extu**   rD,rS1,rS2

Exceptions: None

Description: The **extu** instruction extracts a bit field from the S1 register. The bit-field width is specified by the W5 field, and the offset of the bit field from bit 0 of the S1 register is specified by the O5 field. The extracted bit field is zero-extended to 32 bits and placed in the D register. If the bit field extends beyond bit 31 of the S1 register, then the portion of the bit field contained in bits 31 and lower is extracted and zero-extended in the D register.

For triadic register addressing, bits 9-5 and 4-0 of the S2 register are used for the W5 and O5 fields, respectively and the rest of the S2 register is ignored.

The following illustration shows the operation of the **extu** instruction:
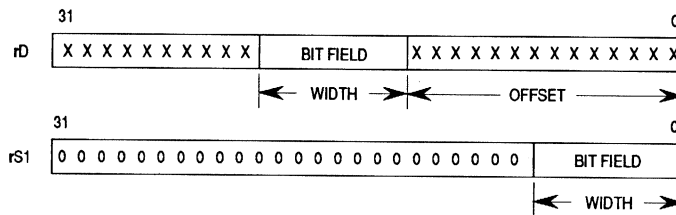
When the W5 field contains all zeros (specifying a bit field width of 32 bits), the **extu** instruction operates as a logical shift right instruction. The O5 field specifies the number of positions to shift, and the high-order bits are zero filled in the D register. The following illustration shows an example of a shift operation performed by the **extu** instruction:
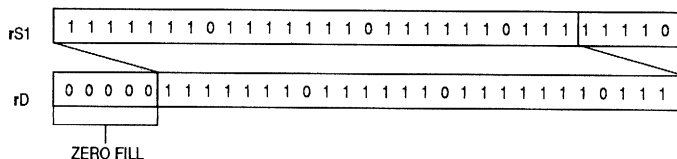
Instruction Encoding:
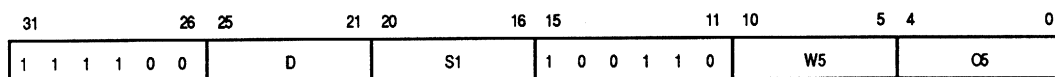
Bit Field Category—Register with 10-Bit Immediate

31				26	25			21	20			16	15				11	10		5	4		0
1 1 1 1 0 0					D				S1				1 0 0 1 1 0					W5			O5		

Bit Field Category—Triadic Register

31				26	25			21	20			16	15					5	4		0
1 1 1 1 0 1					D				S1				1 0 0 1 1 0 0 0 0 0 0						S2		

D:	Destination Register (rD)
S1:	Source 1 Register (rS1)
W5:	5-bit unsigned integer denoting a bit-field width (0 denotes 32 bits)
O5:	5-bit unsigned integer denoting a bit-field offset
S2:	Source 2 Register (rS2)

10
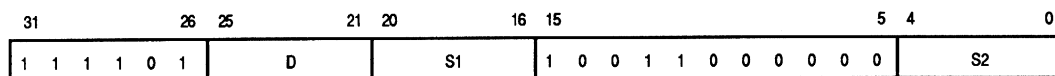
fadd   Floating-Point Add   fadd

Operation: Destination ← Source 1 + Source 2

Assembler   **fadd.sss**   rD,rS1,rS2   **fadd.sss**   xD,xS1,xS2
Syntax: **fadd.ssd**   rD,rS1,rS2   **fadd.ssd**   xD,xS1,xS2
                  fadd.sds   rD,rS1,rS2   **fadd.sds**   xD,xS1,xS2
                  fadd.sdd   rD,rS1,rS2   **fadd.sdd**   xD,xS1,xS2
                  fadd.dss   rD,rS1,rS2   **fadd.dss**   xD,xS1,xS2
                  fadd.dsd   rD,rS1,rS2   **fadd.dsd**   xD,xS1,xS2
                  fadd.dds   rD,rS1,rS2   **fadd.dds**   xD,xS1,xS2
                  fadd.ddd   rD,rS1,rS2   **fadd.ddd**   xD,xS1,xS2
                                                   fadd.ssx   xD,xS1,xS2
                                                   fadd.sdx   xD,xS1,xS2
                                                   fadd.sxs   xD,xS1,xS2
                                                   fadd.sxd   xD,xS1,xS2
                                                   fadd.sxx   xD,xS1,xS2
                                                   fadd.dsx   xD,xS1,xS2
                                                   fadd.ddx   xD,xS1,xS2
                                                   fadd.dxs   xD,xS1,xS2
                                                   fadd.dxd   xD,xS1,xS2
                                                   fadd.dxx   xD,xS1,xS2
                                                   fadd.xss   xD,xS1,xS2
                                                   fadd.xsd   xD,xS1,xS2
                                                   fadd.xsx   xD,xS1,xS2
                                                   fadd.xds   xD,xS1,xS2
                                                   fadd.xdd   xD,xS1,xS2
                                                   fadd.xdx   xD,xS1,xS2
                                                   fadd.xxs   xD,xS1,xS2
                                                   fadd.xxd   xD,xS1,xS2
                                                   fadd.xxx   xD,xS1,xS2

Exceptions: Floating-Point Reserved Operand
                  Floating-Point Overflow
                  Floating-Point Underflow
                  Floating-Point Inexact (if not masked)
                  Floating-Point Unimplemented

10

Description: The **fadd** instruction checks the data in the S1 and S2 registers for reserved operands (NaNs, denormalized or unnormalized numbers). If reserved operands are found, a floating-point reserved operand exception is taken. If no reserved operands are found, the S1 and S2 operands are added according to the IEEE 754 standard, and the result is placed in the D register. Exception conditions occur when an overflow, underflow, or inexact result is detected. If execution of **fadd** is attempted while SFU1 is disabled, a floating-point unimplemented exception is taken.

Any combination of single- and double-precision operands can be specified in the general register file and any combination of single-, double-, or double-extended-precision operands can be specified in the extended register file.

NOTE

The MC88110 performs IEEE 754 infinity arithmetic directly in hardware. Thus, unlike the MC88100, the MC88110 does not treat infinity (∞) as a reserved operand and infinity does not cause an exception.

When the processor is in TCFP mode (i.e., one of the TCFP bits in the FPCR is set), reserved operands do not cause SFU1 exceptions; instead, when a reserved operand is detected, the hardware delivers a default result approximating the IEEE defined result. See **Section 4 Floating-Point Implementation** for more details on TCFP mode.

Instruction Encoding:

Floating-Point Category—Triadic Register

31				26	25		21	20		16	15	14		11	10	9	8	7	6	5	4		0
1	0	0	0	0	1	D		S1			R	0	1	0	1	T1		T2		TD		S2	

D:	Destination Register (rD or xD)
S1:	Source 1 Register (rS1 or xS1)
R:	0—Source Operands in GRF
	1—Source Operands in XRF
T1:	Source 1 Operand Size
T2:	Source 2 Operand Size
TD:	Destination Operand Size
	Note: For the T1, T2, and TD Fields:
	00—Single-Precision
	01—Double-Precision
	10—Double-Extended-Precision
	11—Unused
S2:	Source 2 Register (rS2 or xS2)

Operation: Destination ← Source 1 :: Source 2

Assembler	**fcmp.sss**	rD,rS1,rS2	**fcmp.sss**	rD,xS1,xS2
Syntax:	**fcmp.ssd**	rD,rS1,rS2	**fcmp.ssd**	rD,xS1,xS2
	fcmp.sds	rD,rS1,rS2	**fcmp.sds**	rD,xS1,xS2
	fcmp.sdd	rD,rS1,rS2	**fcmp.sdd**	rD,xS1,xS2
			fcmp.ssx	rD,xS1,xS2
			fcmp.sdx	rD,xS1,xS2
			fcmp.sxs	rD,xS1,xS2
			fcmp.sxd	rD,xS1,xS2
			fcmp.sxx	rD,xS1,xS2

Exceptions: Floating-Point Reserved Operand
                   Floating-Point Unimplemented

Description: The **fcmp** instruction checks the contents of the S1 and S2 registers for reserved operands (NaNs, denormalized or unnormalized numbers). If a reserved operand is found, a floating-point reserved operand exception is taken.

NOTE

If the reserved operand is a NaN, the reserved operand exception handler sets the FINV bit in the FPSR if either a nonsignaling or signaling NaN is found. For the **fcmpu** instruction, the handler only sets the FINV bit when a signaling NaN is found. This is the only difference between the **fcmp** and **fcmpu** instructions.

If no reserved operands are found, the **fcmp** instruction subtracts the S2 operand from the S1 operand, and based on the result of this subtraction, evaluates a number of conditions according to the IEEE 754 standard. The evaluation results are returned as a bit string in the D register and the subtraction result is discarded (no arithmetic overflow or underflow exceptions are ever generated). A comparison to zero and to the bound value in register S2 is also performed, returning bits in the bit string that correspond to the following conditions: ou (out of range or unordered), ib (in range or on boundary), in (in range), and ob (out of range or on boundary or unordered). If the S2 operand is negative, ou, ib, in, and ob are set to zero. If execution of **fcmp** is attempted while SFU1 is disabled, a floating-point unimplemented exception is taken.

The returned comparison results can be used by branch on bit instructions (**bb0, bb1**) to synthesize conditional branch on comparison operations (branch equal, branch greater, etc).

10

NOTE

The MC88110 performs IEEE 754 infinity arithmetic directly in hardware. Thus, unlike the MC88100, the MC88110 does not treat infinity (∞) as a reserved operand and infinity does not cause an exception.

When the processor is in TCFP mode (i.e., one of the TCFP bits in the FPCR is set), reserved operands do not cause SFU1 exceptions; instead, when a reserved operand is detected, the hardware delivers a default result approximating the IEEE defined result.

See **Section 4 Floating-Point Implementation** for more information on the floating-point implementation.

Result String:

31														18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	uge	ul	ule	ug	lg	ue	ob	in	ib	ou	ge	lt	le	gt	ne	eq	leg	un

Bits 31–18 are not guaranteed to be zeros in future implementations.

uge:	unordered or greater than or equal
ul:	unordered or less than
ule:	unordered or less than or equal
ug:	unordered or greater than
lg:	less than or greater than
ue:	unordered or equal

ob — out of range or on boundary

in — in range

ib — in range or on boundary

ou — out of range

ge:	true (1) if and only if (rS1) ≥ (rS2) (signed greater than or equal)
lt:	true (1) if and only if (rS1) < (rS2) (signed less than)
le:	true (1) if and only if (rS1) ≤ (rS2) (signed less than or equal)
gt:	true (1) if and only if (rS1) > (rS2) (signed greater than)
ne:	true (1) if and only if (rS1) ≠ (rS2) (not equal)
eq:	true (1) if and only if (rS1) = (rS2) (equal)
leg:	true (1) if and only if the two operands are less than, greater than, or equal
un:	true (1) if and only if the two operands are unordered (i.e., one or both operands is a NaN).

10

Instruction Encoding:

Floating-Point Category—Triadic Register

31					26	25			21	20				16	15	14			11	10	9	8	7	6	5	4			0
1	0	0	0	0	1		D				S1				R	0	1	1	1	T1		T2		0	0		S2		

D:	Destination Register (rD)
S1:	Source 1 Register (rS1 or xS1)
R:	Register File:
	0—Source Operands in GRF
	1—Source Operands in XRF
T1:	Source 1 Operand Size
T2:	Source 2 Operand Size
	Note: For the T1 and T2 Fields:
	00—Single-Precision
	01—Double-Precision
	10—Double-Extended-Precision
	11—Unused
S2:	Source 2 Register (rS2 or xS2)

fcmpu   Unordered Floating-Point Compare   fcmpu

Operation: Destination ← Source 1 :: Source 2

Assembler   **fcmpu.sss** rD,rS1,rS2   **fcmpu.sss** rD,xS1,xS2
Syntax: **fcmpu.ssd** rD,rS1,rS2   **fcmpu.ssd** rD,xS1,xS2
                  fcmpu.sds rD,rS1,rS2   **fcmpu.sds** rD,xS1,xS2
                  fcmpu.sdd rD,rS1,rS2   **fcmpu.sdd** rD,xS1,xS2
                                                    fcmpu.ssx rD,xS1,xS2
                                                    fcmpu.sdx rD,xS1,xS2
                                                    fcmpu.sxs rD,xS1,xS2
                                                    fcmpu.sxd rD,xS1,xS2
                                                    fcmpu.sxx rD,xS1,xS2

Exceptions: Floating-Point Reserved Operand
                  Floating-Point Unimplemented

Description: The **fcmpu** instruction checks the contents of the S1 and S2 registers for reserved operands (NaNs, denormalized or unnormalized numbers). If a reserved operand is found, a floating-point reserved operand exception is taken.

NOTE

If the reserved operand is a NaN, the reserved operand exception handler only sets the FINV bit when a signaling NaN is found. For the **fcmp** instruction, the handler sets the FINV bit in the FPSR if either a nonsignaling or signaling NaN is found. This is the only difference between the **fcmp** and **fcmpu** instructions.
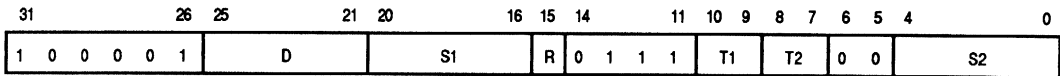
If no reserved operands are found, the **fcmpu** instruction subtracts the S2 operand from the S1 operand, and based on the result of this subtraction, evaluates a number of conditions according to the IEEE 754 standard. The evaluation results are returned as a bit string in the D register and the subtraction result is discarded (no arithmetic overflow or underflow exceptions are ever generated). A comparison to zero and to the bound value in register S2 is also performed, returning bits in the bit string that correspond to the following conditions: ou (out of range or unordered), ib (in range or on boundary), in (in range), and ob (out of range or on boundary or unordered). If the S2 operand is negative, ou, ib, in, and ob are set to zero. If execution of **fcmpu** is attempted while SFU1 is disabled, a floating-point unimplemented exception is taken.

The returned comparison results can be used by branch on bit instructions (**bb0, bb1**) to synthesize conditional branch on comparison operations (branch equal, branch greater, etc).

NOTE

The MC88110 performs IEEE 754 infinity arithmetic directly in hardware. Thus, unlike the MC88100, the MC88110 does not treat infinity (∞) as a reserved operand and infinity does not cause an exception.

When the processor is in TCFP mode (i.e., one of the TCFP bits in the FPCR is set), reserved operands do not cause SFU1 exceptions; instead, when a reserved operand is detected, the hardware delivers a default result approximating the IEEE defined result.

See **Section 4 Floating-Point Implementation** for more information on the floating-point implementation.

Result String:

31														18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	uge	ul	ule	ug	lg	ue	ob	in	ib	ou	ge	lt	le	gt	ne	eq	leg	un

Bits 31–18 are not guaranteed to be zeros in future implementations.

uge:	unordered or greater than or equal
ul:	unordered or less than
ule:	unordered or less than or equal
ug:	unordered or greater than
lg:	less than or greater than
ue:	unordered or equal

ob   out of range or on boundary

in   in range

ib   in range or on boundary

ou   out of range

ge:	true (1) if and only if (rS1) $\geq$ (rS2) (signed greater than or equal)
lt:	true (1) if and only if (rS1) $<$ (rS2) (signed less than)
le:	true (1) if and only if (rS1) $\leq$ (rS2) (signed less than or equal)
gt:	true (1) if and only if (rS1) $>$ (rS2) (signed greater than)
ne:	true (1) if and only if (rS1) $\neq$ (rS2) (not equal)
eq:	true (1) if and only if (rS1) $=$ (rS2) (equal)
leg:	true (1) if and only if the two operands are less than, greater than, or equal
un:	true (1) if and only if the two operands are unordered (i.e., one or both operands is a NaN).

10

Instruction Encoding:

Floating-Point Category—Triadic Register

31					26	25			21	20			16	15	14			11	10	9	8	7	6	5	4		0
1	0	0	0	0	1			D				S1		R	0	1	1	1		T1		T2	0	1		S2	

D: Destination Register (rD)

S_1:   Source 1 Register (rS1 or xS1)

S_2:   Source 2 Register (rS2 or xS2)

R: 0—Source Operands in GRF

            1—Source Operands in XRF

T1:   Source 1 Operand Size

T2:   Source 2 Operand Size

            Note: For the T1 and T2 Fields:

            00—Single-Precision

            01—Double-Precision

            10—Double-Extended-Precision

            11—Unused

10

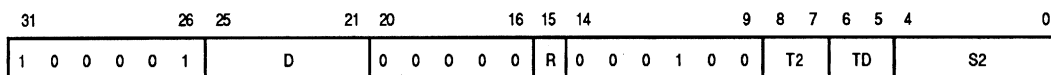# fcvt

Operation: Destination ← Convert (Source 2)

Assembler	fcvt.sd	rD,rS2	fcvt.sd	xD,xS2
Syntax:	fcvt.ds	rD,rS2	fcvt.ds	xD,xS2
			fcvt.sx	xD,xS2
			fcvt.dx	xD,xS2
			fcvt.xs	xD,xS2
			fcvt.xd	xD,xS2

Exceptions: Floating-Point Reserved Operand
Floating-Point Overflow
Floating-Point Underflow
Floating-Point Inexact (if not masked)
Floating-Point Unimplemented

Description: The **fcvt** instruction checks the contents of the S2 register for a reserved operand (NaN, denormalized, or unnormalized number). If a reserved operand is found, a floating-point reserved operand exception is taken. If no reserved operand is found, the floating-point value contained in the S2 register is converted from the precision designated by the source type specifier to the precision designated by the destination type specifier with the sign of the source operand being strictly preserved. The result of the conversion is placed in the D register. Both the original operand and the converted operand must reside in the same register file.

Instruction Encoding:

Floating-Point Category—Triadic Register

31					26	25				21	20					16	15	14					9	8	7	6	5	4			0
1	0	0	0	0	1			D			0	0	0	0	0	0	R	0	0	0	1	0	0	T2		TD			S2	

D:	Destination Register (rD or xD)
S2:	Source 2 Register (rS2 or xS2)
R:	0—Source Operand in GRF
	1—Source Operand in XRF
T2:	Source 2 Operand Size
TD:	Destination Operand Size
	Note: For the T2 and TD Fields:
	00—Single-Precision
	01—Double-Precision
	10—Double-Extended-Precision
	11—Unused

10

fdiv

Floating-Point Divide

Operation: Destination ← Source 1 / Source 2

Assembler Syntax:				
fdiv.sss	rD,rS1,rS2	**fdiv.sss**	xD,xS1,xS2
fdiv.ssd	rD,rS1,rS2	**fdiv.ssd**	xD,xS1,xS2
fdiv.sds	rD,rS1,rS2	**fdiv.sds**	xD,xS1,xS2
fdiv.sdd	rD,rS1,rS2	**fdiv.sdd**	xD,xS1,xS2
fdiv.dss	rD,rS1,rS2	**fdiv.dss**	xD,xS1,xS2
fdiv.dsd	rD,rS1,rS2	**fdiv.dsd**	xD,xS1,xS2
fdiv.dds	rD,rS1,rS2	**fdiv.dds**	xD,xS1,xS2
fdiv.ddd	rD,rS1,rS2	**fdiv.ddd**	xD,xS1,xS2
		fdiv.ssx	xD,xS1,xS2
		fdiv.sdx	xD,xS1,xS2
		fdiv.sxs	xD,xS1,xS2
		fdiv.sxd	xD,xS1,xS2
		fdiv.sxx	xD,xS1,xS2
		fdiv.dsx	xD,xS1,xS2
		fdiv.ddx	xD,xS1,xS2
		fdiv.dxs	xD,xS1,xS2
		fdiv.dxd	xD,xS1,xS2
		fdiv.dxx	xD,xS1,xS2
		fdiv.xss	xD,xS1,xS2
		fdiv.xsd	xD,xS1,xS2
		fdiv.xsx	xD,xS1,xS2
		fdiv.xds	xD,xS1,xS2
		fdiv.xdd	xD,xS1,xS2
		fdiv.xdx	xD,xS1,xS2
		fdiv.xxs	xD,xS1,xS2
		fdiv.xxd	xD,xS1,xS2
		fdiv.xxx	xD,xS1,xS2

Exceptions: Floating-Point Reserved Operand
Floating-Point Divide-by-Zero
Floating-Point Overflow
Floating-Point Underflow
Floating-Point Inexact (if not masked)
Floating-Point Unimplemented

Description: The **fdiv** instruction checks the contents of the S1 and S2 registers for reserved operands (NaNs, denormalized or unnormalized numbers). If reserved operands are found, a floating-point reserved operand exception is taken. If no reserved operands are found, the S1 operand is divided by the S2 operand according to the IEEE 754 standard, and the result is placed in the D register. Any combination of single-, double-, and double-extended-precision operands can be specified. Attempting to divide by zero causes a floating-point divide-by-zero exception. Exception conditions also

10

occur when an overflow, underflow, or inexact result is detected. If execution of **fdiv** is attempted while SFU1 is disabled, a floating-point unimplemented exception is taken.
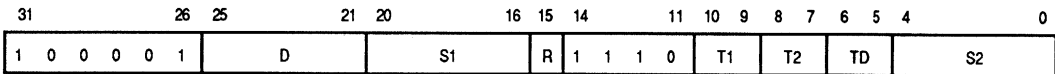
NOTE

The MC88110 performs IEEE 754 infinity arithmetic directly in hardware. Thus, unlike the MC88100, the MC88110 does not treat infinity (∞) as a reserved operand and infinity does not cause an exception.

See **Section 4 Floating-Point Implementation** for more information on the floating-point implementation.

Instruction Encoding:

Floating-Point Category—Triadic Register

31					26	25		21	20		16	15	14		11	10	9	8	7	6	5	4		0
1	0	0	0	0	1		D			S1		R	1	1	1	0		T1		T2		TD		S2

D:	Destination Register (rD or xD)
S1:	Source 1 Register (rS1 or xS1)
R:	0—Source Operands in GRF
	1—Source Operands in XRF
T1:	Source 1 Operand Size
T2:	Source 2 Operand Size
TD:	Destination Operand Size
	Note: For the T1, T2, and TD Fields:
	00—Single-Precision
	01—Double-Precision
	10—Double-Extended-Precision
	11—Unused
S2:	Source 2 Register (rS2 or xS2)

10

Operation: Destination ← (bit number) of Source 2 Scanned for First Bit Clear

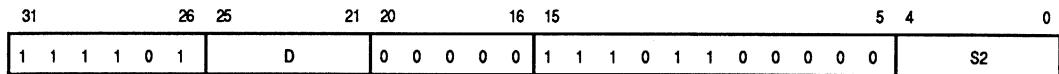**Assembler**   **ff0**   rD,rS2
Syntax:

Exceptions: None

Description: The **ff0** instruction scans the S2 register from the most significant bit to the least significant bit. The D register is loaded with the bit number of the first bit that is found clear. Zero corresponds to the least significant bit and 31 corresponds to the most significant bit. If no bits are found clear, the D register is loaded with 32.

Instruction Encoding:

Bit Field Category—Triadic Register

31					26	25				21	20				16	15										5	4				0
1	1	1	1	0	1			D			0	0	0	0	0	1	1	1	0	1	1	0	0	0	0	0			S2		

D: Destination Register (rD)
S2:   Source 2 Register (rS2)

10

Operation: Destination ← (bit number) of Source 2 Scanned for First Bit Set

Assembler Syntax: **ff1**   rD,rS2

Exceptions: None

Description: The **ff1** instruction scans the S2 register from the most significant bit to the least significant bit. The D register is loaded with the bit number of the first bit that is found set. Zero corresponds to the least significant bit and 31 corresponds to the most significant bit. If no bits are found set, the D register is loaded with 32.

Instruction Encoding:

Bit Field Category—Triadic Register

31					26	25			21	20					16	15											5	4				0
1	1	1	1	0	1		D			0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0			S2			
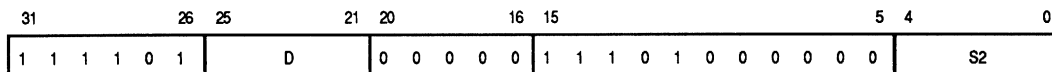
D: Destination Register (rD)
S2:   Source 2 Register (rS2)

fldcr   Load From Floating-Point Control Register   fldcr

Operation: Destination ← Floating-Point Control Register

Assembler   **fldcr**   rD,fcrS
Syntax:

Exceptions: Floating-Point Privilege Violation
Floating-Point Unimplemented

Description: The **fldcr** instruction moves the contents of the floating-point unit control register specified by the FCRS field to the specified D register. Floating-point control register **fcr0** is a privileged register and can only be accessed in the supervisor mode. Floating-point control registers **fcr62** and **fcr63** are floating-point control and status registers, respectively, and can be accessed in either the supervisor or user mode.

Floating-point control registers **fcr1** through **fcr61** are unimplemented and privileged. An **fldcr** instruction which addresses these registers causes a floating-point unimplemented exception in supervisor mode, or a floating-point privilege violation exception in user mode.

Refer to **Section 4 Floating-Point Implementation** for more information on floating-point control registers.

Instruction Encoding:

Floating-Point Category—Control Register

31				26	25		21	20				16	15			11	10		5	4				0
1	0	0	0	0	0	D		0	0	0	0	0	0	1	0	0	1	FCRS		0	0	0	0	0
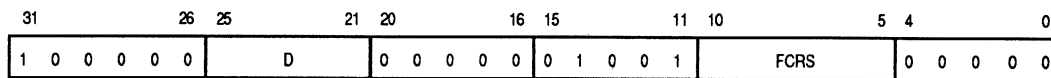
D: Destination Register (rD)
FCRS: Floating-Point Control Register Source (fcrS)

10

Operation: Destination ← Float (Source 2)

| **Assembler**	**flt.ss**	rD,rS2	**flt.ss**	xD,rS2
Syntax:	**flt.ds**	rD,rS2	**flt.ds**	xD,rS2
			flt.xs	xD,rS2

Exceptions: Floating-Point Inexact (if not masked)
Floating-Point Unimplemented

Description: The **flt** instruction converts the signed integer contained in the S2 register to floating-point representation. The result is placed in the D register. Since the S2 register contains an integer, it can only be specified as single precision; however, the D register can be single-, double-, or double-extended-precision. Double-extended-precision results cannot be stored in the general register file.

See **Section 4 Floating-Point Implementation** for more information on the floating-point implementation.

Instruction Encoding:

Floating-Point Category—Triadic Register, Destination in General Register File
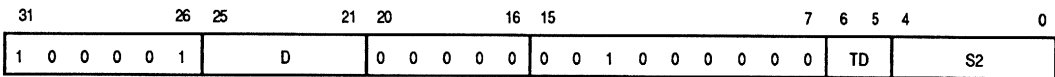
31					26	25				21	20					16	15										7	6	5	4				0
1	0	0	0	0	1			D			0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	TD				S2			

Floating-Point Category—Triadic Register, Destination in Extended Register File

31					26	25				21	20					16	15										7	6	5	4				0
1	0	0	0	0	1			D			0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	TD				S2			
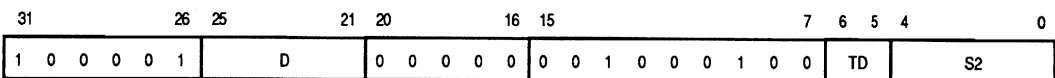
D:	Destination Register (rD or xD)
TD:	Destination Operand Size
	Note: For the TD Field:
	00—Single-Precision
	01—Double-Precision
	10—Double-Extended-Precision
	11—Unused
S2:	Source 2 Register (rS2)

10

Operation: Destination ← Source 1 * Source 2

Assembler	**fmul.sss**	rD,rS1,rS2	**fmul.sss**	xD,xS1,xS2
Syntax:	**fmul.ssd**	rD,rS1,rS2	**fmul.ssd**	xD,xS1,xS2
	fmul.sds	rD,rS1,rS2	**fmul.sds**	xD,xS1,xS2
	fmul.sdd	rD,rS1,rS2	**fmul.sdd**	xD,xS1,xS2
	fmul.dss	rD,rS1,rS2	**fmul.dss**	xD,xS1,xS2
	fmul.dsd	rD,rS1,rS2	**fmul.dsd**	xD,xS1,xS2
	fmul.dds	rD,rS1,rS2	**fmul.dds**	xD,xS1,xS2
	fmul.ddd	rD,rS1,rS2	**fmul.ddd**	xD,xS1,xS2
			fmul.ssx	xD,xS1,xS2
			fmul.sdx	xD,xS1,xS2
			fmul.sxs	xD,xS1,xS2
			fmul.sxd	xD,xS1,xS2
			fmul.sxx	xD,xS1,xS2
			fmul.dsx	xD,xS1,xS2
			fmul.ddx	xD,xS1,xS2
			fmul.dxs	xD,xS1,xS2
			fmul.dxd	xD,xS1,xS2
			fmul.dxx	xD,xS1,xS2
			fmul.xss	xD,xS1,xS2
			fmul.xsd	xD,xS1,xS2
			fmul.xsx	xD,xS1,xS2
			fmul.xds	xD,xS1,xS2
			fmul.xdd	xD,xS1,xS2
			fmul.xdx	xD,xS1,xS2
			fmul.xxs	xD,xS1,xS2
			fmul.xxd	xD,xS1,xS2
			fmul.xxx	xD,xS1,xS2

Exceptions: Floating-Point Reserved Operand
Floating-Point Overflow
Floating-Point Underflow
Floating-Point Inexact (if not masked)
Floating-Point Unimplemented

Description: The **fmul** instruction checks the contents of the S1 and S2 registers for reserved operands. If reserved operands are found, a floating-point reserved operand exception is taken. If no reserved operands are found, the S1 and S2 operands are multiplied according to the IEEE 754 standard, and the result is placed in the D register. Exception conditions also occur when an overflow, underflow, or inexact result is detected. If execution of **fmul** is attempted while SFU1 is disabled, a floating-point unimplemented exception is taken.
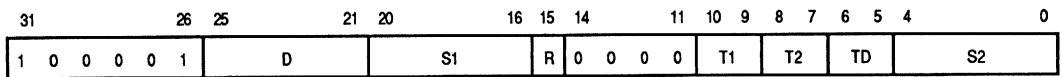
NOTE

The MC88110 performs IEEE 754 infinity arithmetic directly in hardware. Thus, unlike the MC88100, the MC88110 does not treat infinity (∞) as a reserved operand and infinity does not cause an exception.

When the processor is in TCFP mode (i.e., one of the TCFP bits in the FPCR is set), reserved operands do not cause SFU1 exceptions; instead, when a reserved operand is detected, the hardware delivers a default result approximating the IEEE defined result. See **Section 4 Floating-Point Implementation** for more details on TCFP mode.

Instruction Encoding:

Floating-Point Category—Triadic Register

31					26	25		21	20		16	15	14		11	10	9	8	7	6	5	4		0
1	0	0	0	0	1		D			S1		R	0	0	0	0		T1		T2		TD		S2

D:	Destination Register (rD or xD)
S1:	Source 1 Register (rS1 or xS1)
R:	0—Source Operands in GRF
	1—Source Operands in XRF
T1:	Source 1 Operand Size
T2:	Source 2 Operand Size
TD:	Destination Operand Size
	Note: For T1, T2, and TD Fields:
	00—Single-Precision
	01—Double-Precision
	10—Double-Extended-Precision
	11—Unused
S2:	Source 2 Register (rS2 or xS2)

10

fsqrt

Operation: Destination ← Square Root (Source 2)

Assembler   **fsqrt.ss**   rD,rS2   **fsqrt.ss**   xD,xS2
Syntax: **fsqrt.sd**   rD,rS2   **fsqrt.sd**   xD,xS2
                fsqrt.ds   rD,rS2   **fsqrt.ds**   xD,xS2
                fsqrt.dd   rD,rS2   **fsqrt.dd**   xD,xS2
                                            fsqrt.sx   xD,xS2
                                            fsqrt.dx   xD,xS2
                                            fsqrt.xs   xD,xS2
                                            fsqrt.xd   xD,xS2
                                            fsqrt.xx   xD,xS2

Exceptions: Floating-Point Unimplemented

Description: The **fsqrt** instruction calculates the square root of the floating-point value contained in the S2 register and places the result in the specified D register. The S2 and D registers must reside in the same register file.

NOTE

The MC88110 does not implement the square root instruction in hardware. Instead, executing the **fsqrt** instruction causes a floating-point unimplemented exception, and a software handler is provided to emulate the square root operation.

Instruction Encoding:
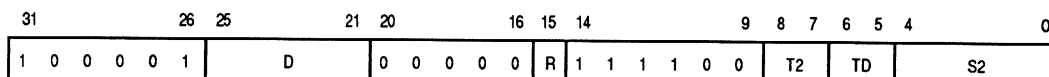
Floating-Point Category—Triadic Register

31					26	25				21	20					16	15	14					9	8	7	6	5	4			0
1	0	0	0	0	1			D			0	0	0	0	0	R	1	1	1	1	0	0		T2		TD			S2		

D: Destination Register (rD or xD)
S2:   Source 2 Register (rS2 or xS2)
R: 0—Source Operand in GRF
         1—Source Operand in XRF
T2:   Source 2 Operand Size
TD: Destination Operand Size
         Note: For the T2 and TD Fields:
             00—Single-Precision
             01—Double-Precision
             10—Double-Extended-Precision
             11—Unused

Operation: Floating-Point Control Register ← Destination

Assembler   **fstcr**   rS1,fcrD
Syntax:

Exceptions: Floating-Point Privilege Violation
                         Floating-Point Unimplemented

Description: The **fstcr** instruction moves the contents of the S1 register to the floating-point unit control register specified by the FCRD field. Floating-point control register **fcr0** is a privileged register and can only be accessed in the supervisor mode. Floating-point control registers **fcr62** and **fcr63** are the floating-point control and status registers, respectively, and can be accessed in either the supervisor or user mode.

Floating-point control registers **fcr1** through **fcr61** are unimplemented and privileged. An **fstcr** instruction which addresses any of these registers causes a floating-point unimplemented exception in supervisor mode, or a floating-point privilege violation exception in user mode.

Instruction Encoding:

Floating-Point Category—Control Register

31					26	25				21	20		16	15				11	10		5	4		0
1	0	0	0	0	0	0	0	0	0	0	S1			1	0	0	0	1	FCRD			S2		
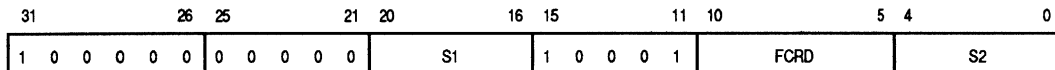
     S1:   Source 1 Register (rS1)
     FCRD: Floating-Point Control Destination Register
     S2:   Source 2 Register (rS2)
                      Note: S1 and S2 must contain the same value.

10

fsub

Operation: Destination ← Source 1–Source 2

Assembler	**fsub.sss**	rD,rS$_1$,rS$_2$	**fsub.sss**	xD,xS$_1$,xS$_2$
Syntax:	**fsub.ssd**	rD,rS$_1$,rS$_2$	**fsub.ssd**	xD,xS$_1$,xS$_2$
	fsub.sds	rD,rS$_1$,rS$_2$	**fsub.sds**	xD,xS$_1$,xS$_2$
	fsub.sdd	rD,rS$_1$,rS$_2$	**fsub.sdd**	xD,xS$_1$,xS$_2$
	fsub.dss	rD,rS$_1$,rS$_2$	**fsub.dss**	xD,xS$_1$,xS$_2$
	fsub.dsd	rD,rS1,rS2	**fsub.dsd**	xD,xS1,xS2
	fsub.dds	rD,rS$_1$,rS$_2$	**fsub.dds**	xD,xS$_1$,xS$_2$
	fsub.ddd	rD,rS1,rS2	**fsub.ddd**	xD,xS1,xS2
			fsub.ssx	xD,xS$_1$,xS$_2$
			fsub.sdx	xD,xS$_1$,xS$_2$
			fsub.sxs	xD,xS$_1$,xS$_2$
			fsub.sxd	xD,xS$_1$,xS$_2$
			fsub.sxx	xD,xS$_1$,xS$_2$
			fsub.dsx	xD,xS$_1$,xS$_2$
			fsub.ddx	xD,xS$_1$,xS$_2$
			fsub.dxs	xD,xS$_1$,xS$_2$
			fsub.dxd	xD,xS$_1$,xS$_2$
			fsub.dxx	xD,xS$_1$,xS$_2$
			fsub.xss	xD,xS$_1$,xS$_2$
			fsub.xsd	xD,xS$_1$,xS$_2$
			fsub.xsx	xD,xS$_1$,xS$_2$
			fsub.xds	xD,xS$_1$,xS$_2$
			fsub.xdd	xD,xS$_1$,xS$_2$
			fsub.xdx	xD,xS$_1$,xS$_2$
			fsub.xxs	xD,xS$_1$,xS$_2$
			fsub.xxd	xD,xS$_1$,xS$_2$
			fsub.xxx	xD,xS$_1$,xS$_2$

10

Exceptions: Floating-Point Reserved Operand
Floating-Point Overflow
Floating-Point Underflow
Floating-Point Inexact (if not masked)
Floating-Point Unimplemented

Description: The **fsub** instruction checks the contents of the S1 and S2 registers for reserved operands. If reserved operands are found, a floating-point reserved operand exception is taken. If no reserved operands are found, the S2 operand is subtracted from the S1 operand according to the IEEE 754 standard, and the result is placed in the D register. If execution of **fsub** is attempted while the FPU is disabled, a floating-point unimplemented exception is taken. Exception conditions also occur when an overflow, underflow, or inexact result is detected.

NOTE

The MC88110 performs IEEE 754 infinity arithmetic directly in hardware. Thus, unlike the MC88100, the MC88110 does not treat infinity (∞) as a reserved operand and infinity does not cause an exception.

When the processor is in TCFP mode (i.e., one of the TCFP bits in the FPCR is set), reserved operands do not cause SFU1 exceptions; instead, when a reserved operand is detected, the hardware delivers a default result approximating the IEEE defined result. See **Section 4 Floating-Point Implementation** for more details on TCFP mode.

Instruction Encoding:

Floating-Point Category—Triadic Register

31					26	25			21	20				16	15	14			11	10	9	8	7	6	5	4				0
1	0	0	0	0	1		D				S1				R	0	1	1	0	T1		T2		TD			S2			

D:	Destination Register (rD or xD)
S1:	Source 1 Register (rS1 or xS1)
R:	0—Source Operands in GRF
	1—Source Operands in XRF
T1:	Source 1 Operand Size
T2:	Source 2 Operand Size
TD:	Destination Operand Size
	Note: For the T1, T2, and TD Fields:
	00—Single-Precision
	01—Double-Precision
	10—Double-Extended-Precision
	11—Unused
S2:	Source 2 Register (rS2 or xS2)

10

Operation: Destination ← Floating-Point Control Register
Floating-Point Control Register ← Source 1

**Assembler
Syntax:**   **fxcr**   rD,rS1,fcrS/D

Exceptions: Floating-Point Privilege Violation
Floating-Point Unimplemented

Description: The **fxcr** instruction transfers the contents of the S1 register to the floating-point unit control register specified by the FCRS/D field and transfers the contents of the **fcr**S/D register to the D register. Floating-point control register **fcr0** is a privileged register and can only be accessed in the supervisor mode. Floating-point control registers **fcr62** and **fcr63** are the floating-point control and status registers, respectively, and can be accessed in either the supervisor or the user mode.

Floating-point control registers **fcr1** through **fcr61** are unimplemented and privileged. An **fxcr** instruction which address any of these registers causes a floating-point unimplemented exception in supervisor mode, or a floating-point privilege violation exception in user mode.

Instruction Encoding:

Floating-Point Category—Control Register

31					26	25			21	20			16	15			11	10					5	4				0
1	0	0	0	0	0		D				S1			1	1	0	0	1		FCRS/D						S2		
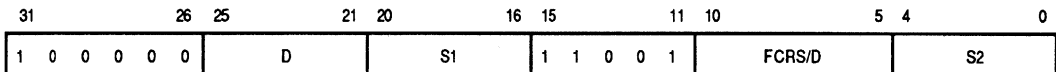
D:	Destination Register (rD)
S1:	Source 1 Register (rS1)
FCRS/D:	Floating-Point Control Register Source/Destination (**fcr**S/D)
S2:	Source 2 Register (rS2)
	Note: The S1 and S2 fields must contain the same value.

10

illop

illop

Operation: None

Assembler   **illop1**
Syntax: **illop2**
    illop3

Exceptions: Unimplemented Opcode

Description: The three **illop** instructions perform no user visible operation but unconditionally cause an unimplemented opcode exception to be taken.

Instruction Encoding:

Flow Control Category—Triadic Register

31					26	25										16	15													2	1	0
1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	IL		

    IL: Identifies the illegal opcode instruction
        01—Illegal Opcode 1
        10—Illegal Opcode 2
        11—Illegal Opcode 3

Operation: Destination ← Round (Source 2)

Assembler   **int.ss**   rD,rS2   **int.ss**   rD,xS2
Syntax: **int.sd**   rD,rS2   **int.sd**   rD,xS2
                                                   int.sx   rD,xS2

Exceptions: Floating-Point Reserved Operand
                      Floating-Point Inexact (if not masked)
                      Floating-Point Integer Conversion Overflow
                      Floating-Point Unimplemented

Description: The **int** instruction converts the floating-point number in the S2 register to a signed 32-bit integer using the rounding mode specified in the floating-point status register (FPSR). The result is placed in the D register. If the result exceeds 32-bits, then a floating-point integer conversion overflow exception is taken. If reserved operands are found, a floating-point reserved operand exception is taken. If execution of **int** is attempted while the FPU is disabled, a floating-point unimplemented exception is taken.

See **Section 4 Floating-Point Implementation** for more information on the floating-point implementation.

Instruction Encoding:

Floating-Point Category—Triadic Register

31					26	25				21	20					16	15	14					9	8	7	6	5	4				0
1	0	0	0	0	1			D			0	0	0	0	0	R	1	0	0	1	0	0	T2		0	0			S2		

      D: Destination Register (rD)
      R: 0—Source Operand in GRF
                        1—Source Operand in XRF
      T2:   Source 2 Operand Size
                        00—Single-Precision
                        01—Double-Precision
                        10—Double-Extended-Precision
                        11—Unused
      S2:   Source 2 Register (rS2 or xS2)

10

jmp
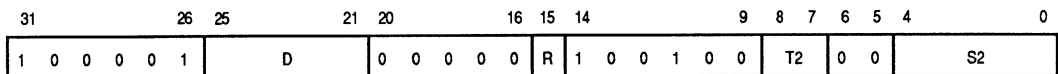
Unconditional Jump

jmp

Operation: Transfer program flow to Source 2

Assembler   jmp   rS2
Syntax: jmp.n   rS2

Exceptions: None

Description: The **jmp** instruction performs an unconditional transfer of program flow to the absolute address contained in the S2 register. The two least significant bits of that register are masked in order to force the target address to an instruction (word) boundary. The **.n** (delayed branch) option causes the instruction following the **jmp.n** instruction to execute before the target instruction.

To ensure future compatibility, the instruction following a **jmp.n** instruction should not be a trap, jump, branch or any other instruction that modifies the instruction pointers. Using such an instruction constitutes a programming error which is not detected.

The **jmp** instruction can be used to return from subroutines as shown in the following example:

> **jmp r1**

When branching or jumping to a subroutine, the **bsr** and **jsr** instructions, respectively, place the return address in register **r1** as a hardware convention; therefore, specifying register **r1** as the S2 register for a subroutine **jmp** instruction causes program flow to be transferred to the return address.

Instruction Encoding:
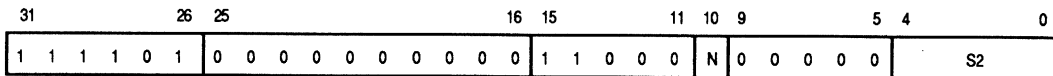
Flow Control Category—Triadic Register

31					26	25											16	15				11	10	9					5	4				0
1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	N	0	0	0	0	0			S2			

N: 0—Next sequential instruction suppressed
         1—Next sequential instruction executed before branch is taken

S2:   Source 2 Register (rS2)

10

Operation: Transfer program flow to Source 2

r1 ← address of first instruction (second if .n) after **jsr**

Assembler Syntax:	**jsr**	**rS2**
	jsr.n	**rS2**

Exceptions: None

Description: The **jsr** instruction performs an unconditional transfer of program control to a target address and saves the return address in register **r1**. The **jsr** target address is contained in the S2 register. The two least-significant bits of that register are masked, forcing the target address to an instruction (word) boundary. The return address is the address of the instruction following the **jsr** instruction. The **.n** (delayed branch) option causes the instruction following the **jsr.n** instruction to execute before the jump target instruction.

When the **.n** option is specified, the return address written to **r1** is the address of the second instruction following the **jsr.n** instruction. If the instruction in the delay slot uses **r1** as an operand, the contents of **r1** will be the new return address. If the instruction in the delay slot modifies **r1**, its result will supersede the **jsr** return address.

To ensure future compatibility, the instruction following a **jsr.n** instruction should not be a trap, jump, branch or any other instruction that modifies the instruction pointers. Using such an instruction constitutes a programming error which is not detected.

Instruction Encoding:

Flow Control Category—Triadic Register

31			26	25										16	15				11	10	9					5	4				0
1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	1	N	0	0	0	0	0			S2		
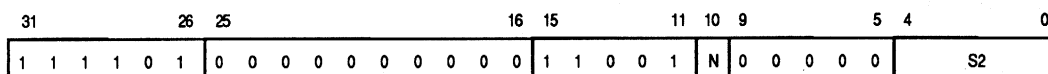
N: 0—Next sequential instruction suppressed

1—Next sequential instruction executed before branch is taken

S2:   Source 2 Register (rS2)

Operation: Destination Register ← Memory Location

Assembler Syntax:

UNSCALED		UNSCALED		SCALED	
ld.b	rD,rS1,SI16	ld.b	rD,rS1,rS2	ld.b	rD,rS1[rS2]
ld.bu	rD,rS1,SI16	ld.bu	rD,rS1,rS2	ld.bu	rD,rS1[rS2]
ld.h	rD,rS1,SI16	ld.h	rD,rS1,rS2	ld.h	rD,rS1[rS2]
ld.hu	rD,rS1,SI16	ld.hu	rD,rS1,rS2	ld.hu	rD,rS1[rS2]
ld	rD,rS1,SI16	ld	rD,rS1,rS2	ld	rD,rS1[rS2]
ld.d	rD,rS1,SI16	ld.d	rD,rS1,rS2	ld.d	rD,rS1[rS2]
		ld.b.usr	rD,rS1,rS2	ld.b.usr	rD,rS1[rS2]
		ld.bu.usr	rD,rS1,rS2	ld.bu.usr	rD,rS1[rS2]
		ld.h.usr	rD,rS1,rS2	ld.h.usr	rD,rS1[rS2]
		ld.hu.usr	rD,rS1,rS2	ld.hu.usr	rD,rS1[rS2]
		ld.usr	rD,rS1,rS2	ld.usr	rD,rS1[rS2]
		ld.d.usr	rD,rS1,rS2	ld.d.usr	rD,rS1[rS2]
ld	xD,rS1,SI16	ld	xD,rS1,rS2	ld	xD,rS1[rS2]
ld.d	xD,rS1,SI16	ld.d	xD,rS1,rS2	ld.d	xD,rS1[rS2]
ld.x	xD,rS1,SI16	ld.x	xD,rS1,rS2	ld.x	xD,rS1[rS2]
		ld.usr	xD,rS1,rS2	ld.usr	xD,rS1[rS2]
		ld.d.usr	xD,rS1,rS2	ld.d.usr	xD,rS1[rS2]
		ld.x.usr	xD,rS1,rS2	ld.x.usr	xD,rS1[rS2]

Exceptions: Data Access Exception
Misaligned Access Exception (if not masked)
Privilege Violation (**.usr** option only)

Description: The **ld** instruction reads data from the specified memory location and loads it into the D register. The memory base address is contained in the S1 register. Added to this base is either an unsigned 32-bit word index contained in the S2 register or a 16-bit immediate index. An immediate index is sign-extended if the processor is in signed immediate mode or zero-extended if the processor is in unsigned mode. An index in the S2 register can be scaled or unscaled. Scaled index mode is indicated by enclosing the S2 register within square brackets. When a **ld** instruction is being executed, the D register is marked "in use" in the register scoreboard until the memory fetch completes.

The **ld** instruction with no options specifies word (32-bit) operation. The **.b** option specifies signed byte (8-bit) operation, **.bu** specifies unsigned byte (8-bit), **.h** specifies signed half-word (16-bit), **.hu** specifies unsigned half-word (16-bit), **.d** specifies double word (64-bit), and **.x** specifies quad word (128-bit). For the scaled index modes, the scale factor is determined by the size option of the instruction. Operations that are byte, half-word, word, double word, and quad word in size have scale factors of 1, 2, 4, 8, and 16, respectively.

10

NOTE

Although the extended register file is 80 bits wide in the MC88110, all memory accesses from the extended register file must be aligned to quad-word (128-bit) boundaries. Thus, the **ld.x** instruction provides a scale factor of 16.

When the MODE bit of the PSR is set, the memory access is normally to supervisor memory space; when MODE is clear, the memory access is normally to user memory space. The **.usr** option specifies that the memory access must be to the user address space regardless of the MODE bit. The **.usr** option is only available in supervisor mode.

If the D register is **r0**, a special cache control operation (touch, allocate, or flush) may be performed. See **Section 6 Instruction and Data Caches** for more information on these operations.

Instruction Encoding:

Load/Store/Exchange Category—Register Indirect with Immediate Index (GRF)

31	28	27 26	25	21	20	16	15	0
0 0 0 1		TY	D		S1		SI16	
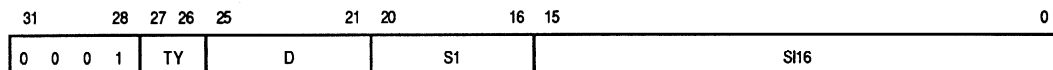
Load/Store/Exchange Category—Register Indirect with Immediate Index
(GRF—Unsigned Load)
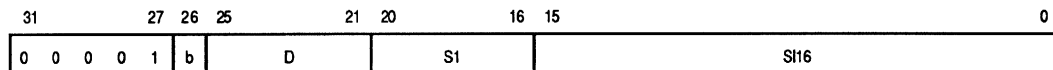
31	27	26	25	21	20	16	15	0
0 0 0 0 1		b	D		S1		SI16	

Load/Store/Exchange Category—Register Indirect with Immediate Index (XRF—Single)

31	26	25	21	20	16	15	0
0 0 0 0 0 1		D		S1		SI16	

Load/Store/Exchange Category—Register Indirect with Immediate Index (XRF—Double)

31	26	25	21	20	16	15	0
0 0 0 0 0 0		D		S1		SI16	

10

Load/Store/Exchange Category—Register Indirect with Immediate Index (XRF—Extended)

31	26	25	21	20	16	15	0
0 0 1 1 1 1		D		S1		SI16	

Load/Store/Exchange Category—Register Indirect with Scaled or Unscaled Index (Signed Load)

31	27	26	25	21	20	16	15	12	11 10	9	8	7	5	4	0
1 1 1 1 0		R	D		S1		0 0 0 1	TY	S	U	0 0 0		S2	
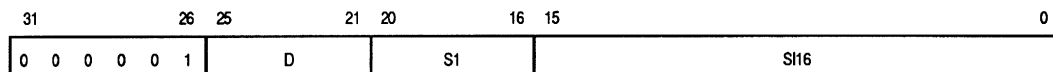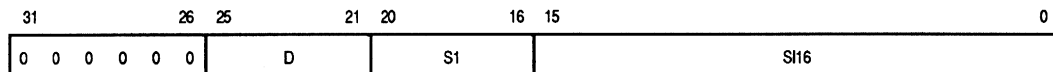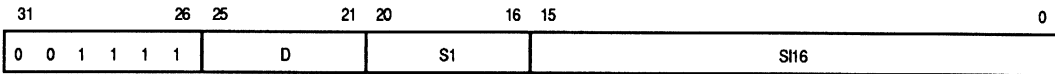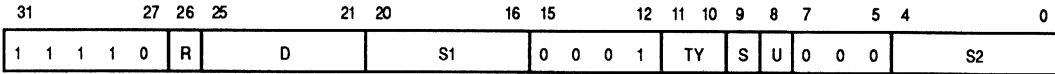
Load/Store/Exchange Category—Register Indirect with Scaled or Unscaled Index (Unsigned Load)

31	26	25	21	20	16	15	11	10	9	8	7	5	4	0
1 1 1 1 0 1		D		S1		0 0 0 0 1	B	S	U	0 0 0		S2	
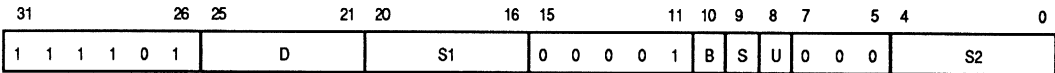
TY (GRF, R=1):	00—Double Word
	01—Word
	10—Half-Word
	11—Byte
TY (R=0):	00—Double Word
	01—Word
	10—Quad Word
	11—Unused
B:	0—Half-Word
	1—Byte
R:	0—Destination Register in XRF
	1—Destination Register in GRF
S:	0—Unscaled Index
	1—Scaled Index
D:	Destination Register (rD or xD)
S1:	Source 1 Register (rS1)
SIMM16:	16-Bit Immediate Index
U:	0—access per user/supervisor bit in PSR (normal mode)
	1—access user space regardless of PSR
S2:	Source 2 Register (rS2)

10

Operation: Destination ← Source 1 + Source 2

Assembler	**lda.h**	$rD,rS_1[rS_2]$
Syntax:	**lda**	$rD,rS_1[rS_2]$
	lda.d	$rD,rS_1[rS_2]$
	lda.x	$rD,rS_1[rS_2]$

Exceptions: None

Description: The **lda** instruction creates a memory address from the specified operands. The memory base address is contained in the S1 register. Added to this base is an unsigned 32-bit scaled word index contained in the S2 register. Note that scaled index mode is indicated by square brackets enclosing the S2 register. The resulting address is placed in the D register. This address is not checked for alignment relative to the operation type.

The **lda** instruction with no options specifies word (32-bit) operation. The **.h** option specifies half-word (16-bit), **.d** specifies double word (64-bit), and **.x** specifies quad word (128-bit) operation. The scale factor is determined by the size option of the instruction. Operations that are half-word, word, double word, and quad word in size have scale factors of 2, 4, 8, and 16, respectively.

NOTE

Although the extended register file is 80 bits wide in the MC88110, all memory accesses from the extended register file must be aligned to quad-word (128-bit) boundaries. Thus, the **lda.x** instruction has been added to provide a scale factor of 16. (The **.b** option, along with all unscaled versions of the **lda** instruction, is not included in the MC88110 instruction set).

10

Instruction Encoding:

Load/Store/Exchange Category—Register Indirect With Scaled Index

31					26	25		21	20		16	15		12	11	10	9		5	4		0
1	1	1	1	0	1		D			S1		0	0	1	1	TY	1	0	0	0	0	S2
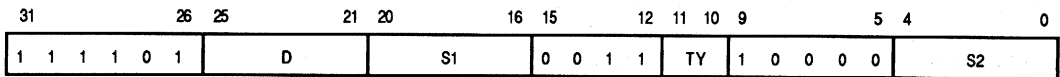
TY:
  00—Double Word
  01—Word
  10—Half-Word
  11—Quad Word*

D: Destination Register (rD)
S1: Source 1 Register (rS1)
S2: Source 2 Register (rS2)

* Encoding for **lda.x** on the MC88110 replaces the encoding for **lda.b** on the MC88100.

10

ldcr

**Load From Control Register
(Privileged Instruction)**

Operation: Destination Register ← Control Register

**Assembler   ldcr   rD,crS
Syntax:**

Exceptions: Privilege Violation
Unimplemented Opcode

Description: The **ldcr** instruction moves data to the D register from the integer unit
control register specified by the CRS field. Integer unit control registers may only be
accessed in the supervisor mode; a privilege violation occurs if they are accessed in
user mode. If the **crS** field specifies a reserved control register, then an unimplemented
opcode exception occurs.

Instruction Encoding:

Load/Store/Exchange Category —Control Register

31					26	25			21	20					16	15				11	10		5	4				0
1	0	0	0	0	0		D			0	0	0	0	0	0	0	1	0	0	0		CRS		0	0	0	0	0
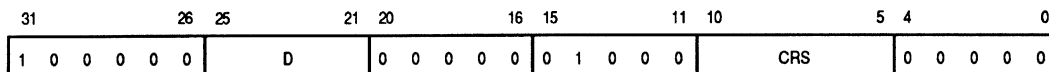
D: Destination Register (rD)
CRS: Control Register Source (crS)

10

Operation: (bit field) Destination ← (bit field) of Source 1

Assembler **mak** rD,rS1,W5<O5>
Syntax: **mak** rD,rS1,rS2

Exceptions: None

Description: The **mak** instruction extracts a bit field from the S1 register. The bit field, whose width is specified by the W5 field, begins with the least significant bit of the S1 register. The extracted field is placed in the D register, offset from the least significant bit by the amount specified in the O5 field. Any bits outside of the field are cleared. If any bits of the extracted field fall outside of the D register, those bits are ignored.

For triadic register addressing, bits 9–5 and bits 4–0 of the S2 register are used for the W5 and O5 fields, respectively, and the rest of S2 is ignored.

The following illustration shows the operation of the **mak** instruction:
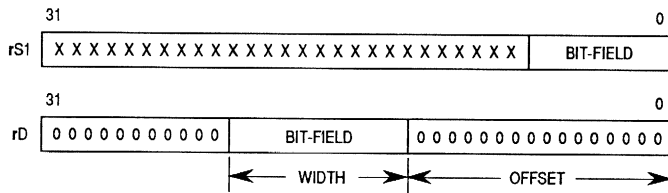
When the W5 field contains all zeros (specifying a width of 32 bits), the **mak** instruction operates as a shift left instruction. The O5 field specifies the number of positions to shift, and the low-order bits are zero filled in the D register. The following illustration shows an example of a shift left operation using the **mak rD,rS1,30<5>** instruction:
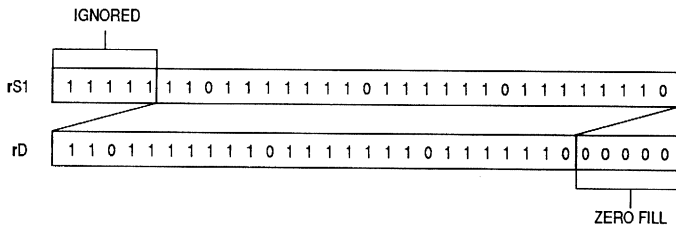
10

Instruction Encoding:

Bit Field Category—Register with 10-Bit Immediate

| 31 | | | | 26 | 25 | | | 21 | 20 | | | 16 | 15 | | | | | 10 | 9 | | W5 | 5 | 4 | | O5 | 0 |

```
31         26 25       21 20        16 15              10 9        5 4         0
+-----------+-----------+------------+-----------------+----------+-----------+
|1 1 1 1 0 0|     D     |     S1     |1  0  1  0  0  0 |    W5    |    O5     |
+-----------+-----------+------------+-----------------+----------+-----------+
```

Bit Field Category—Triadic Register

```
31         26 25       21 20        16 15                       5 4         0
+-----------+-----------+------------+--------------------------+-----------+
|1 1 1 1 0 1|     D     |     S1     |1 0 1 0 0 0 0 0 0 0 0 0    |    S2     |
+-----------+-----------+------------+--------------------------+-----------+
```

D:	Destination Register (rD)
S1:	Source 1 Register (rS1)
W5:	5-bit unsigned integer denoting a bit-field width (0 denotes 32 bits)
O5:	5-bit unsigned integer denoting a bit-field offset
S2:	Source 2 Register (rS2)

10
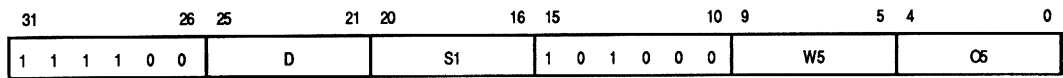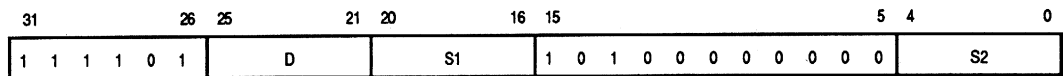
mask   Logical Mask Immediate   mask

Operation: Destination ← Source 1 ∧ IMM16

Assembler   **mask**   rD,rS1,IMM16
Syntax: **mask.u**   rD,rS1,IMM16

Exceptions: None

Description: The **mask** instruction logically ANDs the lower 16 bits of the S1 register with the unsigned 16-bit immediate value encoded in the instruction and places the result in the lower 16 bits of the D register. The upper 16 bits of the D register are cleared. If the **.u** (upper word) option is specified, the upper 16 bits of the S1 register are ANDed with the 16-bit immediate value and the result is placed in the upper 16 bits of the D register. In this case, the lower 16 bits of the D register are cleared.

Instruction Encoding:
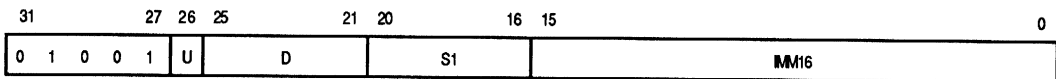
Logical Category—Register with 16-Bit Immediate

31				27	26	25			21	20			16	15		0
0	1	0	0	1	U		D				S1				IMM16	

U: 0—Apply IMM16 to Bits 15–0 of S1
         1—Apply IMM16 to Bits 31–16 of S1
D: Destination Register (rD)
S1:   Source 1 Register (rS1)
IMM16:   16-Bit Unsigned Immediate Operand

10

Operation: Destination ← Source 2

Assembler   **mov.s**   rD,xS2
Syntax: **mov.d**   rD,xS2
           mov.s   xD,rS2
           mov.d   xD,rS2
           mov   xD,xS2

Exceptions: Floating-Point Unimplemented

Description: The **mov** instruction moves the data from the S2 register to the D register using the specified precision for both the source and destination registers. When data is moved within the extended register file, the entire contents of the register are moved, so it is not necessary to specify an operand precision. When single- or double-precision data is moved from the general register file to the extended register file, the value of the unused bits is undefined. Double-precision operands require a register pair when moved into the general register file, and no double-extended-precision values may be moved into or out of the general register file.

The **mov** instruction may not be used to move data between registers in the general register file. Also, the **mov** instruction may not be used to move double-extended-precision data between the two register files. If a double-extended-precision value must be stored in the general register file, it should be moved via memory using **st** and **ld** instructions.

10

Instruction Encoding:

Floating-Point Category—Triadic Register (Destination Operand in GRF)
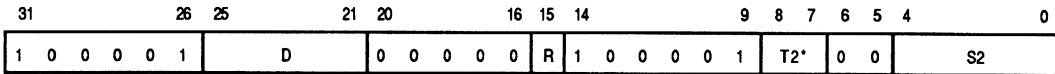
31					26	25				21	20					16	15							9	8	7	6	5	4			0
1	0	0	0	0	1			D			0	0	0	0	0	0	1	1	0	0	0	0	0	T2		0	0			S2	

Floating-Point Category—Triadic Register (Destination Operand in XRF)

31					26	25				21	20					16	15	14					9	8	7	6	5	4			0
1	0	0	0	0	1			D			0	0	0	0	0	0	R	1	0	0	0	0	1	T2*		0	0			S2	

D:	Destination Register (rD or xD)
S2:	Source 2 Register (rS2 or xS2)
R:	0—Source Operand in GRF
	1—Source Operand in XRF
T2:	Source 2 Operand Size
	00—Single Precision
	01—Double Precision
	10—Unused
	11—Unused
T2*:	If R = 1 then T2* = 10 (Double-Extended-Precision in XRF), else T2* = T2

10

Operation: Destination ← Source 1 * Source 2

Assembler   **muls**   rD,rS1,rS2
Syntax:

Exceptions: Integer Overflow

Description: The **muls** instruction multiplies the signed 32-bit integer value in the S1 register by the signed 32-bit integer value in the S2 register using 32-bit two's complement multiplication. The result is written into the D register. If the product cannot be represented as a signed 32-bit result, an overflow exception is taken and no result is written into D.

Instruction Encoding:
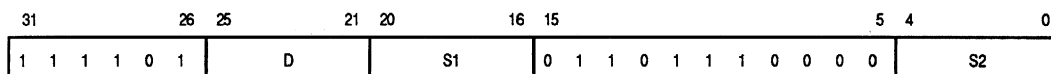
Integer Category—Triadic Register

31					26	25			21	20			16	15										5	4			0
1	1	1	1	0	1		D				S1			0	1	1	0	1	1	1	0	0	0	0		S2		

        D: Destination Register (rD)
        S1:   Source 1 Register (rS1)
        S2:   Source 2 Register (rS2)

10

mulu   Unsigned Integer Multiply   mulu

Operation: Destination ← Source 1 * Source 2

Assembler   **mulu**   rD,rS1,rS2
Syntax: **mulu**   rD,rS1,IMM16
                mulu.d   rD,rS1,rS2

Exceptions: None

Description: The **mulu** instruction multiplies the data in the S1 register by either the data in the S2 register or by the unsigned, zero-extended 16-bit immediate value. Thirty-two-bit two's complement multiplication is used. The least significant 32 bits of the product are stored into the D register. This instruction was referred to as "**mul**" in the MC88100 User's Manual.

If the **.d** (double) option is specified, the 64-bit product is placed in register pair D:D+1.

NOTE

Unlike the MC88100, this instruction does not cause a floating-point unimplemented exception when SFU1 is disabled.

Instruction Encoding:

Integer Category—Register with 16-Bit Immediate

31				26	25			21	20			16	15				0
0	1	1	0	1	1		D			S1					IMM16		

Integer Category—Triadic Register

31				26	25		21	20		16	15						9	8	7		5	4		0
1	1	1	1	0	1		D			S1	0	1	1	0	1	1	0	d	0	0	0		S2	

10
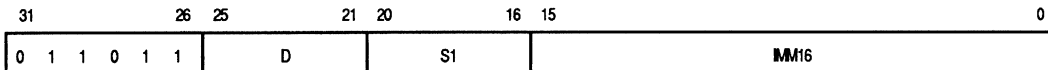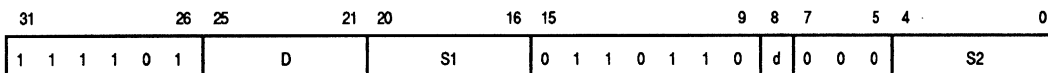
D:	Destination Register (rD)
S1:	Source 1 Register (rS1)
IMM16:	16-Bit Zero-Extended Immediate Operand
d:	0—Single-Word Destination
	1—Double-Word Destination
S2:	Source 2 Register (rS2)

Operation: Destination ← Round-Nearest (Source 2)

Assembler	**nint.ss**	**rD,rS2**		**nint.ss**	**rD,xS2**
Syntax:	**nint.sd**	**rD,rS2**		**nint.sd**	**rD,xS2**
				nint.sx	**rD,xS2**

Exceptions: Floating-Point Reserved Operand
Floating-Point Integer Conversion Overflow
Floating-Point Inexact (if not masked)
Floating-Point Unimplemented

Description: The **nint** instruction converts the floating-point number contained in the S2 register to a signed 32-bit integer using the IEEE 754 round-to-nearest rounding method and places the result in the D register. If the result exceeds 32-bits, a floating-point integer conversion overflow exception is taken. If a reserved operand is found, a floating-point reserved operand exception is taken.

See **Section 4 Floating-Point Implementation** for more information on the floating-point implementation.

Instruction Encoding:

Floating-Point Category—Triadic Register

31					26	25			21	20					16	15	14						9	8	7	6	5	4			0
1	0	0	0	0	1		D			0	0	0	0	0	R	1	0	1	0	0	0			T2		0	0		S2		

D: Destination Register (rD)
R: 0—Source Operand in GRF
1—Source Operand in XRF
T2: Source 2 Operand Size
00—Single-Precision
01—Double-Precision
10—Double-Extended-Precision
11—Unused
S2: Source 2 Register (rS2 or xS2)

10
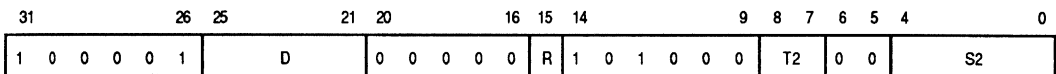
Operation: Destination ← Source 1 V Source 2

Assembler	**or**	rD,rS1,rS2
Syntax:	**or.c**	rD,rS1,rS2
	or	rD,rS1,IMM16
	or.u	rD,rS1,IMM16

Exceptions: None

Description: For triadic register addressing, the contents of the S1 register are logically ORed with the contents of the S2 register. The result is stored in the D register. If the **.c** (complement) option is specified, the S2 operand is complemented before being ORed.

For register with immediate addressing, the contents of the S1 register are ORed with the unsigned 16-bit immediate operand. The result is stored in the lower 16 bits of D, and the upper 16 bits of S1 are copied unchanged to D. If the **.u** (upper word) option is specified, the upper 16 bits of the S1 operand are ORed with the immediate operand, and the result is stored in the upper 16 bits of D. In this case, the lower 16 bits of S1 are copied unchanged to D.

Instruction Encoding:

Logical Category—Register with 16-Bit Immediate

31				27	26	25		21	20		16	15		0
0	1	0	1	1	U		D			S1			IMM16	

Logical Category—Triadic Register

31				26	25		21	20		16	15				11	10	9		5	4		0
1	1	1	1	0	1		D			S1	0	1	0	1	1	C	0	0	0	0	0	S2
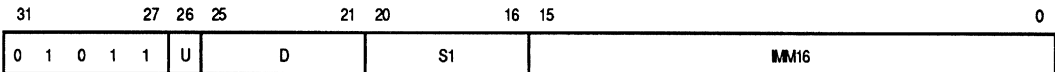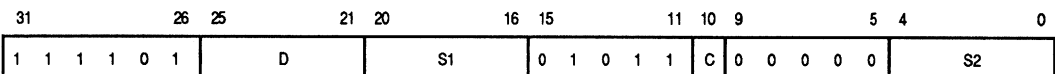
U:	0—OR IMM16 with Bits 15–0 of S1
	1—OR IMM16 with Bits 31–16 of S1
D:	Destination Register (rD)
S1:	Source 1 Register (rS1)
IMM16:	16-Bit Unsigned Immediate Operand
C:	0—Second operand not complemented before the operation
	1—Second operand complemented before the operation
S2:	Source 2 Register (rS2)

10

padd   Pixel Add   padd

Operation: Destination ← Source 1 + Source 2

Assembler   **padd.b**   rD,rS1,rS2
Syntax: **padd.h**   rD,rS1,rS2
               padd   rD,rS1,rS2
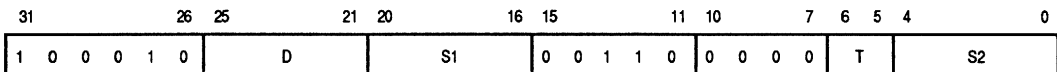
Exceptions: Graphics (SFU2) Unimplemented

Description: The **padd** instruction adds the 8-, 16-, or 32-bit pixel fields contained in the S1:S1+1 register pair to equivalent fields in the S2:S2+1 register pair and places the result in the registers D:D+1. The addition is carried out using modulo 2^T arithmetic, where T is the number of bits in each field. Overflow and underflow conditions wrap around within the fields rather than causing exceptions.

Instruction Encoding:

Graphics Category—Triadic Register

31					26	25			21	20				16	15				11	10			7	6	5	4			0
1	0	0	0	1	0		D				S1				0	0	1	0	0	0	0	0	0	T		S2			

         D: Destination Register (rD)
         S_1:   Source 1 Register (rS1)
         S_2:   Source 2 Register (rS2)
         T: Bit Field Size
                      00—Unused
                      01—8-bit
                      10—16-bit
                      11—32-bit

10

padds

Pixel Add and Saturate

padds

Operation: Destination ← Source 1 + Source 2
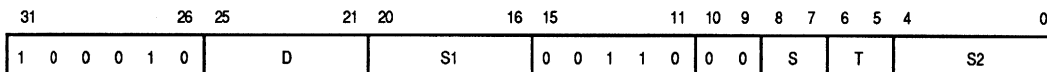
Assembler
Syntax:

padds.u.b rD,rS1,rS2
padds.u.h rD,rS1,rS2
padds.u   rD,rS1,rS2
padds.us.b   rD,rS1,rS2
padds.us.h   rD,rS1,rS2
padds.us   rD,rS1,rS2
padds.s.b rD,rS1,rS2
padds.s.h rD,rS1,rS2
padds.s   rD,rS1,rS2

Exceptions: Graphics (SFU2) Unimplemented

Description: The **padds** instruction adds the 8-, 16-, or 32-bit pixel fields contained in the S1:S1+1 register pair to equivalent fields in the S2:S2+1 register pair and places the result in the registers D:D+1. The addition is carried out using signed (**.s**), unsigned (**.u**), or mixed (**.us**) saturation arithmetic. See **Section 5 Graphics Processing Unit** for more information on saturation arithmetic.

Instruction Encoding:

Graphics Category—Triadic Register

31				26	25		21	20		16	15			11	10	9	8	7	6	5	4		0
1	0	0	0	1	0	D		S1			0	0 1 0 0			0	0	S		T		S2	
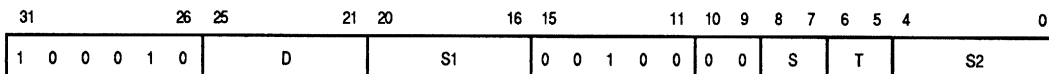
D:	Destination Register (rD)
S$_1$:	Source 1 Register
S$_2$:	Source 2 Register
T:	Bit Field Size
	00—Unused
	01—8-Bit
	10—16-Bit
	11—32-Bit
S:	Saturation Mode
	00—Nonsaturating (Unused for **padds**)
	01—Unsigned + Unsigned = Unsigned Saturation
	10—Unsigned + Signed = Signed Saturation
	11—Signed + Signed = Signed Saturation

10

pcmp Pixel Compare pcmp

Operation: Destination ← Source 1 :: Source 2

Assembler   **pcmp**   rD,rS1,rS2
Syntax:

Exceptions: Graphics (SFU2) Unimplemented

Description: The **pcmp** instruction compares the two 32-bit fields contained in the S1:S1+1 register pair to the corresponding fields in the S2:S2+1 register pair using unsigned arithmetic. An 8-bit result string is returned in the D register. The format of the result string is as follows:

rD[3:0]:	0				
rD[4]:	(rS1[63:32]	≥ rS2[63:32])	and	(rS1[31:0]	≥ rS2[31:0]])
rD[5]:	(rS1[63:32]	< rS2[63:32])	and	(rS1[31:0]	< rS2[31:0]])
rD[6]:	(rS1[63:32]	≥ rS2[63:32])	and	(rS1[31:0]	< rS2[31:0]])
rD[7]:	(rS1[63:32]	< rS2[63:32])	and	(rS1[31:0]	≥ rS2[31:0]])
rD[8]:	!rD[4]				
rD[9]:	!rD[5]				
rD[10]:	!rD[6]				
rD[11]:	!rD[7]				
rD[31:12]:	0				

Instruction Encoding:
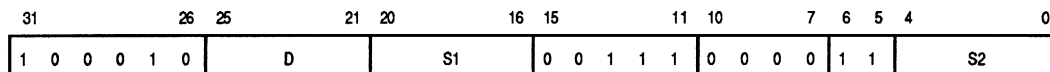
Graphics Category—Triadic Register

31					26	25			21	20			16	15			11	10			7	6	5	4		0
1	0	0	0	1	0		D				S1			0	0	1	1	1	0	0	0	0	1	1	S2	

D: Destination Register (rD)
S1:   Source 1 Register (rS1)
S2:   Source 2 Register (rS2)

10

pmul   Pixel Multiply   pmul

Operation: Destination ← Source 1 x Source 2

Assembler   **pmul**   rD,rS1,rS2
Syntax:

Exceptions: Graphics (SFU2) Unimplemented

Description: The **pmul** instruction multiplies the 32-bit value in the S1 register by the 64-bit value in the S2:S2+1 register pair and places the 64 least significant bits of the result in the D:D+1 register pair. The multiplication is carried out using unsigned arithmetic.

NOTE

The **pmul** instruction is intended to be used in conjunction with the **punpk** and **ppack** instructions. See **Section 5 Graphics Processing Unit** for further details and an illustration of typical usage.

Instruction Encoding:

Graphics Category—Triadic Register

31					26	25			21	20				16	15					11	10				7	6	5	4				0
1	0	0	0	1	0		D				S1				0	0	0	0	0		0	0	0	0		0	0		S2			

      D: Destination Register (rD)
      S_1:   Source 1 Register (rS1)
      S_2:   Source 2 Register (rS2)

10

ppack   Pixel Truncate, Insert and Pack   ppack

Operation: Destination ← Truncated and Packed (Source 1 and Source 2)

Assembler   **ppack.32.b**   rD,rS1,rS2
Syntax: **ppack.16.h**   rD,rS1,rS2
    ppack.32.h   rD,rS1,rS2
    ppack.8   rD,rS1,rS2
    ppack.16   rD,rS1,rS2
    ppack.32   rD,rS1,rS2

Exceptions: Graphics (SFU2) Unimplemented

Description: The **ppack** instruction takes the $(t^*r)/64$ most significant bits of each of the fields (field size = **t**) in the S2:S2+1 register pair and concatenates them, resulting in a field of width **r**. This field replaces the most significant bits of the data from the S1:S1+1 register pair and the result is stored in the D:D+1 register pair. The data in the D:D+1 register pair is then rotated left by **r** bits.

The values of **t** and **r** are specified in the instruction. The value of **r** can be 8, 16, or 32 bits, and is specified in the first option field after the mnemonic. The value of **t** can be byte, half-word, or word. Byte and half-word are specified in the second option field by **b** or **h**, respectively. Word is specified by leaving the second option field empty. The following table shows the possible values of $(t^*r)/64$ for all possible combinations of **t** and **r**.

		r		
		8	16	32
	8	x	x	4
t	16	x	4	8
	32	4	8	16

x = undefined operation

ppack   Pixel Truncate, Insert and Pack   ppack

Operation: Destination ← Truncated and Packed (Source 1 and Source 2)

Assembler   **ppack.32.b**   rD,rS1,rS2
Syntax: **ppack.16.h**   rD,rS1,rS2
    ppack.32.h   rD,rS1,rS2
    ppack.8   rD,rS1,rS2
    ppack.16   rD,rS1,rS2
    ppack.32   rD,rS1,rS2

Exceptions: Graphics (SFU2) Unimplemented

Description: The **ppack** instruction takes the $(t^*r)/64$ most significant bits of each of the fields (field size = **t**) in the S2:S2+1 register pair and concatenates them, resulting in a field of width **r**. This field replaces the most significant bits of the data from the S1:S1+1 register pair and the result is stored in the D:D+1 register pair. The data in the D:D+1 register pair is then rotated left by **r** bits.

The values of **t** and **r** are specified in the instruction. The value of **r** can be 8, 16, or 32 bits, and is specified in the first option field after the mnemonic. The value of **t** can be byte, half-word, or word. Byte and half-word are specified in the second option field by **b** or **h**, respectively. Word is specified by leaving the second option field empty. The following table shows the possible values of $(t^*r)/64$ for all possible combinations of **t** and **r**.

		r		
		8	16	32
	8	x	x	4
t	16	x	4	8
	32	4	8	16

x = undefined operation

Instruction Encoding:

Graphics Category—Triadic Register

31				26	25			21	20			16	15				11	10			7	6	5	4			0
1	0	0	0	1	0		D			S1			0	1	1	0	0		R			T			S2		

D:	Destination Register (rD)
S1:	Source 1 Register (rS1)
S2:	Source 2 Register (rS2)
T:	Bit-Field Size
	00—Unused
	01—8-Bit
	10—16-Bit
	11—32-Bit
R:	Rotation Size
	0000—Rotate 64-bit register pair left 0 (or 64) bits
	0001—Rotate left 4 bits
	0010—Rotate left 8 bits
	r r r r—Rotate left (r r r r x 4) bits
	Note: Only rotations of 8, 16, and 32 are valid for **ppack**

prot   Pixel Rotate Left   prot

Operation: Destination ← Source 1 rotated left

Assembler   **prot**   rD,rS1,<O6>
Syntax: **prot**   rD,rS1,rS2

Exceptions: Graphics (SFU2) Unimplemented

Description: The **prot** instruction rotates the value in the S1:S1+1 register pair to the left by either the number of bits specified in the S2 register or by the 6-bit immediate value specified in the O6 field, with the result being placed in the D:D+1 register pair. The rotation count must be an integral multiple of 4 bits in the range of 0 to 60 bits. If a nonintegral multiple of 4 bits is specified, the rotation count will be truncated to the next lower integral multiple of 4 bits. A count greater than 60 bits will be truncated to less than or equal to 60 bits.

Instruction Encoding:

Graphics Category—Triadic Register

31					26	25				21	20				16	15					11	10			7	6	5	4			0
1	0	0	0	1	0		D				S1					0	1	1	1	1		0	0	0	0	0	0		S2		
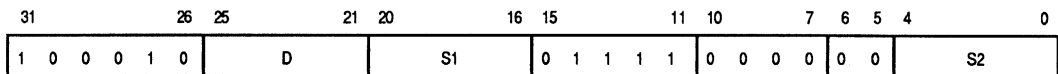
Graphics Category—Register with 6-Bit Immediate

31					26	25				21	20				16	15					11	10			7	6	5	4			0
1	0	0	0	1	0		D				S1					0	1	1	1	0			R			0	0	0	0	0	0
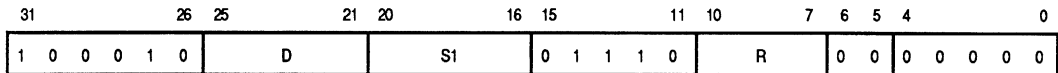
D: Destination Register (rD)
S1:   Source 1 Register (rS1)
S2:   Source 2 Register (rS2)
R: Rotation Size
    0000—Rotate 64-bit register pair left 0 (or 64) bits
    0001—Rotate left 4 bits
    0010—Rotate left 8 bits
    r r r r—Rotate left (r r r r x 4) bits

NOTE

The nibble-wise (4-bit) rotation count is specified in bits 5–2 of rS2. Other bits (31–6, 1–0) are ignored, but should be set to zero to assure future compatibility.

psub
psub

Operation: Destination ← Source 1–Source 2

Assembler   **psub.b**   rD,rS1,rS2
Syntax: **psub.h**   rD,rS1,rS2
               psub   rD,rS1,rS2

Exceptions: Graphics (SFU2) Unimplemented

Description: The **psub** instruction subtracts the 8-, 16-, or 32-bit pixel fields in the S2:S2+1 register pair from equivalent fields in the S1:S1+1 register pair and places the result in the D:D+1 register pair. The subtraction is carried out using modulo 2^T arithmetic, where T is the number of bits in each field. Overflow and underflow conditions wrap around within the fields, rather than causing exceptions.

Instruction Encoding:

Graphics Category—Triadic Register

31				26	25		21	20		16	15			11	10			7	6	5	4		0
1	0	0	0	1	0	D		S1			0	0	1	1	0	0	0	0	0	T		S2	

D: Destination Register (rD)
S_1:   Source 1 Register (rS1)
S_2:   Source 2 Register (rS2)
T: Bit Field Size
                    00—Unused
                    01—8-bit
                    10—16-bit
                    11—32-bit

10

psubs   Pixel Subtract and Saturate   psubs

Operation: Destination ← Source 1–Source 2

Assembler   **psubs.u.b**   rD,rS1,rS2
Syntax: **psubs.u.h**   rD,rS1,rS2
          psubs.u   rD,rS1,rS2
          psubs.us.b   rD,rS1,rS2
          psubs.us.h   rD,rS1,rS2
          psubs.us   rD,rS1,rS2
          psubs.s.b   rD,rS1,rS2
          psubs.s.h   rD,rS1,rS2
          psubs.s   rD,rS1,rS2

Exceptions: Graphics (SFU2) Unimplemented

Description: The **psubs** instruction subtracts the 8-, 16-, or 32-bit pixel fields in the S2:S2+1 register pair from equivalen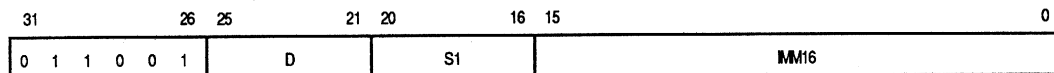t fields in the S1:S1+1 register pair and places the result in the D:D+1 register pair. The subtraction is carried out using signed (**.s**), unsigned (**.u**), or mixed (**.us**) saturation arithmetic. See **Section 5 Graphics Processing Unit** for more information on saturation arithmetic.

Instruction Encoding:

Graphics Category—Triadic Register

31					26	25			21	20				16	15					11	10	9	8	7	6	5	4				0
1	0	0	0	1	0		D				S1				0	0	1	1	0		0	0	S		T			S2		

          D: Destination Register (rD)
          S1:   Source 1 Register (rS1)
          S2:   Source 2 Register (rS2)
          T: Bit Field Size
                    00—Unused
                    01—8-bit
                    10—16-bit
                    11—32-bit
          S: Saturation Mode
                    00—Non-Saturating (Unused for **psubs**)
                    01—Unsigned–Unsigned = Unsigned Saturation
                    10—Unsigned–Signed = Signed Saturation
                    11—Signed–Signed = Signed Saturation

punpk   Pixel Unpack   punpk

Operation: Destination ← Unpack (Source 1)

Assembler   **punpk.n**   rD,rS1
Syntax: **punpk.b**   rD,rS1
                 punpk.h   rD,rS1

Exceptions: Graphics (SFU2) Unimplemented

Description: The **punpk** instruction places 4-, 8-, or 16-bit fields from the S1 register into the lower half of fields twice as large (8, 16, or 32 bits) with zero fill. These fields are then concatenated to form a 64-bit result that is placed in the D:D+1 register pair.

Instruction Encoding:

Graphics Category—Triadic Register

31				26	25			21	20				16	15				11	10			7	6 5	4				0
1	0	0	0	1 0		D				S1				0	1	1	0	1	0	0	0 0		T	0	0	0	0 0
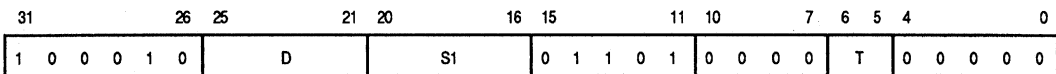
D: Destination Register (rD)
S1:   Source 1 Register (rS1)
T: Bit Field Size
                 00—4-Bit (Nibble)
                 01—8-Bit (Byte)
                 10—16-Bit (Half-Word)
                 11—32-Bit (Word)

10

rot

Rotate Register

rot

Operation: Destination ← Source 1 rotated by O5

Assembler   **rot**   rD,rS1,<O5>
Syntax: **rot**   rD,rS1,rS2

Exceptions: None

Description: The **rot** instruction rotates the bits in the S1 register to the right by the number of bits specified in the O5 field. The result is placed in the D register. For triadic register addressing, bits 4–0 of the S2 register are used as the O5 field. Bits 9–5 in the S2 register should be zero to guarantee future compatibility; the other bits in the S2 register are ignored.

Instruction Encoding:

Bit Field Category—Register with 10-Bit Immediate

31					26	25				21	20				16	15					10	9				5	4				0
1	1	1	1	0	0			D					S1			1	0	1	0	1	0	0	0	0	0	0			O5		

Bit Field Category—Triadic Register

31					26	25				21	20				16	15									5	4				0
1	1	1	1	0	1			D					S1			1	0	1	0	1	0	0	0	0	0	0			S2		
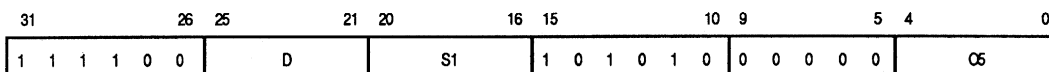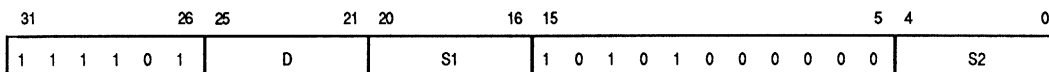
D: Destination Register (rD)
S1:   Source 1 Register (rS1)
O5:   5-bit unsigned integer denoting a bit-field offset
S2:   Source 2 Register (rS2)

10

Operation: PSR ← EPSR
                XIP ← EXIP
                NIP ← ENIP

Assembler   **rte**
Syntax:

Exceptions: Privilege Violation

Description: The **rte** instruction provides an orderly termination of exception processing. First, the **rte** instruction causes the machine to be serialized to guarantee that all exception handler instructions complete. Next, the PSR is restored from the EPSR, and the machine is placed in the correct mode (user or supervisor). Finally, the XIP is restored from the EXIP and the instruction at that address is fetched. If the excepting instruction was in the delay slot of a branch as indicated by the D bit in the EXIP, then the NIP is restored from the ENIP, the instruction at the address in the ENIP is fetched, and normal execution resumes. If the instruction was not in a branch delay slot, the NIP is calculated by normal instruction processing when execution resumes. An **rte** instruction executed in the user mode causes a privilege violation.

See **Section 7 Exceptions** for more information on exceptions and the side effects of executing an **rte** instruction.

Instruction Encoding:

Flow Control Category—Triadic Register

31					26	25											16	15											5	4					0
1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0

10
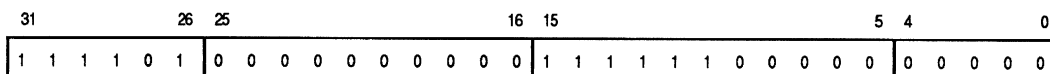
Operation: Destination ← (Source 1 V (Bit Field of 1's))

Assembler   **set**   rD,rS1,W5<O5>
Syntax: **set**   rD,rS1,rS2

Exceptions: None

Description: The **set** instruction reads the data from register S1 and inserts a field of ones into the data. The result is placed in the D register. The width of the bit field is specified by the W5 field, and the offset of the bit field from bit zero of the S1 data is specified by the O5 field. A W5 field of all zeros specifies a 32-bit wide bit field. If any bits of the inserted bit field extend beyond bit 31 of the S1 data, those bits are ignored.

For triadic register addressing, bits 9–5 and bits 4–0 of the S2 register are used for the W5 and O5 fields, respectively.

The following illustration shows the operation of the **set rD,rS1,5<16>** instruction. In this example, W5 contains 5 and O5 contains 16, thereby placing a field of 5 ones in bits 16 through 20 of the S1 data.
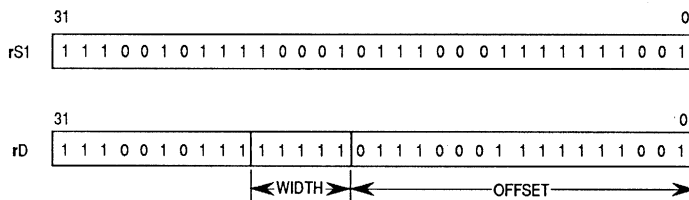
```
       31                                                          0
rS1  | 1 1 1 0 0 1 0 1 1 1 1 0 0 0 1 0 1 1 1 0 0 0 1 1 1 1 1 1 1 0 0 1 |

       31                                                          0
rD   | 1 1 1 0 0 1 0 1 1 1 | 1 1 1 1 1 | 0 1 1 1 0 0 0 1 1 1 1 1 1 1 0 0 1 |
                          |◄WIDTH►|◄────────OFFSET────────►|
```

10

Instruction Encoding:

Bit Field Category—Register with 10-Bit Immediate

31					26	25				21	20				16	15						10	9			5	4			0
1	1	1	1	0	0			D					S1			1	0	0	0	1	0			W5				O5		

Bit-Field Category—Triadic Register

31					26	25				21	20				16	15										5	4			0
1	1	1	1	0	1			D					S1			1	0	0	0	1	0	0	0	0	0	0			S2	
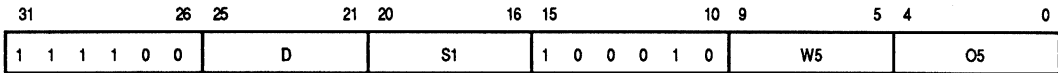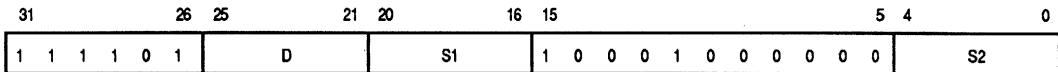
D: Destination Register (rD)
S1: Source 1 Register (rS1)
W5: Unsigned 5-bit integer denoting a bit-field width (0 denotes 32 bits)
O5: Unsigned 5-bit integer denoting a bit-field offset
S2: Source 2 Register (rS2)

10

Operation: Memory Location ← Source Register (specified as rD)

Assembler Syntax:

UNSCALED		UNSCALED		SCALED	
st.b	rD,rS1,SIMM16	st.b	rD,rS1,rS2	st.b	rD,rS1[rS2]
st.h	rD,rS1,SIMM16	st.h	rD,rS1,rS2	st.h	rD,rS1[rS2]
st	rD,rS1,SIMM16	st	rD,rS1,rS2	st	rD,rS1[rS2]
st.d	rD,rS1,SIMM16	st.d	rD,rS1,rS2	st.d	rD,rS1[rS2]
		st.b.usr	rD,rS1,rS2	st.b.usr	rD,rS1[rS2]
		st.h.usr	rD,rS1,rS2	st.h.usr	rD,rS1[rS2]
		st.usr	rD,rS1,rS2	st.usr	rD,rS1[rS2]
		st.d.usr	rD,rS1,rS2	st.d.usr	rD,rS1[rS2]
		st.b.wt	rD,rS1,rS2	st.b.wt	rD,rS1[rS2]
		st.h.wt	rD,rS1,rS2	st.h.wt	rD,rS1[rS2]
		st.wt	rD,rS1,rS2	st.wt	rD,rS1[rS2]
		st.d.wt	rD,rS1,rS2	st.d.wt	rD,rS1[rS2]
		st.b.usr.wt	rD,rS1,rS2	st.b.usr.wt	rD,rS1[rS2]
		st.h.usr.wt	rD,rS1,rS2	st.h.usr.wt	rD,rS1[rS2]
		st.usr.wt	rD,rS1,rS2	st.usr.wt	rD,rS1[rS2]
		st.d.usr.wt	rD,rS1,rS2	st.d.usr.wt	rD,rS1[rS2]
st	xD,rS1,SIMM16	st	xD,rS1,rS2	st	xD,rS1[rS2]
st.d	xD,rS1,SIMM16	st.d	xD,rS1,rS2	st.d	xD,rS1[rS2]
st.x	xD,rS1,SIMM16	st.x	xD,rS1,rS2	st.x	xD,rS1[rS2]
		st.usr	xD,rS1,rS2	st.usr	xD,rS1[rS2]
		st.d.usr	xD,rS1,rS2	st.d.usr	xD,rS1[rS2]
		st.x.usr	xD,rS1,rS2	st.x.usr	xD,rS1[rS2]
		st.wt	xD,rS1,rS2	st.wt	xD,rS1[rS2]
		st.d.wt	xD,rS1,rS2	st.d.wt	xD,rS1[rS2]
		st.x.wt	xD,rS1,rS2	st.x.wt	xD,rS1[rS2]
		st.usr.wt	xD,rS1,rS2	st.usr.wt	xD,rS1[rS2]
		st.d.usr.wt	xD,rS1,rS2	st.d.usr.wt	xD,rS1[rS2]
		st.x.usr.wt	xD,rS1,rS2	st.x.usr.wt	xD,rS1[rS2]

10

Exceptions: Data Access Exception
Misaligned Access Exception (if not masked)
Privilege Violation (**.usr** option only)

Description: The **st** instruction writes the contents of a specified register to a specified memory location. The D register contains the data to be stored in memory. The memory base address is contained in the S1 register. The memory location is calculated by adding to the base address either a 16-bit immediate index or the signed 32-bit word index contained in the S2 register. An immediate index is sign-extended if the processor is in the signed immediate mode or zero-extended if the processor is in the unsigned

mode. An index in the S2 register can be scaled or unscaled. Scaled index mode is indicated by enclosing the S2 register within square brackets.

The **st** instruction with no options specifies word (32-bit) operation. The **.b** option specifies byte (8-bit) operation, **.h** specifies half-word (16-bit), **.d** specifies double word (64-bit), and **.x** specifies a quad word (128-bit). For the scaled index modes, the scale factor is determined by the size option of the instruction. Operations that are byte, half-word, word, double-word, and quad word in size have scale factors of 1, 2, 4, 8, and 16, respectively. A **st** instruction with a **.wt** option causes the store operation to write-through the cache and unconditionally update memory.

NOTE

Although the extended register file is 80 bits wide in the MC88110, all memory accesses from the extended register file must be aligned to quad-word (128-bit) boundaries. Thus, the **st.x** instruction provides a scale factor of 16.

Data transfers are always aligned on their size boundary in memory. If a misaligned access is attempted with the MXM bit in the PSR cleared, a misaligned access exception is taken. If the misaligned access exception is disabled (MXM bit is set), the least significant bits of the address are ignored—i.e., the reference is performed to the next lower address boundary which is size aligned.

When the MODE bit of the PSR is set, the memory access is normally to supervisor memory space; when MODE is clear, the memory access is normally to user memory space. The **.usr** option specifies that the memory access must be to the user memory space regardless of the MODE bit in the PSR. The **.usr** option is only available in supervisor mode.

Instruction Encoding:

Load/Store/Exchange Category—Register Indirect with Immediate Index (GRF)

31	28	27 26	25	21	20	16	15	0
0 0 1 0	TY		D		S1		SI16	

Load/Store/Exchange Category—Register Indirect with Immediate Index (XRF—Single)

31	26	25	21	20	16	15	0
0 0 1 1 0 1		D		S1		SI16	

10

Load/Store/Exchange Category—Register Indirect with Immediate Index (XRF—Double)

31						26	25			21	20			16	15				0
0	0	1	1	0	0			D				S1					SI16		

Load/Store/Exchange Category—Register Indirect with Immediate Index (XRF—Extended)

31						26	25			21	20			16	15				0
0	0	1	1	1	0			D				S1					SI16		

Load/Store/Exchange Category—Register Indirect with Scaled or Unscaled Index (Signed Load)

31				27	26	25			21	20			16	15				12	11	10	9	8	7	6	5	4			0
1	1	1	1	0	R		D					S1			0	0	1	0	TY		S	U	T	0	0	S2		

D:	Destination Register (rD or xD)
S1:	Source 1 Register
S2:	Source 2 Register
SIMM16:	16-Bit Signed or Unsigned Immediate Operand
R:	0—Source Operands in XRF
	1—Source Operands in GRF
S:	0—Unscaled Index
	1—Scaled Index
T:	0—Normal Store
	1—Store Through the Cache (Write-Through)
U:	0—Normal Access
	1—Access User Space Regardless of PSR
TY (R=1):	00—Double Word
	01—Word
	10—Half-Word
	11—Byte
TY (R=0):	00—Double Word
	01—Word
	10—Quad Word
	11—Unused

Store To Control Register
(Privileged Instruction)

Operation: Control Register ← Source Register

Assembler **stcr** rS1,crD
Syntax:

Exceptions: Privilege Violation
Unimplemented Opcode

Description: The **stcr** instruction moves the data contained in the S1 register to the integer unit control register specified by **cr**D field. The general control registers can only be accessed in supervisor mode; a privilege violation occurs if they are accessed in user mode. If the **cr**D field specifies a reserved control register, then an unimplemented opcode exception occurs.

Instruction Encoding:

Load/Store/Exchange Category—Control Register

31					26	25					21	20				16	15				11	10				5	4				0
1	0	0	0	0	0	0	0	0	0	0				S1			1	0	0	0	0			CRD					S2		
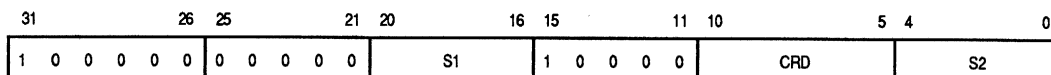
S1:	Source 1 Register (rS1)
CRD:	Control Register Destination (**cr**D)
S2:	Source 2 Register (rS2)
	Note: The S1 and S2 fields must contain the same register number

10

sub   Integer Subtract   sub

Operation: Destination ← Source 1 − Source 2

Assembler   **sub**   rD,rS1,rS2   subtract (without borrow)
Syntax: **sub.ci**   rD,rS1,rS2   subtract and use borrow in
                  sub.co   rD,rS1,rS2   subtract and propagate borrow out
                  sub.cio rD,rS1,rS2   subtract and propagate borrow in and
                                               out
                  sub   rD,rS1,SIMM16   subtract immediate (without borrow)

Exceptions: Integer Overflow

Description: The **sub** instruction subtracts either the data contained in the S2 register or the 16-bit immediate operand from the data contained in the S1 register. The immediate operand is zero-extended in unsigned mode or sign-extended in signed immediate mode. The result of the subtraction is placed in the D register. The carry bit can optionally be used to perform subtract with borrow operations: a cleared carry bit indicates a borrow and a set carry bit indicates no borrow (i.e., effectively, borrow for subtraction is the opposite of carry for addition). If the results cannot be represented as a signed 32-bit integer, an integer overflow exception occurs and the contents of rD and the carry bit are unchanged.

Subtraction is performed by adding the one's complement of the S2 operand and either a constant one or the carry bit to the S1 operand. All 32 bits of the operands participate in the addition (i.e., there is no sign bit). If the carry out of the sign bit position and the carry into the sign bit are not the same, an overflow exception occurs.

10

Instruction Encoding:

Integer Category—Register with 16-Bit Immediate

```
31        26 25       21 20      16 15                              0
┌─────────────┬──────────┬──────────┬─────────────────────────────┐
│ 0 1 1 1 0 1 │    D     │    S1    │            SIMM16           │
└─────────────┴──────────┴──────────┴─────────────────────────────┘
```

Integer Category—Triadic Register

```
31        26 25       21 20      16 15            10 9 8 7    5 4    0
┌─────────────┬──────────┬──────────┬─────────────┬─┬─┬──────┬──────┐
│ 1 1 1 1 0 1 │    D     │    S1    │ 0 1 1 1 0 1 │I│O│0 0 0 │  S2  │
└─────────────┴──────────┴──────────┴─────────────┴─┴─┴──────┴──────┘
```

D:	Destination Register (rD)
S1:	Source 1 Register (rS1)
SIMM16:	16-Bit Signed Immediate Operand
I:	0—Disable Carry In
	1—Enable Carry In
O:	0—Disable Carry Out
	1—Enable Carry Out
S2:	Source 2 Register (rS2)

10
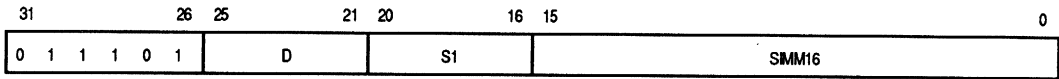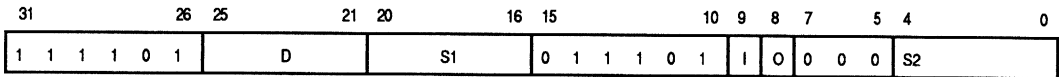
subu   Unsigned Integer Subtract   subu

Operation: Destination ← Source 1 – Source 2

Assembler   **subu**   rD,rS1,rS2
Syntax: **subu.ci**   rD,rS1,rS2
     subu.co   rD,rS1,rS2
     subu.cio   rD,rS1,rS2
     subu   rD,rS1,IMM16

Exceptions: None

Description: The **subu** instruction subtracts the data contained in the rS2 register from the data contained in the rS1 register, or subtracts a zero-extended 16-bit immediate operand from the data contained in the rS1 register. The result of the subtraction is placed in the rD register. The carry bit can optionally be used to perform subtract with borrow operations: a cleared carry bit indicates a borrow and a set carry bit indicates no borrow (i.e., effectively, borrow for subtraction is the opposite of carry for addition).

Subtraction is performed by adding the one's complement of the rS2 operand and either a constant one or the carry bit to the rS1 operand. All 32 bits of the operand participate in the addition.

Instruction Encoding:

Integer Category—Register with 16-Bit Immediate

31					26	25				21	20			16	15								0
0	1	1	0	0	1			D					S1						IMM16				

Integer Category—Triadic Register

10

31					26	25			21	20			16	15						10	9	8	7		5	4			0
1	1	1	1	0	1			D				S1		0	1	1	0	0	1		I	O	0	0	0			S2	
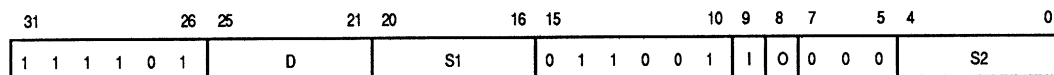
D:	Destination Register (rD)
S1:	Source 1 Register (rS1)
IMM16:	16-Bit Zero-Extended Immediate Operand
I:	0—Disable Carry In
	1—Enable Carry In
O:	0—Disable Carry Out
	1—Enable Carry Out
S2:	Source 2 Register (rS2)

Trap On Bit Clear

Operation: If Bit B5 Clear: Trap VEC9

Assembler   **tb0**   B5,rS1,VEC9
Syntax:

Exceptions: Trap VEC9
Privilege Violation

Description: The **tb0** instruction examines a bit in the S1 register specified by the B5 field. If the bit is clear, exception processing is initiated. The exception vector address is formed by concatenating the upper 20 bits of the vector base register with the contents of the 9-bit VEC9 field followed by a 3-bit field of zeros.

The **tb0** instruction serializes the MC88110 and allows all previous operations to complete (effectively clearing the register scoreboard and data unit pipeline) before the **tb0** executes.

When executed in user mode, a trap to a hardware vector (vectors 0 through 127) causes a privilege violation exception.

Instruction Encoding:

Flow Control Category—9-Bit Vector Table Address

31				26	25		21	20		16	15							9	8					0
1	1	1	1	0 0		B5			S1		1	1	0	1	0	0	0				VEC9			

B5:   Unsigned 5-bit integer denoting a bit number
S1:   Source 1 Register (rS1)
VEC9:   Vector number from the start of the address in the vector base register

10

Trap On Bit Set
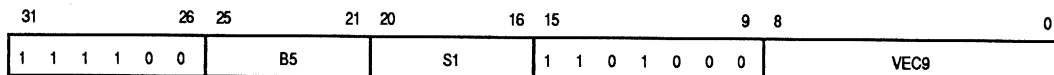
Operation: If Bit B5 Set: Trap VEC9

Assembler   **tb1**   B5,rS1,VEC9
Syntax:

Exceptions: Trap VEC9
                  Privilege Violation

Description: The **tb1** instruction examines a bit in the S1 register specified by the B5 field. If the bit is set, exception processing is initiated. The exception vector address is formed by concatenating the upper 20 bits of the vector base register with the contents of the 9-bit VEC9 field followed by a 3-bit field of zeros.

The **tb1** instruction serializes the MC88110 and allows all previous operations to complete (effectively clearing the register scoreboard and data unit pipeline) before the **tb1** executes.

When in the user mode, a trap to a hardware vector (vectors 0 through 127) causes a privilege violation exception.

Instruction Encoding:

Flow Control Category—9-Bit Vector Table Address

31					26	25			21	20			16	15						9	8				0
1	1	1	1	0	0		B5				S1			1	1	0	1	1	0	0			VEC9		
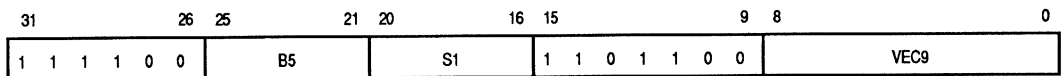
B5:   5-bit unsigned integer denoting a bit number
S1:   Source 1 Register (rS1)
VEC9:   Vector number from the start of the address in the vector base register

tbnd

tbnd

Operation: If unsigned(S1) > unsigned(S2): Trap (bounds check vector)
If unsigned(S1) > unsigned (IMM16): Trap (bounds check vector)

Assembler	**tbnd**	rS1,rS2
Syntax:	**tbnd**	rS1,IMM16

Exceptions: Bounds Check

Description: The **tbnd** instruction uses unsigned arithmetic to compare the data contained in the S1 register either to the data contained in the S2 register or to a zero-extended 16-bit immediate operand. If the S1 operand is larger (out of bounds), a bounds check trap is taken.

Instruction Encoding:
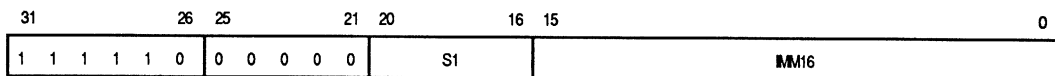
Flow Control Category—Register with 16-Bit Immediate

31	26	25	21	20	16	15	0
1 1 1 1 1 0		0 0 0 0 0		S1		IMM16	

Flow Control Category—Triadic Register

31	26	25	21	20	16	15	5	4	0
1 1 1 1 0 1		0 0 0 0 0		S1		1 1 1 1 1 0 0 0 0 0		S2	

S1:	Source 1 Register (rS1)
IMM16:	16-Bit Zero-Extended Immediate Operand
S2:	Source 2 Register (rS2)

10

tcnd   Conditional Trap
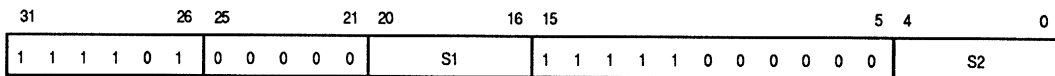
Operation:	If Condition True: Trap

Assembler	tcnd	eq0,rS1,D16
Syntax:	tcnd	ne0,rS1,D16
	tcnd	gt0,rS1,D16
	tcnd	lt0,rS1,D16
	tcnd	ge0,rS1,D16
	tcnd	le0,rS1,D16
	tcnd	M5,rS1,D16

Exceptions:	Trap VEC9
	Privilege Violation

Description: The **tcnd** instruction provides conditional trapping in one instruction without requiring an explicit compare instruction. The **tcnd** instruction examines the data contained in the S1 register and initiates exception processing if the value in the register meets the specified condition (**eq0** for equals zero, etc.). The exception vector address is formed by concatenating the upper 20 bits of the vector base register with the 9-bit VEC9 field followed by a 3-bit field of zeros. The **.n** (delayed trap) option causes the instruction following the **tcnd.n** instruction to be executed before the branch target instruction.

NOTE

In user mode, a trap to a hardware vector (vectors 0–127) causes a privilege violation exception.

The Motorola MC88110 assembler provides mnemonics for commonly used comparison conditions. The following chart lists these mnemonics and their corresponding bit values for the M5 field. The M5 field may also be indicated explicitly by a literal value.

	Bit:	25	24	23	22	21
eq0 (equals zero)		0	0	0	1	0
ne0 (not equal to zero)		0	1	1	0	1
gt0 (greater than zero)		0	0	0	0	1
lt0 (less than zero)		0	1	1	0	0
ge0 (greater than/equals zero)		0	0	0	1	1
le0 (less than/equals zero)		0	1	1	1	0

The **tcnd** instruction serializes the MC88110 and allows all previous operations to complete (effectively clearing the register scoreboard and data unit pipeline) before the **tcnd** executes.

10

Instruction Encoding:

Flow Control Category—9-Bit Vector Table Address

31				26	25		21	20		16	15						9	8		0
1	1	1	1	0 0		M5			S1		1	1	1	0	1	0 0			VEC9	
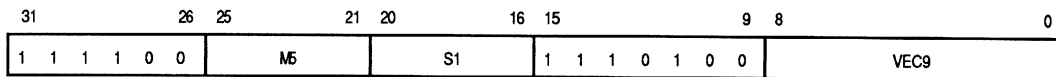
M5:   5-Bit Condition Match Field
              bit 25: reserved, unused by the condition selection logic
              bit 24: maximum negative number   [Sign and Zero]
              bit 23: less than zero   [Sign and (not Zero)]
              bit 22 equal to zero   [(not Sign) and Zero]
              bit 21: greater than zero   [(not Sign) and (not Zero)]
S1:   Source 1 Register (rS1)
VEC9:   Vector number from the start of the address in the vector base register

10

Operation: Destination ← Truncate (Source 2)

Assembler	trnc.ss	rD,rS2	trnc.ss	rD,xS2
Syntax:	trnc.sd	rD,rS2	trnc.sd	rD,xS2
			trnc.sx	rD,xS2

Exceptions: Floating-Point Reserved Operand
Floating-Point Integer Conversion Overflow
Floating-Point Inexact (if not masked)
Floating-Point Unimplemented

Description: The **trnc** instruction converts the single-, double-, or double-extended-precision number contained in the S2 register to a signed 32-bit integer using the IEEE 754 round-toward-zero rounding method. The result is placed in the D register. If the result exceeds 32-bits, the floating-point integer conversion overflow exception is taken. If reserved operands are found, a floating-point reserved operand exception is taken. If execution of the **trnc** instruction is attempted while the FPU is disabled, a floating-point unimplemented exception is taken.

See **Section 4 Floating-Point Implementation** for more information on the floating-point implementation.

Instruction Encoding:

Floating-Point Category—Triadic Register

31					26	25			21	20					16	15	14						9	8	7	6	5	4			0
1	0	0	0	0	1		D			0	0	0	0	0	R	1	0	1	1	0	0		T2		0	0			S2		
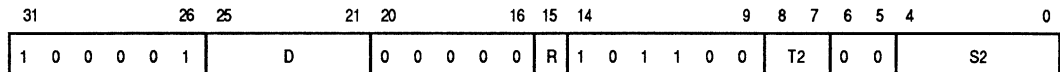
D: Destination Register (rD)
R: 0—Source Operands in GRF
1—Source Operands in XRF
T2: Source 2 Operand Size
00—Single-Precision
01—Double-Precision
10—Double-Extended-Precision
11—Unused
S2: Source 2 Register (rS2 or xS2)

10

Operation: (temp)← Source 1
Destination Register ← Control Register
Control Register ← (temp)

Assembler   **xcr**   rD,rS1,crS/D
Syntax:

Exceptions: Privilege Violation
Unimplemented Opcode

Description: The **xcr** instruction copies the data contained in the S1 register to the control register specified by the CRS/D field, and the contents of the specified control register are loaded into the D register. The general control registers can only be accessed in supervisor mode; a privilege violation occurs if they are accessed in user mode. If the CRS/D field specifies an reserved control register, then a unimplemented opcode exception occurs.

Instruction Encoding:

Load/Store/Exchange Category—Control Register

31	26	25	21	20	16	15	11	10	5	4	0
1 0 0 0 0 0		D		S1		1 1 0 0 0		CRS/D		S2	
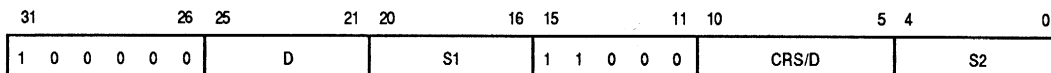
D: Destination Register (rD)
S1:   Source 1 Register
CRS/D: Control Register Source and/or Destination
S2:   Source 2 Register
            Note: The S1 and S2 fields must contain the same register number.

10

xmem   Exchange Register With Memory   xmem

Operation: (temp) ← Source Register (specified as **rD**)
Source Register ← Memory Location
Memory Location ← (temp)

	UNSCALED		SCALED	
Assembler	**xmem.bu**	rD,rS1,rS2	**xmem.bu**	rD,rS1[rS2]
Syntax:	**xmem**	rD,rS1,rS2	**xmem**	rD,rS1[rS2]
	xmem.bu.usr	rD,rS1,rS2	**xmem.bu.usr**	rD,rS1[rS2]
	xmem.usr	rD,rS1,rS2	**xmem.usr**	rD,rS1[rS2]

Exceptions: Data Access Exception
Misaligned Access Exception (if not masked)
Privilege Violation (**.usr** option only)

Description: The **xmem** instruction exchanges the contents of the D register with a memory location. The memory address base is contained in the S1 register. The memory location is calculated by adding the 32-bit word index contained in the S2 register to the base address. The index in the S2 register can be scaled or unscaled. Scaled index mode is indicated by enclosing the S2 register within square brackets.

The **xmem** instruction with no options specifies word (32-bit) operation. The **.bu** option specifies unsigned byte (8-bit) operation. For the scaled index modes, the scale factor is determined by the size option of the instruction. Operations that are byte and word in size define scale factors of 1 and 4 respectively.

Execution of the **xmem** instruction serializes the MC88110 and allows all previous operations to complete (effectively clearing the register scoreboard and data unit pipeline) before the **xmem** executes.

The current memory space is defined by the value of the MODE bit. When MODE is set, the memory access is normally to supervisor memory space; when MODE is clear, the memory access is to user memory space. The MODE bit is located in bit 31 of the PSR and the value of the MODE bit is reflected on the DS/Ū external bus signal. The **.usr** option specifies that the memory access must be to the user address space regardless of the mode bit in the PSR. The **.usr** option is only available in supervisor mode.

In most cases, the **xmem** accesses must be atomic—i.e., the load and the store must not be interrupted. Therefore, the L̄K̄ (bus lock) signal on the external bus is asserted to indicate that the bus arbitration circuitry should not allow another bus master to gain control of the bus between cycles. The only interruption which will occur in this case is a data access exception caused by the store operation after the load has already been performed. If a data access exception occurs, the **xmem** instruction must be re-executed after the software handles the exception to ensure operand consistency.

10

The **xmem** data accesses are normally implemented as a locked sequence of a load followed by a store. If the **xmem** data access is referencing a remote location, it may be desirable to break up the load and store accesses. This is possible if the XMEM bit in the data MMU/cache control register (DCTL) is set, causing the access to be implemented as a store followed by a load. In this reverse case, the memory system must latch the data in the memory location before storing the data from the processor. Then, on the subsequent load, the latched data is loaded into the D register.

Instruction Encoding:

Load/Store/Exchange Category—Register Indirect with Scaled or Unscaled Index

31		26	25		21	20		16	15				11	10	9	8	7		5	4		0
1	1 1 1 0 1			D			S1		0	0 0 0 0	0	W	S	U	0	0 0		S2			

TY:	00—Byte	
	01—Word	
D:	Destination Register (rD)	
S1:	Source 1 Register (rS1)	
S2:	Source 2 Register (rS2)	
W:	0—Byte	
	1—Word	
S:	0—Unscaled	
	1—Scaled	
U:	0—access per user/supervisor bit in PSR (normal mode)	
	1—access user space regardless of PSR	

10

Operation: Destination ← Source 1 ⊕ Source 2

Assembler   **xor**   rD,rS1,rS2
Syntax: **xor.c**   rD,rS1,rS2
                  xor   rD,rS1,IMM16
                  xor.u   rD,rS1,IMM16

Exceptions: None

Description: For triadic register addressing, the **xor** instruction logically XORs the contents of the S1 register with the contents of the S2 register. The result is stored in the D register. If the **.c** (complement) option is specified, the S2 operand is complemented before being XORed.

For register with immediate addressing, the **xor** instruction logically XORs the contents of the S1 register with the unsigned 16-bit immediate operand. The result is placed in the lower 16 bits of the D register, and the upper 16 bits of S1 are copied unchanged to the D register. f the **.u** (upper word) option is specified, the upper 16 bits of the S1 operand are XORed with the unsigned 16-bit immediate operand, and the result is placed in the upper 16 bits of the D register. In this case, the lower 16 bits of S1 are copied unchanged to the D register.

Instruction Encoding:

Logical Category—Register with 16-Bit Immediate

31				27	26	25			21	20			16	15			0
0	1	0	1	0	U		D			S1					IMM16		

Logical Category—Triadic Register

31					26	25			21	20			16	15					11	10	9				5	4			0
1	1	1	1	0	1		D				S1			0	1	0	1	0	C	0	0	0	0	0			S2		
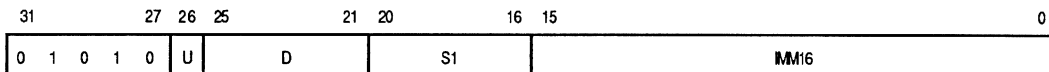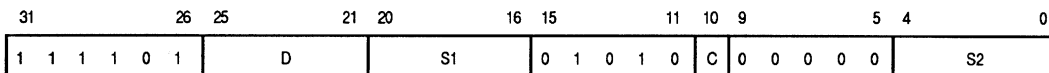
U: 0—XOR IMM16 with Bits 15–0 of S1
          1—XOR IMM16 with Bits 31–16 of S1
D: Destination Register (rD)
S1:   Source 1 Register (rS1)
IMM16:   16-Bit Unsigned Immediate Operand
C: 0—Second operand not complemented before the operation
          1—Second operand complemented before the operation
S2:   Source 2 Register (rS2)

10.2 OPCODE SUMMARY

The following tables present two maps of the MC88110 instruction encodings. The tables are organized by instruction category and provide definitions for all of the instruction fields. See **10.2.7 Instruction Encodings in Numeric Order** for a list of instructions in ascending order by opcode.

NOTE

Attempting to execute any instruction with an unimplemented opcode and/or opcode field causes an exception to occur. Unimplemented opcodes in the base processor cause an unimplemented opcode exception, and unimplemented SFU opcodes cause the respective SFU exception.

10.2.1 Logical Instructions

Table 10-1 lists the Logical:opcode map for the logical instructions category.

Table 10-1. Logical Instructions

Mnemonic	Encoding
	31 27 26 25 21 20 16 15 0
and	`0 1 0 0 0` U D S1 IMM16
mask	`0 1 0 0 1` U D S1 IMM16
xor	`0 1 0 1 0` U D S1 IMM16
or	`0 1 0 1 1` U D S1 IMM16
	31 26 25 21 20 16 15 11 10 9 5 4 0
and	`1 1 1 1 0 1` D S1 `0 1 0 0 0` C `0 0 0 0 0` S2
xor	`1 1 1 1 0 1` D S1 `0 1 0 1 0` C `0 0 0 0 0` S2
or	`1 1 1 1 0 1` D S1 `0 1 0 1 1` C `0 0 0 0 0` S2

U:	0—Apply IMM16 to Bits 15-0 of S1
	1—Apply IMM16 to Bits 31-16 of S1
D:	Destination Register
S1:	Source 1 Register
IMM16:	16-Bit Unsigned Immediate Operand
C:	0—Second operand not complemented before the operation
	1—Second operand complemented before the operation
S2:	Source 2 Register

10

10.2.2 Integer Arithmetic Instructions

Table 10-2 lists the opcode map for the integer arithmetic instructions category.

Table 10-2. Integer Arithmetic Instructions

Mnemonic	Encoding			
	31 — 26	25 — 21	20 — 16	15 — 0
addu	0 1 1 0 0 0	D	S1	IMM16
subu	0 1 1 0 0 1	D	S1	IMM16
divu	0 1 1 0 1 0	D	S1	IMM16
mulu	0 1 1 0 1 1	D	S1	IMM16
add	0 1 1 1 0 0	D	S1	SIMM16
sub	0 1 1 1 0 1	D	S1	SIMM16
divs	0 1 1 1 1 0	D	S1	SIMM16
cmp	0 1 1 1 1 1	D	S1	SIMM16

Mnemonic	31 — 26	25 — 21	20 — 16	15 — 10	9	8	7 — 5	4 — 0
addu	1 1 1 1 0 1	D	S1	0 1 1 0 0 0	I	O	0 0 0	S2
subu	1 1 1 1 0 1	D	S1	0 1 1 0 0 1	I	O	0 0 0	S2
divu	1 1 1 1 0 1	D	S1	0 1 1 0 1 0 0	d		0 0 0	S2
mulu	1 1 1 1 0 1	D	S1	0 1 1 0 1 1 0	d		0 0 0	S2
muls	1 1 1 1 0 1	D	S1	0 1 1 0 1 1 1	0	0 0 0	S2
add	1 1 1 1 0 1	D	S1	0 1 1 1 0 0	I	O	0 0 0	S2
sub	1 1 1 1 0 1	D	S1	0 1 1 1 0 1	I	O	0 0 0	S2
divs	1 1 1 1 0 1	D	S1	0 1 1 1 1 0 0 0		0 0 0	S2
cmp	1 1 1 1 0 1	D	S1	0 1 1 1 1 1 0 0		0 0 0	S2

D:	Destination Register
S1:	Source 1 Register
IMM16:	16-Bit Unsigned Immediate Operand
SIMM16:	16-Bit Signed or Unsigned Operand depending on immediate mode
I:	0—Disable Carry In
	1—Add Carry to Result
O:	0—Disable Carry Out
	1—Generate Carry
S2:	Source 2 Register
d:	0—Single-Word
	1—Double-Word

10.2.3 Special Function Unit (SFU) Instructions

The general opcode map for instructions executed by an SFU is shown in the following illustration:

31	29	28	26	25		21	20		16	15		5	4		0
1 0 0		SFU ID			D			S1			SUBOPCODE			S2	
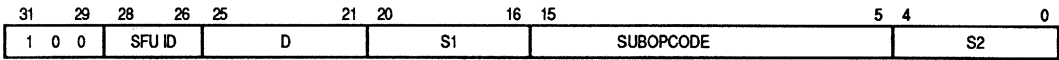
The SFU ID field (bits 28–26) identifies which SFU is specified. An SFU ID of 001 corresponds to the floating-point unit and indicates that the instruction is a floating-point instruction. An SFU ID of 010 corresponds to the graphics unit and indicates that the instruction is a graphics instruction.

An SFU ID of 000 is used to specify SFU control register instructions. The general opcode map for SFU control register instructions is shown in the following illustration:

31	29	28	26	25		21	20		16	15	14	13		11	10		5	4		0
1 0 0		0 0 0			D			S1				DIR		SFU#		SFU CR			S2	
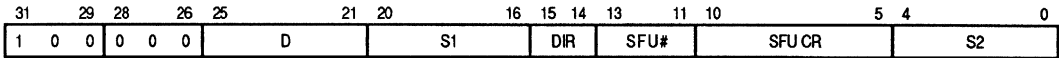
The SFU# field (bits 13–11) identifies which SFU control registers are to be accessed. An SFU# of 000 is used to indicate general control register instructions and an SFU# of 001 is used for floating-point unit control register instructions.

10

10.2.3.1 Floating-Point Instructions.
Table 10-3 lists the opcode map for the floating-point instruction category.

Table 10-3. Floating-Point Instructions

Mnemonic	Encoding (31–26)	D (25–21)	(20–16)	R (15)	(14–11)	(10 9)	(8 7)	(6 5)	(4–0)
fmul	1 0 0 0 0 1	D	S1	R	0 0 0 0	T1	T2	TD	S2
fcvt	1 0 0 0 0 1	D	0 0 0 0 0	R	0 0 0 1 0 0		T2	TD	S2
flt (GRF)	1 0 0 0 0 1	D	0 0 0 0 0	0 0 1 0 0 0 0 0 0				TD	S2
flt (XRF)	1 0 0 0 0 1	D	0 0 0 0 0	0 0 1 0 0 0 1 0 0				TD	S2
fadd	1 0 0 0 0 1	D	S1	R	0 1 0 1	T1	T2	TD	S2
fsub	1 0 0 0 0 1	D	S1	R	0 1 1 0	T1	T2	TD	S2
fcmp	1 0 0 0 0 1	D	S1	R	0 1 1 1	T1	T2	0 0	S2
fcmpu	1 0 0 0 0 1	D	S1	R	0 1 1 1	T1	T2	0 1	S2
mov (GRF)	1 0 0 0 0 1	D	0 0 0 0 0	1 1 0 0 0 0 0			T2*	0 0	S2
mov (XRF)	1 0 0 0 0 1	D	0 0 0 0 0	R	1 0 0 0 0 1		T2†	0 0	S2
int	1 0 0 0 0 1	D	0 0 0 0 0	R	1 0 0 1 0 0		T2	0 0	S2
nint	1 0 0 0 0 1	D	0 0 0 0 0	R	1 0 1 0 0 0		T2	0 0	S2
trnc	1 0 0 0 0 1	D	0 0 0 0 0	R	1 0 1 1 0 0		T2	0 0	S2
fdiv	1 0 0 0 0 1	D	S1	R	1 1 1 0	T1	T2	TD	S2
fsqrt	1 0 0 0 0 1	D	0 0 0 0 0	R	1 1 1 1 0 0		T2	TD	S2

Mnemonic	(31–26)	D (25–21)	(20–16)	(15–11)	(10–5)	(4–0)
fldcr	1 0 0 0 0 0	D	0 0 0 0 0	0 1 0 0 1	FCRS	0 0 0 0 0
fstcr	1 0 0 0 0 0 0 0 0 0 0		S1	1 0 0 0 1	FCRD	S2
fxcr	1 0 0 0 0 0	D	S1	1 1 0 0 1	FCRS/D	S2

D: Destination Register
S1:   Source 1 Register
S2:   Source 2 Register
CRS,CRD: Source Control Register / Destination Control Register
R: 0—Source operands in GRF, 1—Source operands in XRF
$T1_{R=0}$: Source 1 size: 00—single, 01—double, 10—unused, 11—unused
$T1_{R=1}$: Source 1 size: 00—single, 01—double, 10—double-extended, 11—unused
$T2_{R=0}$: Source 2 size: 00—single, 01—double, 10—unused, 11—unused
$T2_{R=1}$: Source 2 size: 00—single, 01—double, 10—double-extended, 11—unused
T2*:   Source 2 size: 00—single, 01—double, 10—unused, 11—unused
$T2†_{R=0}$: Source 2 size: 00—single, 01—double, 10—unused, 11—unused
$T2†_{R=1}$: Source 2 size: 00—unused, 01—unused, 10—double-extended 11—unused
$TD_{R=0}$: Destination size: 00—single, 01—double, 10—unused, 11—unused
$TD_{R=1}$: Destination size: 00—single, 01—double, 10—double-extended, 11—unused

10.2.3.2 Graphics Instructions.
Table 10-4 lists the opcode map for the graphics instruction category.

Table 10-4. Graphics Instructions

Mnemonic	31 26	25 21	20 16	15 11	10 9	8 7	6 5	4 0
pmul	1 0 0 0 1 0	D	S1	0 0 0 0 0	0 0	0 0	0 0	S2
padd	1 0 0 0 1 0	D	S1	0 0 1 0 0	0 0	0 0	T	S2
padds	1 0 0 0 1 0	D	S1	0 0 1 0 0	0 0	S	T	S2
psub	1 0 0 0 1 0	D	S1	0 0 1 1 0	0 0	0 0	T	S2
psubs	1 0 0 0 1 0	D	S1	0 0 1 1 0	0 0	S	T	S2
pcmp	1 0 0 0 1 0	D	S1	0 0 1 1 1	0 0	0 0	1 1	S2
ppack	1 0 0 0 1 0	D	S1	0 1 1 0 0	R		T	S2
punpk	1 0 0 0 1 0	D	S1	0 1 1 0 1	0 0	0 0	T	0 0 0 0 0
prot	1 0 0 0 1 0	D	S1	0 1 1 1 0	R		0 0	0 0 0 0 0
prot	1 0 0 0 1 0	D	S1	0 1 1 1 1	0 0	0 0	0 0	S2*

D: Destination Register
S1:   Source 1 Register
S2:   Source 2 Register
T: 00—4-bit (valid only for **punpk**)
         01—8-bit
         10—16-bit
         11—32-bit
R: 0000—Rotate 64-bit register pair left 0 (or 64) bits.
         0001—Rotate left 4 bits.
         0010—Rotate left 8 bits.
         rrrr—Rotate left (rrrr ¥ 4) bits.
         Note: Only rotations of 8,16, and 32 are meaningful for **ppack** and
         only in limited combinations with T.
S2*:   The nibble-wise (4-bit) rotate count is specified in bits <5:2> of rS2.
         Other bits (<31:6>,<1:0>) are ignored but should be set to zero to assure
         future compatibility.
S: 00—**padd**, nonsaturating
         01—unsigned ± unsigned = unsigned saturation
         10—unsigned ± signed = unsigned saturation
         11—signed ± signed = signed saturation

10

10.2.4 Bit-Field Instructions

Table 10-5 lists the opcode map for the bit-field instruction category.

Table 10-5. Bit-Field Instructions

Mnemonic	Encoding

Mnemonic	31	26	25	21	20	16	15	11	10	5	4	0
clr	1 1 1 1 0 0		D		S1		1 0 0 0 0 0		W5		O5	
set	1 1 1 1 0 0		D		S1		1 0 0 0 1 0		W5		O5	
ext	1 1 1 1 0 0		D		S1		1 0 0 1 0 0		W5		O5	
extu	1 1 1 1 0 0		D		S1		1 0 0 1 1 0		W5		O5	
mak	1 1 1 1 0 0		D		S1		1 0 1 0 0 0		W5		O5	
rot	1 1 1 1 0 0		D		S1		1 0 1 0 1 0		0 0 0 0 0		O5	

Mnemonic	31	26	25	21	20	16	15	5	4	0
clr	1 1 1 1 0 1		D		S1		1 0 0 0 0 0 0 0 0 0 0		S2	
set	1 1 1 1 0 1		D		S1		1 0 0 0 1 0 0 0 0 0 0		S2	
ext	1 1 1 1 0 1		D		S1		1 0 0 1 0 0 0 0 0 0 0		S2	
extu	1 1 1 1 0 1		D		S1		1 0 0 1 1 0 0 0 0 0 0		S2	
mak	1 1 1 1 0 1		D		S1		1 0 1 0 0 0 0 0 0 0 0		S2	
rot	1 1 1 1 0 1		D		S1		1 0 1 0 1 0 0 0 0 0 0		S2	

Mnemonic	31	26	25	21	20	16	15	5	4	0
ff1	1 1 1 1 0 1		D		0 0 0 0 0		1 1 1 0 1 0 0 0 0 0 0		S2	
ff0	1 1 1 1 0 1		D		0 0 0 0 0		1 1 1 0 1 1 0 0 0 0 0		S2	

D: Destination Register
S1: Source 1 Register
W5: 5-bit unsigned integer denoting a bit-field width (0 denotes 32 bits)
O5: 5-bit unsigned integer denoting a bit-field offset
S2: Source 2 Register

10

10.2.5 Load/Store/Exchange Instructions

Table 10-6 lists the opcode map for the load/store/exchange instruction category.

Table 10-6. Load/Store/Exchange Instructions

Mnemonic	Encoding

Mnemonic	31 ... 26	25 ... 21	20 ... 16	15 ... 0
ld.d (XRF)	0 0 0 0 0 0	D	S1	SI16
ld (XRF)	0 0 0 0 0 1	D	S1	SI16
ld.u (GRF)	0 0 0 0 1 B	D	S1	SI16
ld (GRF)	0 0 0 1 TY	D	S1	SI16
st (GRF)	0 0 1 0 TY	D	S1	SI16
st.d (XRF)	0 0 1 1 0 0	D	S1	SI16
st (XRF)	0 0 1 1 0 1	D	S1	SI16
st.x (XRF)	0 0 1 1 1 0	D	S1	SI16
ld.x (XRF)	0 0 1 1 1 1	D	S1	SI16

Mnemonic	31 ... 26	25 ... 21	20 ... 16	15 ... 11	10 ... 5	4 ... 0
ldcr	1 0 0 0 0 0	D	0 0 0 0 0	0 1 0 0 0	CRS	0 0 0 0 0
stcr	1 0 0 0 0 0	0 0 0 0 0	S1	1 0 0 0 0	CRD	S2
xcr	1 0 0 0 0 0	D	S1	1 1 0 0 0	CRS/D	S2

Mnemonic	31 ... 26	25 ... 21	20 ... 16	15 ... 11	10	9	8	7 ... 5	4 ... 0
xmem	1 1 1 1 0 1	D	S1	0 0 0 0 0	W	S	U	0 0 0	S2
ld.u	1 1 1 1 0 1	D	S1	0 0 0 0 1	B	S	U	0 0 0	S2
ld	1 1 1 1 0 R	D	S1	0 0 0 1 TY		S	U	0 0 0	S2
st	1 1 1 1 0 R	D	S1	0 0 1 0 TY		S	U	T 0 0	S2
lda[]	1 1 1 1 0 1	D	S1	0 0 1 1 TY	1	0 0 0 0			S2

TY:	00—Double-Word
	01—Single-Word
	10—Half-Word
	11—Byte
D:	Destination Register
S1:	Source 1 Register
SI16:	16-Bit Signed Immediate Index
S2:	Source 2 Register
.u:	Unsigned
.s:	Single-Word
.d:	Double-Word
.x:	Quad-Word
(XRF):	Destination in extended register file
(GRF):	Destination in general register file
W:	0—Byte, 1—Word
S:	0—Unscaled, 1—Scaled
B:	0—Half-Word, 1—Byte
U:	0—Lower, 1—Upper
T:	0—Normal Store, 1—Store-Through Access

10

10.2.6 Flow Control Instructions

Table 10-7 lists the opcode map for the flow control instruction category.

Table 10-7. Flow Control Instructions

Mnemonic	Encoding

Bit positions: 31   27 26 25 0

Mnemonic	31-27	26 (N)	25-0
br	1 1 0 0 0	N	D26
bsr	1 1 0 0 1	N	D26

Bit positions: 31   27 26 25   21 20   16 15 0

Mnemonic	31-27	26 (N)	25-21	20-16	15-0
bb0	1 1 0 1 0	N	B5	S1	D16
bb1	1 1 0 1 1	N	B5	S1	D16
bcnd	1 1 1 0 1	N	M5	S1	D16

Bit positions: 31   26 25   21 20   16 15   9 8 0

Mnemonic	31-26	25-21	20-16	15-9	8-0
tb0	1 1 1 1 0 0	B5	S1	1 1 0 1 0 0 0	VEC9
tb1	1 1 1 1 0 0	B5	S1	1 1 0 1 1 0 0	VEC9
tcnd	1 1 1 1 0 0	M5	S1	1 1 1 0 1 0 0	VEC9

Bit positions: 31   26 25   16 15   11 10 9   5 4 0

Mnemonic	31-26	25-16	15-11	10 (N)	9-5	4-0
jmp	1 1 1 1 0 1	0 0 0 0 0 0 0 0 0 0	1 1 0 0 0	N	0 0 0 0 0	S2
jsr	1 1 1 1 0 1	0 0 0 0 0 0 0 0 0 0	1 1 0 0 1	N	0 0 0 0 0	S2

Bit positions: 31   26 25   16 15   5 4 0

Mnemonic	31-26	25-16	15-5	4-0
rte	1 1 1 1 0 1	0 0 0 0 0 0 0 0 0 0	1 1 1 1 1 1 0 0 0 0 0	0 0 0 0 0
illop	1 1 1 1 0 1	0 0 0 0 0 0 0 0 0 0	1 1 1 1 1 1 0 0 0 0 0	0 0 0 IL

Bit positions: 31   26 25   21 20   16 15 0

Mnemonic	31-26	25-21	20-16	15-0
tbnd	1 1 1 1 1 0	0 0 0 0 0	S1	IMM16
tbnd	1 1 1 1 0 1	0 0 0 0 0	S1	1 1 1 1 1 0 0 0 0 0 0 S2

N: 1—Execute Next Instr. Unconditionally
D26:   26-Bit Sign Extended Displacement
D16:   16-Bit Sign Extended Displacement
B5:   Bit Number
M5:   Condition Match Field
VEC9:   Vector Number
S1:   Source 1 Register
IMM16: 16-Bit Unsigned Immediate Operand
S2:   Source 2 Register
IL: 01—Illegal Opcode 1
        10—Illegal Opcode 2
        11—Illegal Opcode 3

10.2.7 Instruction Encoding in Numeric Order

Table 10-8 lists the opcode map for the MC88110 instruction set in ascending order.

Table 10-8. Instruction Numeric Listing

Mnemonic	Encoding			
	31 · · · · 26	25 · · 21	20 · · 16	15 · · · · · · · · 0
ld.d (XRF)	0 0 0 0 0 0	D	S1	SI16
ld (XRF)	0 0 0 0 0 1	D	S1	SI16
ld.u (GRF)	0 0 0 0 1 B	D	S1	SI16
ld (GRF)	0 0 0 1 TY	D	S1	SI16
st (GRF)	0 0 1 0 TY	D	S1	SI16
st.d (XRF)	0 0 1 1 0 0	D	S1	SI16
st (XRF)	0 0 1 1 0 1	D	S1	SI16
st.x (XRF)	0 0 1 1 1 0	D	S1	SI16
ld.x (XRF)	0 0 1 1 1 1	D	S1	SI16
and	0 1 0 0 0 U	D	S1	IMM16
mask	0 1 0 0 1 U	D	S1	IMM16
xor	0 1 0 1 0 U	D	S1	IMM16
or	0 1 0 1 1 U	D	S1	IMM16
addu	0 1 1 0 0 0	D	S1	IMM16
subu	0 1 1 0 0 1	D	S1	IMM16
divu	0 1 1 0 1 0	D	S1	IMM16
mulu	0 1 1 0 1 1	D	S1	IMM16
add	0 1 1 1 0 0	D	S1	SIMM16
sub	0 1 1 1 0 1	D	S1	SIMM16
divs	0 1 1 1 1 0	D	S1	SIMM16
cmp	0 1 1 1 1 1	D	S1	SIMM16

Mnemonic	31 · · · 26	25 · · 21	20 · · 16	15 · · 11	10 · · · 5	4 · · 0
ldcr	1 0 0 0 0 0	D	0 0 0 0 0	0 1 0 0 0	CRS	0 0 0 0 0
stcr	1 0 0 0 0 0	0 0 0 0 0	S1	1 0 0 0 0	CRD	S2
xcr	1 0 0 0 0 0	D	S1	1 1 0 0 0	CRS/D	S2
fldcr	1 0 0 0 0 0	D	0 0 0 0 0	0 1 0 0 1	FCRS	0 0 0 0 0
fstcr	1 0 0 0 0 0	0 0 0 0 0	S1	1 0 0 0 1	FCRD	S2
fxcr	1 0 0 0 0 0	D	S1	1 1 0 0 1	FCRS/D	S2

10

Table 10-8. Instruction Numeric Listing (Continued)

Encoding

Mnemonic	31 26	25 (D) 21	20 16	15	14 11	10 9	8 7	6 5	4 0
fmul	1 0 0 0 0 1	D	S1	R	0 0 0 0	T1	T2	TD	S2
fcvt	1 0 0 0 0 1	D	0 0 0 0 0	R	0 0 0 1	0 0	T2	TD	S2
flt (GRF)	1 0 0 0 0 1	D	0 0 0 0 0	0	0 1 0 0	0 0	0 0	TD	S2
flt (XRF)	1 0 0 0 0 1	D	0 0 0 0 0	0	0 1 0 0	0 1	0 0	TD	S2
fadd	1 0 0 0 0 1	D	S1	R	0 1 0 1	T1	T2	TD	S2
fsub	1 0 0 0 0 1	D	S1	R	0 1 1 0	T1	T2	TD	S2
fcmp	1 0 0 0 0 1	D	S1	R	0 1 1 1	T1	T2	0 0	S2
fcmpu	1 0 0 0 0 1	D	S1	R	0 1 1 1	T1	T2	0 1	S2
mov (GRF)	1 0 0 0 0 1	D	0 0 0 0 0	1	1 0 0 0	0 0	T2*	0 0	S2
mov (XRF)	1 0 0 0 0 1	D	0 0 0 0 0	R	1 0 0 0	0 1	T2*	0 0	S2
int	1 0 0 0 0 1	D	0 0 0 0 0	R	1 0 0 1	0 0	T2	0 0	S2
nint	1 0 0 0 0 1	D	0 0 0 0 0	R	1 0 1 0	0 0	T2	0 0	S2
trnc	1 0 0 0 0 1	D	0 0 0 0 0	R	1 0 1 1	0 0	T2	0 0	S2
fdiv	1 0 0 0 0 1	D	S1	R	1 1 1 0	T1	T2	TD	S2
fsqrt	1 0 0 0 0 1	D	0 0 0 0 0	R	1 1 1 1	0 0	T2	TD	S2

Mnemonic	31 26	25 (D) 21	20 16	15 11	10 9	8 7	6 5	4 0
pmul	1 0 0 0 1 0	D	S1	0 0 0 0 0	0 0	0 0	0 0	S2
padd	1 0 0 0 1 0	D	S1	0 0 1 0 0	0 0	0 0	T	S2
padds	1 0 0 0 1 0	D	S1	0 0 1 0 0	0 0	S	T	S2
psub	1 0 0 0 1 0	D	S1	0 0 1 1 0	0 0	0 0	T	S2
psubs	1 0 0 0 1 0	D	S1	0 0 1 1 0	0 0	S	T	S2
pcmp	1 0 0 0 1 0	D	S1	0 0 1 1 1	0 0	0 0	1 1	S2
ppack	1 0 0 0 1 0	D	S1	0 1 1 0 0	R		T	S2
punpk	1 0 0 0 1 0	D	S1	0 1 1 0 1	0 0	0 0	T	0 0 0 0 0
prot	1 0 0 0 1 0	D	S1	0 1 1 1 0	R		0 0	0 0 0 0 0
prot	1 0 0 0 1 0	D	S1	0 1 1 1 1	0 0	0 0	0 0	S2*

Mnemonic	31 27	26	25 0
br	1 1 0 0 0	N	D26
bsr	1 1 0 0 1	N	D26

Mnemonic	31 27	26	25 21	20 16	15 0
bb0	1 1 0 1 0	N	B5	S1	D16
bb1	1 1 0 1 1	N	B5	S1	D16
bcnd	1 1 1 0 1	N	M5	S1	D16

10

Table 10-8. Instruction Numeric Listing (Continued)

Mnemonic	Encoding

Mnemonic	31					26	25				21	20				16	15					11	10				5	4				0
clr	1	1	1	1	0	0	D					S1					1	0	0	0	0	0	W5					O5				
set	1	1	1	1	0	0	D					S1					1	0	0	0	1	0	W5					O5				
ext	1	1	1	1	0	0	D					S1					1	0	0	1	0	0	W5					O5				
extu	1	1	1	1	0	0	D					S1					1	0	0	1	1	0	W5					O5				
mak	1	1	1	1	0	0	D					S1					1	0	1	0	0	0	W5					O5				
rot	1	1	1	1	0	0	D					S1					1	0	1	0	1	0	0	0	0	0	0	O5				

Mnemonic	31					26	25				21	20				16	15							9	8							0
tb0	1	1	1	1	0	0	B5					S1					1	1	0	1	0	0	0		VEC9							
tb1	1	1	1	1	0	0	B5					S1					1	1	0	1	1	0	0		VEC9							
tcnd	1	1	1	1	0	0	M5					S1					1	1	1	0	1	0	0		VEC9							

Mnemonic	31					26	25				21	20				16	15					11	10	9	8	7			5	4		0
xmem	1	1	1	1	0	1	D					S1					0	0	0	0	0	W	S	U	0	0	0		S2			
ld.u	1	1	1	1	0	1	D					S1					0	0	0	0	1	B	S	U	0	0	0		S2			
ld	1	1	1	1	0	R	D					S1					0	0	0	1	TY		S	U	0	0	0		S2			
st	1	1	1	1	0	R	D					S1					0	0	1	0	TY		S	U	T	0	0		S2			
lda[]	1	1	1	1	0	1	D					S1					0	0	1	1	TY		1	0	0	0	0		S2			

Mnemonic	31					26	25				21	20				16	15					11	10	9				5	4			0
and	1	1	1	1	0	1	D					S1					0	1	0	0	0	C	0	0	0	0	0		S2			
xor	1	1	1	1	0	1	D					S1					0	1	0	1	0	C	0	0	0	0	0		S2			
or	1	1	1	1	0	1	D					S1					0	1	0	1	1	C	0	0	0	0	0		S2			

Mnemonic	31					26	25				21	20				16	15					10	9	8	7		5	4				0
addu	1	1	1	1	0	1	D					S1					0	1	1	0	0	0	I	O	0	0	0	S2				
subu	1	1	1	1	0	1	D					S1					0	1	1	0	0	1	I	O	0	0	0	S2				
divu	1	1	1	1	0	1	D					S1					0	1	1	0	1	0	0	d	0	0	0	S2				
mulu	1	1	1	1	0	1	D					S1					0	1	1	0	1	1	0	d	0	0	0	S2				
muls	1	1	1	1	0	1	D					S1					0	1	1	0	1	1	1	0	0	0	0	S2				
add	1	1	1	1	0	1	D					S1					0	1	1	1	0	0	I	O	0	0	0	S2				
sub	1	1	1	1	0	1	D					S1					0	1	1	1	0	1	I	O	0	0	0	S2				
divs	1	1	1	1	0	1	D					S1					0	1	1	1	1	0	0	0	0	0	0	S2				
cmp	1	1	1	1	0	1	D					S1					0	1	1	1	1	1	0	0	0	0	0	S2				

10

Table 10-8. Instruction Numeric Listing (Concluded)

Mnemonic	Encoding

First format (bits 31–26, 25, 21, 20, 16, 15, 5, 4, 0):

Mnemonic	31–26	25 D 21	20 S1 16	15 5	4 S2 0
clr	1 1 1 1 0 1	D	S1	1 0 0 0 0 0 0 0 0 0 0	S2
set	1 1 1 1 0 1	D	S1	1 0 0 0 1 0 0 0 0 0 0	S2
ext	1 1 1 1 0 1	D	S1	1 0 0 1 0 0 0 0 0 0 0	S2
extu	1 1 1 1 0 1	D	S1	1 0 0 1 1 0 0 0 0 0 0	S2
mak	1 1 1 1 0 1	D	S1	1 0 1 0 0 0 0 0 0 0 0	S2
rot	1 1 1 1 0 1	D	S1	1 0 1 0 1 0 0 0 0 0 0	S2

Second format (bits 31–26, 25–16, 15–11, 10, 9, 5, 4, 0):

Mnemonic	31–26	25 16	15 ... 11	10	9 ... 5	4 S2 0
jmp	1 1 1 1 0 1	0 0 0 0 0 0 0 0 0 0	1 1 0 0 0	N	0 0 0 0 0	S2
jsr	1 1 1 1 0 1	0 0 0 0 0 0 0 0 0 0	1 1 0 0 1	N	0 0 0 0 0	S2

Third format (bits 31–26, 25–21, 20–16, 15, 5, 4, 0):

Mnemonic	31–26	25 21	20 16	15 5	4 S2 0
ff1	1 1 1 1 0 1	D	0 0 0 0 0	1 1 1 0 1 0 0 0 0 0 0	S2
ff0	1 1 1 1 0 1	D	0 0 0 0 0	1 1 1 0 1 1 0 0 0 0 0	S2
tbnd	1 1 1 1 0 1	0 0 0 0 0	S1	1 1 1 1 1 0 0 0 0 0 0	S2
rte	1 1 1 1 0 1	0 0 0 0 0 0 0 0 0 0	1 1 1 1 1 1 0 0 0 0 0	0 0 0 0 0
illop	1 1 1 1 0 1	0 0 0 0 0 0 0 0 0 0	1 1 1 1 1 1 0 0 0 0 0 0 0 0	IL
tbnd	1 1 1 1 1 0	0 0 0 0 0	S1	IMM16	

SECTION 11
SYSTEM HARDWARE DESIGN

This section provides a functional description of the bus, the signals that control the bus, and the bus cycles provided for data transfer operations. Descriptions of the data cache operation, bus arbitration, termination, snoop timing, table search timing, reset operation, and test access port are also included.

NOTE

The terms assert and negate are used extensively in this manual to avoid confusion between active-high and active-low signals. Assert or assertion indicates that a signal is active or true, regardless of whether the signal is active high or active low. Negate or negation indicates that the signal is inactive or false.

The timing for the external signals in this section is only accurate to within a half-clock cycle and is included for reference only. The input and output signals are synchronous in that all setup and hold times are specified in reference to the clock signal. MC88110 outputs are driven from a clock edge, and a maximum delay is specified. In addition, minimum hold times are specified in relation to the clock. The minimum setup and hold times must be met in order to guarantee proper device operation. For detailed timing information, refer to the *MC88110 Preliminary Bus Timing Specification*.

11.1 SYSTEM HARDWARE DESIGN OVERVIEW

The instruction unit attempts to fetch two instructions each clock cycle from the instruction cache. If there is an instruction cache miss or if the instruction cache is disabled, the instruction cache requests that the bus interface unit (BIU) run an external bus transaction to fetch the needed instructions. The data cache and data unit may request that the BIU run an external bus transaction as a result of a load, store, or exchange instruction, or for cache coherency reasons, as discussed in **11.3 Data Cache Operation**.

If the instruction and data caches both require that a bus transaction occur at the same time, the BIU gives priority to the instruction cache request unless the data cache must perform a snoop copyback or an **xmem** transaction, or the data cache requests the bus after being retried and forced off the bus.

11

The MC88110 bus interface includes many features to maximize the rate of data transfers between the processor and other devices in the system. All data transfers are synchronous and occur in either single-beat transactions or burst transactions. Burst line fills are performed critical word first, and data streaming is used to provide the data to the central processing unit (CPU) as it is received from the bus.

The MC88110 bus supports multiple processors with a built-in cache coherency mechanism called bus snooping. The MC88110 also supports split bus transactions, in which different devices can control the address bus and data bus at one time. This potentially increases system performance by allowing multiple bus transactions to be in progress simultaneously. The bus also supports pipelining, which allows the address phase of a transaction to overlap the data phase of other transactions. The complexity of the pipeline levels is dependent on external circuitry.

The following paragraphs provide a general discussion of the caches and external bus operations. Sections **11.3 Data Cache Operation** through **11.10 IEEE 1149.1 Test Access Port** provide more detailed descriptions of the specific features of the data cache operation and external bus interface.

11.1.1 Cache Operation Overview

The MC88110 contains a complete mechanism for maintaining maximum data throughput and maintaining coherency between the on-chip data caches in a multiple processor system. The data cache supports both write-through and write-back memory update policies which are selectable on a page-by-page or block-by-block basis. All bus operations that load data into the cache from memory are performed on a line basis (i.e., an entire line is filled). Bus transactions to load data or instructions into the cache always begin with the address of the missed operand or instruction, regardless of the location within a cache line. The missed operand instruction is transferred to the data instruction unit as soon as it is received from the bus so that instruction execution can be resumed as quickly as possible.

The data cache provides a decoupling feature to improve cache performance. When the decoupling feature is enabled, the data unit can continue making cache accesses while the data cache is waiting to receive data from the bus. These cache accesses are called decoupled cache accesses. If a decoupled cache access hits in the cache and does not require an external bus transaction, the access is allowed to complete. If a decoupled cache access requires an external bus transaction, no further decoupled accesses are allowed, and the cache access which requires an external bus transaction is restarted when the cache is available.

Data cache coherency is automatically maintained by hardware bus snooping. There are duplicate address tags and dual-ported state bits associated with each line in the cache to prevent snooping traffic on the bus from interfering with processor operation and degrading performance.

11

The instruction cache is physically addressed and it is never explicitly written to by the program. No hardware support is provided to maintain coherency between multiple instruction caches or between the instruction cache and main memory. In any situation which could cause the instruction cache to have stale data, software must force coherency by invalidating any cache lines which may be stale.

11.3 Data Cache Operation includes an overview of the data cache and a detailed description of the data cache snooping protocol, while **11.7 Data Cache Coherency Timing Considerations** describes the timing for the external snoop transactions. Refer to **Section 6 Instruction and Data Caches** for a complete description of the organization of the instruction and data caches, actions and timings for hits and misses, and cache control.

11.1.2 Bus Arbitration Overview

Although one or more of the devices on the MC88110 bus can have the capability of driving the address and data buses, there can be only one device controlling each bus at any one time. This device is referred to as the bus master. Bus arbitration is the protocol by which a device becomes the bus master. The MC88110 implements an arbitration protocol in which an external arbiter controls bus arbitration, and the processor requests mastership of the bus from the arbiter in order to perform an external access.

The MC88110 bus has separate address and data buses that can be split from each other to enable pipelined bus transactions. Therefore, the MC88110 must arbitrate for mastership of both the address and data bus separately. If the MC88110 is the only possible bus master on the buses, then both buses can be continuously granted to the processor by external logic and no other arbitration is required. For systems with multiple processors but no split bus transactions, the data bus can be continuously granted to the processors and only address bus arbitration is required. To avoid the latency overhead of arbitration, it may be desirable to park the MC88110 on the system address bus. The MC88110 is parked when bus grant is asserted and the processor is not performing a bus transaction.

11

11.1.3 Data Transfer Overview

There are two types of bus transactions that can be used to transfer data on the external bus: single-beat transactions and burst transactions. In general, burst transactions are initiated because of cache misses or snoop copybacks (with the cache enabled), and single-beat transactions are initiated because of disabled caches, cache inhibited accesses, write-through accesses, or similar events.

During single-beat transactions, a byte, half-word, word, or double word is transferred between the processor and an external device in a single bus transfer. The seven types of single-beat transactions are described in Table 11-1. The details of single-beat transactions are described in **11.5.3 Single-Beat Transactions**.

Table 11-1. Single-Beat Transaction Overview

Transaction	Description
Single-Beat Read	During single-beat read transactions, the MC88110 reads a byte, half-word, word, or double word from an external device.
Single-Beat Write	During single-beat write transactions, the MC88110 writes a byte, half-word, word, or double word to an external device.
Invalidate	Invalidate transactions are single-beat transactions used by the MC88110 to maintain cache coherency among multiple MC88110 processors. Invalidate transactions broadcast to snooping devices that a shared line in the cache will be modified; thus, snooping processors must invalidate their cached versions of the memory. There is no data transferred during the invalidate cycle, so the $\overline{MC}$ signal is asserted.
xmem;	The **xmem** instruction is a multiprocessor synchronization instruction that uses an indivisible single-beat read/write transaction to exchange the contents of a general register with that of an addressed memory location. The bus lock signal ($\overline{LK}$) is asserted for both the read and write portions of the **xmem** transaction.
Table Search	A table search operation is a series of single-beat transactions performed by the MC88110 when a logical address misses in the block address translation cache (BATC) and page address translation cache (PATC) with address translation enabled.
Store-Through	The store-through option is a feature that unconditionally causes the store instructions to write-through the on-chip data cache directly to memory. The $\overline{WT}$ signal is always asserted for store-through accesses.
Allocate Load	The allocate load option is a user-mode cache control feature that allows the user to allocate a line in the data cache for a series of subsequent store operations while avoiding the normal line fill from memory. In an allocate load transaction, the $\overline{INV}$ signal is asserted and the $\overline{MC}$ signal is negated.

11

During burst transactions, eight words are transferred between the processor and an external device in 4 double-word transfers. The seven types of burst transactions are described in Table 11-2. The details of burst transactions are described in **11.5.4 Burst Transactions**.

Table 11-2. Burst Transaction Overview

Transaction	Description
Burst Read Transactions	
Cache Read Miss Line Fill	A processor read access that misses in the cache causes a bus transaction to occur in which an entire line of data is read from external memory and written to the cache. This operation is called a cache line fill operation. A cache miss occurs when caching is enabled and the instruction/data required by the processor is not resident in the appropriate cache.
Data Cache Read-with Intent-to-Modify	A read-with-intent-to-modify transaction is caused by a write access that misses in the data cache in write-back mode. A read-with-intent-to-modify transaction operates like a burst read transaction for a cache line fill but has the side effect of broadcasting to other processors on the bus that the cache line being read will be modified; thus, the other processors should invalidate any local copy of the cache line (and perform a snoop copyback if the local copy is modified).
Touch Load	The touch load option is a user-mode cache control feature that allows data to be loaded into the data cache under user program control. By forcing certain data be read into the cache ahead of its actual use, the latency of the memory system can be overlapped with useful work, and stalls due to long latency cache misses can be minimized.
Burst Write Transactions	
Replacement Copyback	When a data cache miss occurs and the corresponding cache set has two valid entries, the cache access algorithm selects one of the two lines in the corresponding cache set for replacement. The MC88110 checks the state of the line to be replaced, and if the line is modified, then the line is copied back to memory. This operation is called a replacement copyback
Snoop Copyback	When a snooping MC88110 has a cache hit during a global transaction, the snooping MC88110 determines if the cache line is modified. If the line is modified, the line must be copied back to memory before the device performing the global access can complete its transaction. This operation is called a snoop copyback.
Flush Copyback	The MC88110 has a supervisor mode cache control feature that causes either all modified lines or any individual modified line in the data cache to be transferred out to memory, and causes the transferred line(s) to be marked as unmodified. Each line transferred to memory by this operation is transferred by way of a burst write transaction called a flush copyback.
Flush Load	The flush load option is a user-mode cache control feature that allows the user to force a modified cache line to be written to memory.

Transactions may be terminated normally, indicating that the transfer was completed successfully, or terminated with an error or a retry indication. Two types of retry terminations are possible: transfer retry and address retry. If the access is terminated with a retry before the needed data is transferred, then the access will be re-initiated from the cache lookup operation (see **Section 6 Instruction and Data Caches**).

11

11.2 SIGNAL DESCRIPTION

The following paragraphs describe the input and output signals of the MC88110 in their functional groups. Figure 11-1 shows the functional organization of the MC88110 signals, and Table 11-3 provides a list of the signals organized by function and gives the mnemonic, count, type, active state, and state out of reset for each signal.

Figure 11-1. MC88110 Pinout

Table 11-3. MC88110 Signal Summary

Function	Mnemonic	Count	Type	Active	Reset
Data Transfer					
Data Bus	D63–D0	64	I/O	High	Three-State
Address Bus	A31–A0	32	I/O	High	Three-State
Byte Parity	BP7–BP0	8	I/O	High	Three-State
Transfer Attributes					
Read/Write	R/W̄	1	I/O	High	Three-State
Lock	L̄K̄	1	Output	Low	Three-State
Cache Inhibit	C̄Ī	1	Output	Low	Three-State
Write-Through	W̄T̄	1	Output	Low	Three-State
User Page Attributes	ŪPA1–ŪPA0	2	Output	Low	Three-State
Transfer Burst	T̄B̄S̄T̄	1	I/O	Low	Three-State
Transfer Size	TSIZ1–TSIZ0	2	Output	High	Three-State
Transfer Code	TC3–TC0	4	Output	High	Three-State
Invalidate	ĪN̄V̄	1	I/O	Low	Three-State
Memory Cycle	M̄C̄	1	Output	Low	Three-State
Global	ḠB̄L̄	1	I/O	Low	Three-State
Cache Line	CLINE	1	Output	High	Three-State
Transfer Control					
Transfer Start	T̄S̄	1	Output	Low	Three-State
Transfer Acknowledge	T̄Ā	1	Input	Low	—
Pretransfer Ack	P̄T̄Ā	1	Input	Low	—
Transfer Error Ack	T̄ĒĀ	1	Input	Low	—
Transfer Retry	T̄R̄T̄R̄Ȳ	1	Input	Low	—
Address Acknowledge	ĀĀC̄K̄	1	Input	Low	—
Snoop Control					
Snoop Request	S̄R̄	2	Input	Low	—
Address Retry	ĀR̄T̄R̄Ȳ	1	Input	Low	—
Snoop Status	S̄S̄T̄ĀT̄1– S̄S̄T̄ĀT̄0	2	Output	Low	Three-State
Shared	S̄H̄D̄	1	Input	Low	—
Arbitration					
Bus Request	B̄R̄	1	Output	Low	Negated
Bus Grant	B̄Ḡ	1	Input	Low	—
Address Bus Busy	ĀB̄B̄	1	I/O	Low	Three-State
Data Bus Grant	D̄B̄Ḡ	1	Input	Low	—
Data Bus Busy	D̄B̄B̄	1	I/O	Low	Three-State

11

Table 11-3. MC88110 Signal Summary (Continued)

Function	Mnemonic	Count	Type	Active	Reset
Processor Status					
Processor Status	PSTAT2– PSTAT0	3	Output	High	Input
Interrupt					
Nonmaskable Interrupt	$\overline{\text{NMI}}$	1	Input	Low	Three-State
Interrupt	$\overline{\text{INT}}$	1	Input	Low	Three-State
Reset	$\overline{\text{RST}}$	1	Input	Low	Three-State
Byte Parity Error	$\overline{\text{BPE}}$	1	Output	Low	Three-State
Clock					
Clock	CLK	1	Input	Rising Clock Edge	—
Test Pins					
Debug	$\overline{\text{DBUG}}$	1	Input	Low	—
Resistor 1	RES1	1	Input	N/A	—
Resistor 2	RES2	1	Output	N/A	—
JTAG Test Reset	$\overline{\text{TRST}}$	1	Input	Low	—
JTAG Test Mode Select	TMS	1	Input	High	—
JTAG Test Clock	TCK	1	Input	Clock Edge	—
JTAG Test Data Input	TDI	1	Input	High	—
JTAG Test Data Output	TDO	1	Output	High	—

11.2.1 Data Transfer Signals

The following paragraphs describe the address, data, and byte parity signals of the MC88110.

11.2.1.1 DATA BUS (D63–D0). D63–D0 are bidirectional signals that comprise the data path for all transactions. The data bus is divided into byte lanes as shown in Table 11-4. These signals are outputs during write transactions, inputs during read transactions, and three-stated when the MC88110 does not have mastership of the data bus (i.e., when the MC88110 is not asserting $\overline{\text{DBB}}$).

11

Table 11-4. Data Bus Byte Lanes

Data Bus Signals	Byte Lane
D63–D56	0
D55–D48	1
D47–D40	2
D39–D32	3
D31–D24	4
D23–D16	5
D15–D8	6
D7–D0	7

11.2.1.2 ADDRESS BUS (A31–A0). A31–A0 comprise the address bus for all external bus transactions. The signals are outputs when the MC88110 has mastership of the address bus (i.e., when $\overline{ABB}$ is asserted), inputs when the MC88110 is snooping (see **11.3.3 Data Cache Coherency**), and three-stated at all other times.

11.2.1.3 BYTE PARITY BUS (BP7–BP0). These signals indicate the parity of the data bus. The MC88110 always uses odd parity, checking parity for read transactions and generating parity for write transactions. Each parity signal corresponds to eight data signals as shown in Table 11-5. During read transactions, only the parity bits corresponding to active byte lanes need to be valid. The byte parity signals are three-stated when the MC88110 does not have mastership of the data bus (i.e., when the MC88110 is not asserting $\overline{DBB}$).

Table 11-5. Data Byte Parity Signals

Byte Parity Signals	Data Bus Signals
BP0	D63–D56
BP1	D55–D48
BP2	D47–D40
BP3	D39–D32
BP4	D31–D24
BP5	D23–D16
BP6	D15–D8
BP7	D7–D0

11.2.2 Transfer Attribute Signals

The following paragraphs describe the transfer attribute signals, including the read/write, lock, cache inhibit, write-through, user page attributes, transfer burst, transfer size, transfer code, invalidate, memory cycle, global, and cache line signals. The timing for each of the transfer attribute signals is the same as the timing for addresses.

11.2.2.1 READ/WRITE (R/$\overline{\text{W}}$). The R/$\overline{\text{W}}$ signal indicates whether the transaction is a read (R/$\overline{\text{W}}$ high) or a write (R/$\overline{\text{W}}$ low) transaction. This signal is an output when the MC88110 is driving an address, an input when the MC88110 is snooping (see **11.3.3 Data Cache Coherency**), and three-stated at all other times.

11.2.2.2 LOCK ($\overline{\text{LK}}$). The MC88110 drives the $\overline{\text{LK}}$ signal to indicate that an access is part of an atomic data access sequence. It is asserted during **xmem** transactions only.

11.2.2.3 CACHE INHIBIT ($\overline{\text{CI}}$). The $\overline{\text{CI}}$ signal indicates that the data will not be written into the MC88110 data cache. For single-beat transactions, **xmem** transactions, and touch and allocate load transactions, this signal reflects the value of the CI bit in the address translation cache (ATC) entry (or area descriptor for identity translations) used to map the current address. For all other transactions, this signal is negated.

11.2.2.4 WRITE-THROUGH ($\overline{\text{WT}}$). The $\overline{\text{WT}}$ signal is asserted if the WT bit is set in the corresponding ATC entry (or area descriptor for identity translations) or if a write transaction is the result of a store-through operation. For all other transactions, this signal is negated.

11.2.2.5 USER PAGE ATTRIBUTES ($\overline{\text{UPA1}}$–$\overline{\text{UPA0}}$). These signals reflect the user attribute bits in the ATC entry used to map the current address. Note that the state of the of user attribute bits in ATC is opposite to that of the signals. (i.e., if U1 in the ATC entry is set, then $\overline{\text{UPA1}}$, which is active-low, is asserted). During table search operations and identity translations, $\overline{\text{UPA1}}$ and $\overline{\text{UPA0}}$ reflect the values in the appropriate area descriptor. During copyback operations these signals are negated.

11.2.2.6 TRANSFER BURST ($\overline{\text{TBST}}$). This signal indicates the type of the transaction. This signal is an output when the MC88110 is driving an address, an input when the MC88110 is snooping (see **11.3.3 Data Cache Coherency**), and three-stated at all other times. When the $\overline{\text{TBST}}$ signal is asserted, the transaction is an eight-word burst. If it is negated, the transaction is a single-beat transaction, and the size of the data to be transferred is encoded in the transfer size signals (TSIZ1–TSIZ0).

11.2.2.7 TRANSFER SIZE (TSIZ1–TSIZ0). The TSIZ1–TSIZ0 signals indicate the size of the requested data transfer as shown in Table 11-6. All transfers are aligned to their respective size boundaries. The TSIZ1–TSIZ0 signals may be used along with A2–A0 to determine which portion of the data bus contains valid data for a write transaction or which portion of the bus should contain valid data for a read transaction.

Note that TSIZ1–TSIZ0 indicate the size of the requested data transfer independent of the value of $\overline{\text{TBST}}$. Therefore, it is possible for the TSIZ signals to indicate a byte, half-word, or word transfer even when the $\overline{\text{TBST}}$ signal is asserted (i.e., when a **ld.w** misses the cache, the TSIZ signals indicate a word during the cache line fill). If the $\overline{\text{TBST}}$ signal is asserted, the memory system must transfer double words regardless of the TSIZ1–TSIZ0 encoding.

Table 11-6. Transfer Size Signal Encodings

TSIZ1–TSIZ0	Transfer Size
0 0	Double Word (64 Bits)
0 1	Word (32 Bits)
1 0	Half-Word (16 Bits)
1 1	Byte (8 Bits)

11.2.2.8 TRANSFER CODE (TC3–TC0). These four signals provide supplemental information about the corresponding address. The transfer code signals are encoded as shown in Table 11-7.

Table 11-7. Transfer Code Signal Encodings

TC3–TC0	Transfer Code
0 0 0 0	Reserved
0 0 0 1	User Data Access
0 0 1 0	User Touch, Flush, or Allocate Access
0 0 1 1	Data MMU Table Search Operation
0 1 0 0	Reserved
0 1 0 1	Supervisor Data Access
0 1 1 0	Supervisor Touch, Flush, or Allocate Access
0 1 1 1	Snoop Copyback
1 0 0 0	Reserved
1 0 0 1	User Instruction Access
1 0 1 0	Reserved
1 0 1 1	Instruction MMU Table Search Operation
1 1 0 0	Reserved
1 1 0 1	Supervisor Instruction Access
1 1 1 0	Reserved
1 1 1 1	Reserved

11.2.2.9 INVALIDATE (INV). When asserted, the $\overline{INV}$ output signal indicates that all other caches in the system should invalidate the cache line on a snoop hit. If the snoop hit is to a modified line, the line should be copied back before being invalidated. This signal is an output when the MC88110 is driving an address, an input when the MC88110 is snooping, and three-stated at all other times.

11.2.2.10 MEMORY CYCLE (MC). When asserted, the $\overline{MC}$ output signal indicates that a data transfer transaction is in progress. When $\overline{MC}$ is negated, the current bus transaction is an invalidate cycle, and no data is transferred. On invalidate cycles, valid data is driven, but the memory system is not required to execute the data write.

11

11.2.2.11 GLOBAL (GBL). Address bus masters assert $\overline{\text{GBL}}$ to indicate that the transaction in progress is marked as global. Normally, $\overline{\text{GBL}}$ reflects the value specified for the memory reference in the corresponding memory management unit (MMU). Special transactions, such as table search transactions and copyback transactions, are considered nonglobal and $\overline{\text{GBL}}$ is negated. When the CPU is not the address bus master, $\overline{\text{GBL}}$ is an input. When $\overline{\text{GBL}}$ and $\overline{\text{SR}}$ are asserted, the MC88110 snoops the current address.

11.2.2.12 CACHE LINE (CLINE). The CLINE signal indicates which line in the cache is involved in the current data transfer (see Table 11-8). It can be used with other signals (e.g., R/$\overline{\text{W}}$, $\overline{\text{INV}}$, $\overline{\text{MC}}$, $\overline{\text{LK}}$, $\overline{\text{TBST}}$, TC3–TC0, A11–A5) to determine the next state of a particular instruction or data cache line.

Table 11-8. Cache Line Signal

CLINE	Cache Line
0	Line 0
1	Line 1

11.2.3 Transfer Control Signals

The following paragraphs describe the transfer control signals, which include the transfer start, the transfer acknowledge, the pretransfer acknowledge, the transfer error acknowledge, the transfer retry, and the address acknowledge signals.

11.2.3.1 TRANSFER START (TS). The MC88110 asserts the $\overline{\text{TS}}$ output signal to indicate that a transaction has begun and the driven address is valid. This signal is asserted for one clock cycle, negated, and then three-stated.

11.2.3.2 TRANSFER ACKNOWLEDGE (TA). During a read transaction, $\overline{\text{TA}}$ should be asserted when new data is valid. During a write transaction, $\overline{\text{TA}}$ should be asserted when the data from the MC88110 has been latched by the memory system.

11.2.3.3 PRETRANSFER ACKNOWLEDGE (PTA). The memory system asserts the $\overline{\text{PTA}}$ input signal to indicate that the initial (or only) $\overline{\text{TA}}$ assertion of the transaction may follow on the next rising clock edge. During the time between when $\overline{\text{TS}}$ is asserted and $\overline{\text{PTA}}$ is asserted, the data unit of the MC88110 can continue to access the data cache (cache hits only) even though a bus transaction is in progress. Since data cannot be transferred until one clock after a qualified bus grant, $\overline{\text{PTA}}$ may be connected to $\overline{\text{DBG}}$. For systems which do not require decoupled cache accesses, this signal may be tied to ground.

11.2.3.4 TRANSFER ERROR ACKNOWLEDGE (TEA). The $\overline{\text{TEA}}$ signal indicates that a bus error has occurred. The assertion of $\overline{\text{TEA}}$ results in the immediate termination of the transfer in progress. The actions of the MC88110 after the transfer is terminated are described in **11.6.4 Transfer Error Termination**.

11

11.2.3.5 TRANSFER RETRY (TRTRY). The $\overline{\text{TRTRY}}$ signal indicates that the current transaction should be terminated and re-initiated. The assertion of $\overline{\text{TRTRY}}$ results in the immediate termination of the transaction. The actions of the MC88110 after the transfer is terminated are described in **11.6.3 Transfer Retry Termination**. If the $\overline{\text{TRTRY}}$ signal is asserted at the same time as the $\overline{\text{TEA}}$ signal, the $\overline{\text{TEA}}$ signal gets priority and an error termination occurs.

11.2.3.6 ADDRESS ACKNOWLEDGE (AACK). When the $\overline{\text{AACK}}$ input is asserted, the MC88110 stops driving an address on the address bus and negates $\overline{\text{ABB}}$. $\overline{\text{AACK}}$ is sampled beginning with the rising clock edge following the assertion of $\overline{\text{TS}}$ and ending with the qualified termination of the transaction.

11.2.4 Snoop Control Signals

The following paragraphs describe the snoop control signals, which include the snoop request, address retry, snoop status, and shared signals.

11.2.4.1 SNOOP REQUEST (SR). The snoop request input signal indicates that there is a valid address on the bus and that the MC88110 should snoop the address if the global ($\overline{\text{GBL}}$) signal is asserted. In many systems with multiple MC88110s, the $\overline{\text{TS}}$ output of the MC88110 initiating the transfer may be used to drive the $\overline{\text{SR}}$ input of other MC88110s on the bus.

11.2.4.2 ADDRESS RETRY (ARTRY). The address retry ($\overline{\text{ARTRY}}$) signal is an input signal that indicates to the current address bus master that it should terminate the transaction and re-initiate the transaction at a later time. An MC88110 that is the current address bus master can detect a qualified $\overline{\text{ARTRY}}$ on the clock edge following the assertion of $\overline{\text{TS}}$. The $\overline{\text{ARTRY}}$ signal is qualified with $\overline{\text{AACK}}$ or with a qualified $\overline{\text{TA}}$.

If the MC88110 has requested the bus and $\overline{\text{ARTRY}}$ is asserted (qualified or unqualified) and $\overline{\text{ABB}}$ was asserted on the previous clock cycle, the MC88110 removes its bus request and ignores $\overline{\text{BG}}$. If the MC88110 has not requested the bus and $\overline{\text{ARTRY}}$ is asserted the MC88110 does not assert $\overline{\text{BR}}$ until $\overline{\text{ARTRY}}$ is negated.

11.2.4.3 SHARED (SHD). The assertion of the $\overline{\text{SHD}}$ signal indicates that the cache line currently being read into the data cache should be marked as shared-unmodified. If $\overline{\text{SHD}}$ is negated, the cache line is marked as exclusive-unmodified. If the $\overline{\text{INV}}$ signal is asserted for the transaction, the line is marked exclusive-unmodified regardless of the state of the $\overline{\text{SHD}}$ signal. The timing of the $\overline{\text{SHD}}$ input is the same as the timing for $\overline{\text{ARTRY}}$.

11.2.4.4 SNOOP STATUS (SSTAT1–SSTAT0). The snoop status signals indicate the status of the transaction by the snooping CPU as shown in Table 11-9. The snoop status ($\overline{\text{SSTAT1}}$–$\overline{\text{SSTAT0}}$) signals are output signals that are asserted by a snooping processor when it detects a snoop hit or collision (see **11.7.8 Split-Bus Snoop Collisions**). $\overline{\text{SSTAT1}}$ is asserted for both snoop hits and collisions, so it can be directly or indirectly tied to $\overline{\text{ARTRY}}$. $\overline{\text{SSTAT0}}$ is asserted for all snoop hits, so it can be directly or indirectly tied to $\overline{\text{SHD}}$.

11

Table 11-9. Snoop Status Signals

SSTAT1	SSTAT0	Status
Three-State	Three-State	No Collision, No Snoop Hit
Three-State	Asserted	Snoop Hit Shared
Asserted	Three-State	Pipeline Collision
Asserted	Asserted	Snoop Hit Modified

11.2.5 Bus Arbitration Signals

The following paragraphs describe the bus arbitration signals, including the bus request, bus grant, address bus busy, data bus grant, and data bus busy signals.

11.2.5.1 BUS REQUEST ($\overline{BR}$). The MC88110 asserts the bus request signal to request bus mastership and continues to assert it until it has received a qualified bus grant (see **11.2.5.2 Bus Grant ($\overline{BG}$)**) and has started a transaction or determines that it does not need the bus. For **xmem** operations, the bus request signal remains asserted until $\overline{TS}$ is asserted for the second transaction.

11.2.5.2 BUS GRANT ($\overline{BG}$). The bus grant signal is used by the external bus arbiter to grant address bus mastership in response to a bus request. The MC88110 only assumes address bus mastership if $\overline{BG}$ is asserted and the bus is not already busy ($\overline{ABB}$ is negated). The external arbiter may park the MC88110 on the bus by keeping $\overline{BG}$ asserted after the bus request has been negated (see **11.4.4 Bus Parking**).

11.2.5.3 ADDRESS BUS BUSY ($\overline{ABB}$). The address bus busy signal is asserted by the current address bus master to indicate that potential bus masters must wait to take mastership of the address bus. Potential address bus masters use this input to qualify $\overline{BG}$. It is an output when the MC88110 is the address bus master and an input at all other times.

The address bus busy signal may be a shared signal among multiple MC88110s or other bus masters. It must be tied to a pull-up resistor so that it remains negated when no devices have control of the address bus.

11.2.5.4 DATA BUS GRANT ($\overline{DBG}$). The data bus grant signal is used by the external bus arbiter to grant data bus mastership in response to a data bus request. The assertion of $\overline{TS}$ serves as the data bus request. The MC88110 only assumes data bus mastership if $\overline{DBG}$ is asserted and the data bus is not already busy ($\overline{DBB}$ is negated). Note that it is not possible to park the data bus.

11.2.5.5 DATA BUS BUSY ($\overline{DBB}$). The data bus busy signal is asserted by the current data bus master to indicate that potential data bus masters must wait to take mastership of the data bus. Potential data bus masters use this input to qualify data bus grant. The data bus busy signal is an input when the MC88110 is waiting to obtain data bus mastership, an output when the MC88110 is data bus master, and three-stated at all other times.

The data bus busy signal may be a shared signal among multiple MC88110s or other bus masters. It must be tied to a pull-up resistor so that it remains negated when no devices have control of the data bus.

11.2.6 Processor Status Signals

The three processor status signals provide limited visibility of the CPU status. These bidirectional signals normally function as outputs; however, they function as inputs during reset. The three-bit value loaded through PSTAT2–PSTAT0 at reset determines the function of the signals during normal operation. The selection can only be made during reset, so the function of the signals is not dynamically programmable during normal operation.

The PSTAT2–PSTAT0 signals are sampled on every clock cycle in which $\overline{RST}$ is asserted. When $\overline{RST}$ is negated, the MC88110 waits a minimum of three clock cycles before driving the PSTAT2–PSTAT0 signals. This gives the off-chip driving logic time to go into a high-impedance state to avoid possible bus contention.

Table 11-10 defines the function of the PSTAT signals for each of the possible combinations at reset. Note that several of the available options are reserved for Motorola internal use only.

11.2.7 Interrupt Signals

The following paragraphs describe the interrupt signals used by the MC88110, including the nonmaskable interrupt, interrupt, reset, and byte parity error signals.

11.2.7.1 NONMASKABLE INTERRUPT ($\overline{NMI}$). The assertion of the $\overline{NMI}$ input indicates that a nonmaskable external interrupt has been requested. When a valid interrupt is detected, the MC88110 will unconditionally trap through the nonmaskable interrupt vector. The interrupt signal is sampled by the CPU at the rising edge of each bus clock. The interrupt signal can be completely asynchronous; however, it will only be detected on a given clock if setup and hold times are satisfied, and it must be asserted for two clock cycles to be recognized.

11.2.7.2 INTERRUPT ($\overline{INT}$). The assertion of the $\overline{INT}$ input indicates that an external interrupt has been requested. When a valid interrupt is detected and the interrupt is enabled by the interrupt disable bit in the processor status register (PSR), the MC88110 will trap through the maskable interrupt vector. The interrupt signal is sampled by the CPU at the rising edge of each bus clock. The interrupt signal can be completely asynchronous; however, it will only be detected on a given clock if setup and hold times are satisfied, and it must be asserted for two clock cycles to be recognized.

11

Table 11-10. PSTAT2–PSTAT0 Functionality

PSTAT2–PSTAT0 at Reset	PSTAT2–PSTAT0 Functionality
0 0 0	PSTAT0: Asserted when an instruction is issued in slot 0 of the issue pair PSTAT1: Asserted when an instruction is issued in slot 1 of the issue pair PSTAT2: Asserted when a change of flow is issued Note that if two instructions are issued, both PSTAT0 and PSTAT1 will be asserted. If only one instruction issues, only PSTAT0 will be asserted. If either instruction is decoded as a flow control instruction, PSTAT2 will be asserted.
0 0 1	PSTAT0: Asserted when a store instruction is at the top of the history buffer PSTAT1: Asserted when a store instruction is completed PSTAT2: Reserved for future use Note that both PSTAT0 and PSTAT1 may be set if more than one store instruction is in the history buffer.
0 1 0	PSTAT0: Asserted when instructions are conditionally executing after a branch PSTAT1: Asserted when speculative instructions are being flushed after a misprediction PSTAT2: Asserted when an exception is recognized and unretired instructions are being flushed PSTAT0 can be asserted for more than one clock cycle. PSTAT1 and PSTAT2 will only be asserted for one clock cycle. PSTAT2 will be asserted when the instruction tagged with the exception reaches the top of the history buffer.
0 1 1	PSTAT0: Bit 2 of the instruction address being fetched from the instruction cache PSTAT1: Bit 3 of the instruction address being fetched from the instruction cache PSTAT2: Bit 4 of the instruction address being fetched from the instruction cache
1 0 0	Reserved for Motorola Internal Use Only
1 0 1	Reserved for Motorola Internal Use Only
1 1 0	PSTAT0: Reserved for Motorola Internal Use Only PSTAT1: Asserted when an interrupt is detected PSTAT2: Asserted when an interrupt is taken PSTAT1 is asserted from the time the asserted interrupt signal is recognized until PSTAT2 asserts or the interrupt input is negated. PSTAT2 is asserted after the machine backs out the history buffer and has calculated the vector address. It is asserted for one clock cycle.
1 1 1	Reserved for Motorola Internal Use Only

11.2.7.3 RESET (RST). The $\overline{\text{RST}}$ signal is used to perform an orderly restart of the processor, bringing it to a known state and beginning program execution at address $0 (the reset vector). When $\overline{\text{RST}}$ is asserted, all current operations are suspended and all control registers are set to their default state. When $\overline{\text{RST}}$ is negated, the reset vector is fetched from memory, and execution begins in supervisor mode with all the caches, MMUs, and breakpoints disabled. $\overline{\text{RST}}$ must be asserted for two clock cycles to be recognized.

11.2.7.4 BYTE PARITY ERROR (BPE). The $\overline{\text{BPE}}$ signal is asserted for one clock cycle following detection of incorrect parity on any data byte read into the MC88110. Note that the MC88110 does not take an exception when it detects incorrect parity.

11

11.2.8 Clock (CLK)

The clock input signal generates the internal timing signals for the processor. The processor internal clock is derived from the leading edge of the CLK signal and is phase locked to minimize the skew between the external and internal signals.

11.2.9 Test Signals

The following paragraphs describe the test signals for the MC88110, including the debug, resistor, and JTAG test port signals. For more information on the JTAG test port, refer to **11.10 IEEE 1149.1 Test Access Port**.

11.2.9.1 DEBUG ($\overline{\text{DBUG}}$). When this signal is asserted, all caches, MMUs, and breakpoints are disabled.

11.2.9.2 RESISTOR (RES2–RES1). These signals provide access to an internal resistor for measuring device junction temperatures.

11.2.9.3 JTAG TEST RESET ($\overline{\text{TRST}}$). Assertion of this signal causes asynchronous initialization of the internal JTAG test access port controller. This signal conforms to the *IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture*.

11.2.9.4 JTAG TEST MODE SELECT (TMS). TMS is decoded by the internal JTAG TAP controller to distinguish the primary operations of the test support circuitry. This signal conforms to the *IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture*.

11.2.9.5 JTAG TEST CLOCK (TCK). This signal clocks the internal boundary scan test support circuitry. This signal conforms to the *IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture*.

11.2.9.6 JTAG TEST DATA INPUT (TDI). The state of this signal is clocked into the selected JTAG test instruction or data register on the rising edge of TCK. This signal conforms to the *IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture*.

11.2.9.7 JTAG TEST DATA OUTPUT (TDO). The contents of the selected internal instruction or data test register are shifted out onto this signal on the falling edge of TCK. This signal conforms to the *IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture*.

11

11.3 DATA CACHE OPERATION

This section provides an overview of the operation of the data cache of the MC88110 and how data cache interactions affect the overall operation of the bus. In particular, those subjects that affect hardware design, such as the two models of cache maintenance (three or four state), the three memory update policies, and bus snooping, are described. Refer to **Section 6 Instruction and Data Caches** for a complete description of the organization of the instruction and data caches, actions and timings for hits and misses, and cache control.

11.3.1 Data Cache States

When a data access occurs in the program flow, the actions taken by the cache depend on whether the access is cacheable. If the access is cacheable, then the actions taken by the cache depend on the state of the cache line.

Each data cache line can be in one of four states at any one time. These states reflect the status of the line with respect to memory and whether or not the processor has exclusive ownership of the cached data. The state of each data cache line is indicated by the three state bits in that line. The first bit indicates whether a line is valid or invalid, the second bit indicates whether the line is shared or exclusive to the processor, and the third bit indicates whether the line is modified or unmodified with respect to memory. The following list depicts the four possible data cache line states:

1. Invalid—The data in this line is no longer the most recent copy of the data and should not be used. A line is marked invalid as a result of four conditions: software invalidates the entire cache or a specific line in the cache, the bus snooping logic marks the line as invalid, a bus error occurs during a cache line read access, or a cache hit occurs for a cache inhibited access. Refer to **Section 6 Instruction and Data Caches** for more information on invalidating the data cache.

2. Shared-Unmodified—The data in this line is shared among processors, so other caches may have a copy of this line. However, this line is unmodified with respect to memory.

3. Exclusive-Modified—Only one processor (this processor) has a copy of the data in this line in its internal cache, and the line has been modified with respect to memory (the line is dirty). Note that if any word in the line is modified, then the entire line is dirty.

4. Exclusive-Unmodified—Only one processor (this processor) has a copy of the data in this line in its internal cache, and the line is unmodified with respect to memory.

NOTE

Throughout this section, the following nomenclature is used: when a cache line is referenced as modified, it is exclusive-modified (no shared-modified state exists within the MC88110 caches). When a cache line is referenced as exclusive, it can be assumed that it is not relevant to that context whether it is exclusive-modified or exclusive-unmodified.

During a data cache access, the MC88110 may cause the cache line that contains the data being read or written by the processor to change state. The state of the cache line after the access depends on the previous state of the line, the type of access, and whether the access resulted in a hit or a miss in the data cache.

11.3.2 Memory Update Policy

The MC88110 provides three memory update modes: write-back, write-through, and cache inhibited. Each page or block of memory must be specified to be in one of these modes within the corresponding page or block descriptor in the data memory management unit (DMMU). The MC88110 also has a store-through option for the store instruction which allows individual accesses to be performed in write-through mode, even if the corresponding page or block is designated as operating in write-back mode. If the FWT bit is set in the data MMU/cache control register (DCTL), all store instructions are forced to write through the data cache, regardless of the page or block status or store-through option.

In write-back mode, memory is not updated each time a corresponding cache line is modified. In write-through mode, write operations update memory every time a write occurs. When the access is cache inhibited, data is never stored in the data cache of the MC88110, but read and write operations access main memory directly. All three modes of operation have specific advantages and disadvantages; therefore, the choice of which mode to use depends on the system environment as well as the application.

Figure 11-2 illustrates how the MC88110 selects a memory update policy.
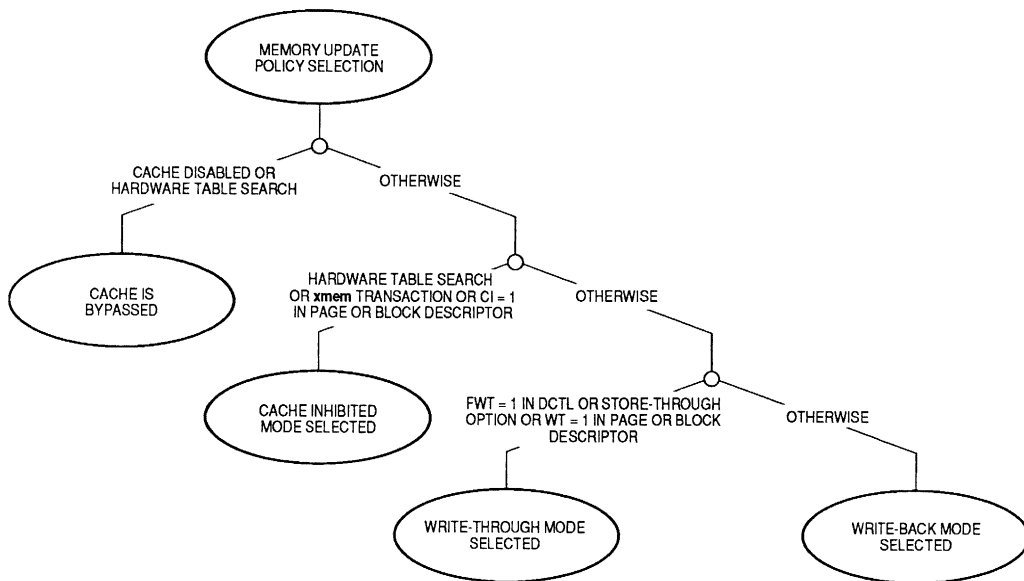
Figure 11-2. Memory Update Policy Selection

11.3.2.1 WRITE-BACK MODE. When writing to memory in write-back mode, store operations for cacheable data do not necessarily cause an external bus transaction to update memory. Instead, memory updates only occur when a modified line is to be replaced due to a cache miss or when another bus master attempts to access a specific address for which the corresponding cache entry has been modified (i.e., a dirty cache entry). For this reason, write-back mode may be preferred when external bus bandwidth is a potential bottleneck—e.g., in a multiprocessor environment without a secondary cache.

The write-back mode is also well suited for high-use data that is closely coupled to a processor, such as local variables. Both reads and writes to memory in write-back mode that hit the on-chip data cache provide maximum data throughput for the program.

In general, addresses at which data is to be used by only one processor and with no other bus master should be designated as local (G = 0) and write-back (WT = 0) by the DMMU for maximum performance. The G and WT bits are set in the corresponding BATC or PATC descriptor (see **Section 8 Memory Management Units**).

If more than one processor uses data stored in a page or block which is designated as write-back, snooping must be enabled to allow copyback operations and cache invalidations of modified data. When snooping is enabled, the page or block should be marked as global (G = 1) and write-back (WT = 0).

11.3.2.2 WRITE-THROUGH MODE. Write operations to memory in write-through mode always update memory as well as the data cache on cache hits. Write-through mode is used when the external memory and on-chip data cache images must be the same, such as occurs with video memory or when there is shared (global) data that may be used frequently.

In write-through mode, memory is always updated during write operations, and global transactions cause other processors' snoop logic to invalidate or copyback their cached images of the memory being updated.

A store-through option may be specified for any triadic register form of a store instruction. A store-through access operates in precisely the same manner as an operation in write-through mode even if write-back mode is specified for the page or block being accessed; however, if the page or block is specified as cache inhibited, the store-through option has no effect.

Also, if the FWT bit is set in the DCTL, all store instructions are forced to write through the data cache, regardless of the page or block status or store-through option. Refer to **Section 8 Memory Management Units** for more information about the programming of the FWT bit in the DCTL register.

11.3.2.3 CACHE INHIBITED MODE. If a memory location is designated as cache inhibited, data from this location is never stored in the data cache. In addition, **xmem** operations and table search operations are always performed as if cache inhibition is in effect regardless of the memory update mode for the location being accessed.

Hardware table search operations automatically bypass the cache; therefore, whenever an MMU performs a hardware table search operation, the segment and page descriptors are never fetched from the data cache. However, the CI signal is not asserted on the external bus during the transactions caused by the table search operation, allowing descriptors to be cached in secondary external caches.

11.3.3 Data Cache Coherency

The data cache can automatically maintain coherency between cached and in-memory copies of data. To maintain this coherency, the MC88110 uses a write invalidate with intervention protocol on the external bus to ensure that, at all times, only one on-chip cache in the system has a modified copy of a given cache line. The protocol allows other caches on the bus to have local copies which are all consistent. When an MC88110 writes data to a memory location shared by other processors, the other processors are notified that their copy of the line containing that data will be stale and must be invalidated.

The MC88110 snoops bus transactions by monitoring externally initiated bus transactions and comparing all global addresses to the internal data cache tags. A snoop hit occurs when the on-chip data cache tag for a valid entry matches the address on the bus. Two separate, independently accessible copies of the tags are maintained to allow bus snooping to occur in parallel with on-chip processor data cache accesses. Processor access to the data cache is interrupted only in the event of a snoop hit when the snooping processor copies back a modified line to memory.

When monitoring external bus transactions, if snooping is enabled and a global address is detected (GBL signal asserted during the transaction) which matches one of the cache tags, a snoop hit occurs. When a snooping CPU hits with a modified entry, the snooping CPU asserts the SSTAT1 (snoop status) signal. The SSTAT1 output may then be directly or indirectly coupled to each CPU's ARTRY input, forcing the CPU that initiated the access to retry the access after the modified data has been written to memory by the CPU that had the snoop hit.

This protocol is referenced as a snoop retry exchange throughout the remainder of this section. In addition, the terms initiating CPU and snooping CPU are used throughout. The initiating CPU is the processor that is the bus master at the beginning of a bus transaction. The snooping CPU is the processor that snoops this transaction.

Snooping is enabled in the MC88110 by setting the SEN bit in the DCTL register (see **Section 8 Memory Management Units**). After a processor reset, snooping is disabled.

Figure 11-3 shows the complete flow followed by the MC88110 for a snoop operation. The following sections describe the operations depicted in the flow diagram.

11
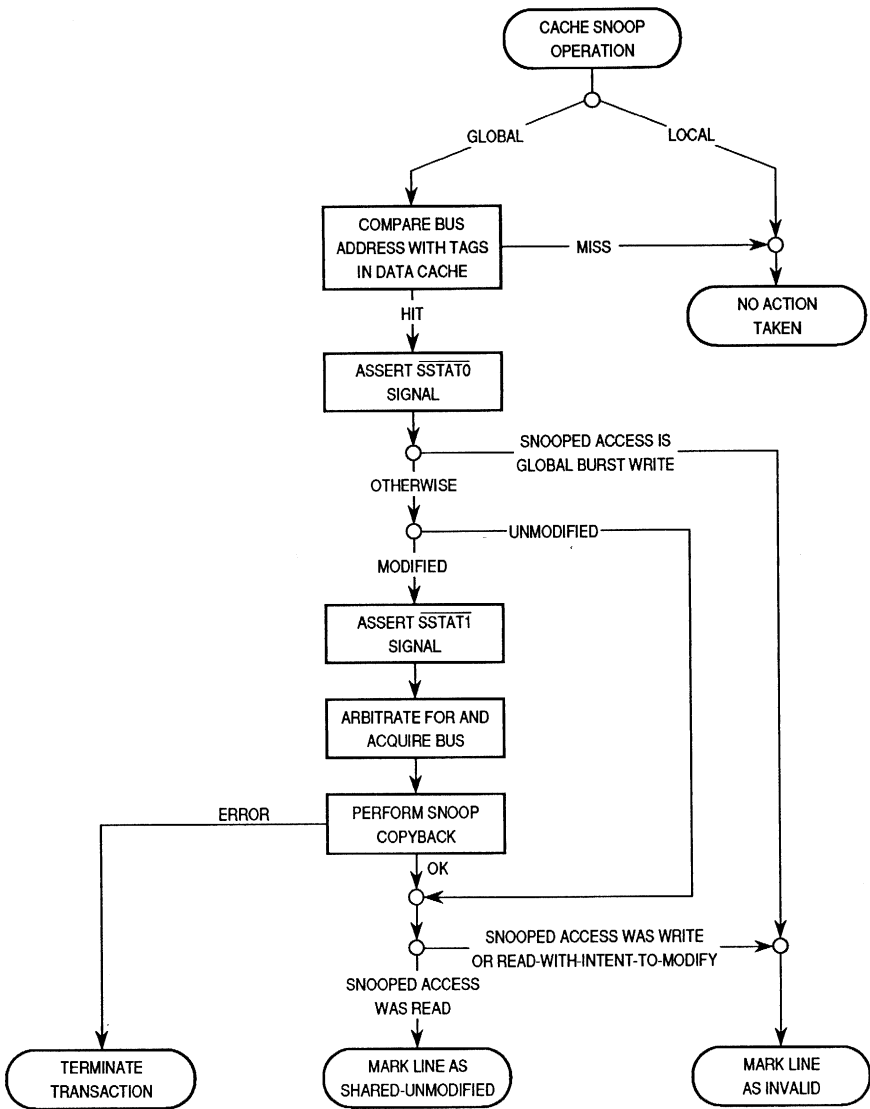
Figure 11-3. Cache Snoop Operation Flow

11.3.3.1 BUS SNOOPING FLOW FOR TRANSACTION WITHOUT INTENT-TO-MODIFY. The MC88110 performs the following actions when snooping an external read transaction. These actions represent the logical flow of operations; since the MC88110 employs a high degree of concurrency, some of the operations are performed in parallel.

When an MC88110 snoops a global read transaction that hits in the data cache, it determines if the cache data is modified or not. If the line is unmodified and exclusive, the MC88110 marks the line as shared-unmodified. In this manner, the MC88110 recognizes that other processors have read access to the global data. If the line is already marked as shared-unmodified, no action is taken.

If the line is internally modified, the MC88110 signals a snoop retry to the processor that initiated the transfer. The initiating processor should then abort its transaction and release the bus. The snooping processor then arbitrates for mastership of the bus, writes its modified copy of the line to memory, and marks the line as shared-unmodified in its cache. The initiating processor then arbitrates for mastership of the bus and attempts the aborted transaction again. The initiating CPU snoops the bus while it is waiting to retry the aborted transaction.

11.3.3.2 BUS SNOOPING FLOW FOR TRANSACTION WITH INTENT-TO-MODIFY. The MC88110 performs the following actions when snooping an external write or an external read-with-intent-to-modify transaction on the bus. These actions represent the logical flow of operations; since the MC88110 employs a high degree of concurrency, some of the operations are performed in parallel.

A snooping processor that has a snoop hit during a global single-beat write or global read-with-intent-to-modify operation must determine if the cache line is modified or not. If the cache line that hit is unmodified, no additional bus transaction occurs, but the cache line is marked as invalid.

If the cache line that hit is modified, the snooping processor signals a snoop retry to the processor that initiated the transfer. The initiating processor then aborts its transaction and releases the bus. The snooping processor then arbitrates for mastership of the bus, writes its modified copy of the line to memory, and marks the line as invalid in its cache. The initiating processor then arbitrates for mastership of the bus and attempts the aborted transaction again. The initiating CPU snoops the bus while it is waiting to retry the aborted transaction.

If the MC88110 has a snoop hit during a global burst write, it invalidates the cache line without copying the line back regardless of whether or not the $\overline{\text{INV}}$ signal is asserted. The MC88110 will never perform a global burst write. If a global burst write is detected, it must have been generated by an external device which is overwriting some portion of memory (e.g., a DMA controller); thus, there is no reason to copyback the line before invalidating.

11

Read-with-intent-to-modify transactions that affect cache coherency are locked read/write transactions (initiated by the **xmem** instruction), cache line fill operations (reads) that occur due to write misses, and allocate loads. In the case of the **xmem** instruction (when the **xmem** is programmed as a read operation followed by a write; see **Section 10 Instruction Set**), a snooping processor can hit if the read-with-intent-to-modify transaction is global, copyback its modified data, and invalidate the line in the data cache. The snooping processor then monitors the write portion of the **xmem** instruction but never hits, since the line was already copied back. When the **xmem** instruction is programmed as a write followed by a read, a snooping processor can hit if the write is global and then cause the write portion of the transaction to be retried.

When a read-with-intent-to-modify access caused by a write miss or an allocate load occurs, other caches on the bus must invalidate local copies of that cache line. If another processor on the bus recognizes the address as global and has a modified copy of the data in its on-chip cache, it signals a snoop retry. Upon receipt of the retry signal, the initiating CPU aborts the cache line fill transaction and relinquishes the bus. The snooping CPU then acquires the bus and updates memory with its copy of the cache line. The initiating CPU then arbitrates for mastership of the bus and attempts the aborted cache line fill again.

11.3.3.3 EXAMPLE FLOW FOR SNOOPING PROTOCOL. Figures 11-4 through 11-10 illustrate an example of how snooping maintains cache coherency in a multiprocessor configuration. The example assumes that there are two MC88110 CPUs that share one common external bus with main memory and illustrates the progression of events for the case of a snoop hit for a transaction without intent-to-modify. Each of the figures show a cache line within CPU1 and CPU2 and the associated line address tags. The state of the cache line (invalid (INV), shared-unmodified (SU), exclusive-unmodified (EU), or exclusive-modified (EM)) is also shown as well as the next state of the line as a result of bus transactions or snooping. This example only shows one line in the data caches for simplicity.

In normal operation, with address translation enabled, the addresses generated by the program are logical addresses (LA). The logical addresses are then translated by the MMU into physical addresses (PA). For this example, address translation is disabled, so the PA is the same as the LA. Also, to simplify this example, the starting address is shown as $0000. Address $0008 corresponds to double word 1, address $0010 corresponds to double word 2, etc. Line read operations perform four consecutive double-word reads from memory addresses $0000, $0008, $0010, and $0018 to the cache line using the efficient burst mode transfer mechanism of the MC88110. Line copyback operations write (burst) the four double words from the cache line back to memory.

For this example, all addresses are assumed to be mapped as global, write-back, cacheable, and not write-protected (G = 1, WT = 0, CI = 0, WP = 0). Also, in this example, the caches are assumed to be operating in the four state model since the shared input (SHD) is connected to $\overline{\text{SSTAT0}}$ (for more information on the four state model, see **11.3.4 Data Cache State Transitions**).

Figure 11-4 shows the caches in their initial state, with both lines invalidated and their contents unknown. This is the state of the data cache after reset, assuming that the system software has invalidated all the data cache lines.

Figure 11-5 shows CPU2 performing a load word operation from location $0000. There is a data cache miss and the CPU reads a line from memory to fill the cache line. CPU1 monitors (snoops) the bus transaction, but does not find a tag match (a miss) since the entire data cache is marked as invalid. Cache2 supplies CPU2 with the required word and the state of the cache line is updated to the exclusive-unmodified state.

Figure 11-6 shows CPU1 reading a word from address $0008, which misses for the selected cache line. A line fill operation is performed as before, which reads four double words from memory starting at location $0008 (forwarding this data directly to CPU1) and wrapping around to location $0000. CPU2 snoops the global transaction and finds a tag match (a snoop hit). The state of the line changes to shared-unmodified in both caches since both have a copy of the data that is unmodified with respect to memory.

Figure 11-7 shows CPU2 performing a store operation of a word to address $0000. A cache hit occurs, and, since the address was global, an invalidation bus transaction is performed. The invalidation transaction notifies CPU1 that its local copy of the line is no longer valid, so CPU1 marks its cache line as invalid. CPU2 then updates the line with the new data and marks the line exclusive-modified.

CPU2 now has exclusive ownership of the entire line of data that is modified with respect to memory. The exclusive status guarantees CPU2 that no other processor on the bus can cache a valid copy of the line. All subsequent load and store operations performed by CPU2 that map to this line complete without accessing memory.

Figure 11-8 shows CPU1 attempting a load from location $0008. The transaction misses in the cache (because the entire line is marked as invalid), which forces CPU1 to access memory. CPU2 snoops the access, recognizes that it has cached modified data requested by CPU1, and aborts the line read operation by CPU1.

Figure 11-9 shows CPU2 writing back the exclusive-modified line to memory and marking the cache line as shared-unmodified. Since CPU2 had exclusive ownership of the line, no other MC88110 will have a snoop hit on the copyback of the exclusive-modified data. Exclusive ownership implies that only one CPU has a copy of the line cached.

Figure 11-10 shows CPU1 regaining control of the bus to complete the read that was previously aborted by CPU2. The cache line is updated from memory (critical word first), the required word is supplied to CPU1, and the line is marked as shared-unmodified.

11
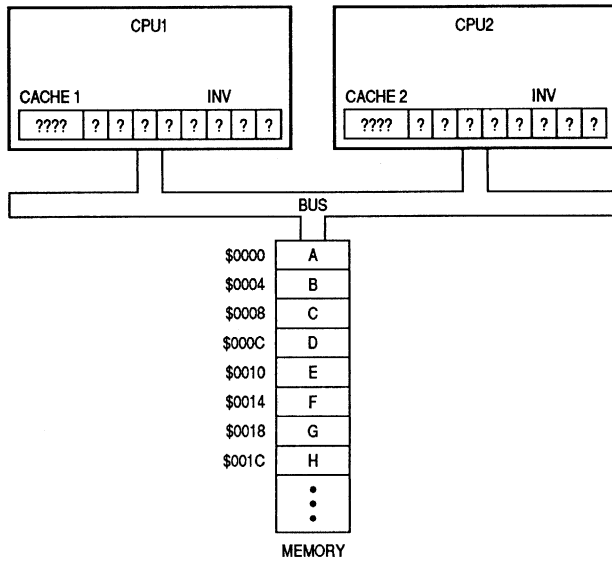
Figure 11-4. Initial State of System
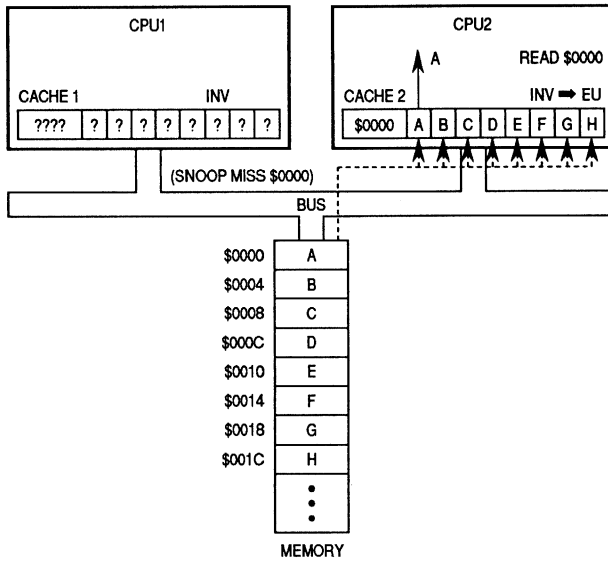
Figure 11-5. CPU2 Load, Data Cache Miss
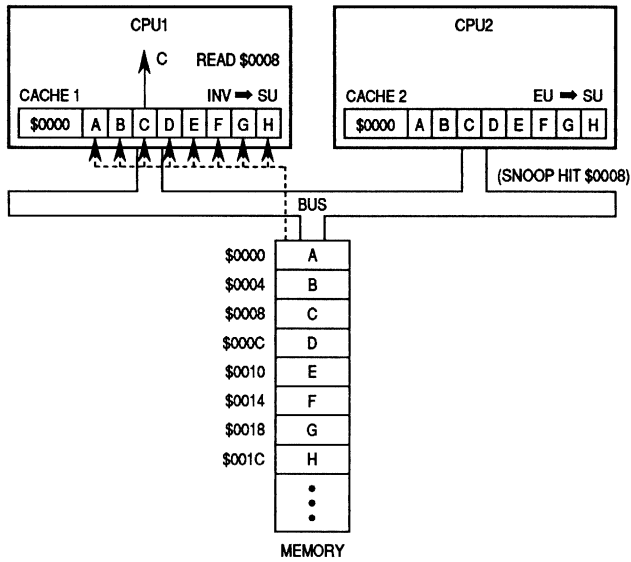
Figure 11-6. CPU1 Load, Data Cache Miss
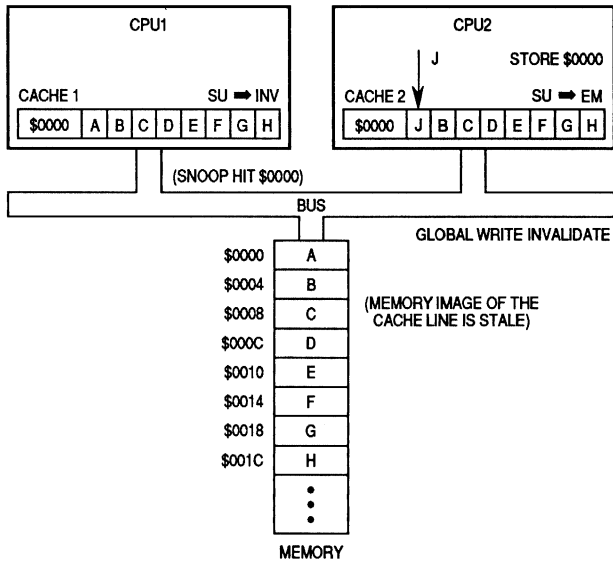
Figure 11-7. CPU2 Store, Data Cache Hit

11
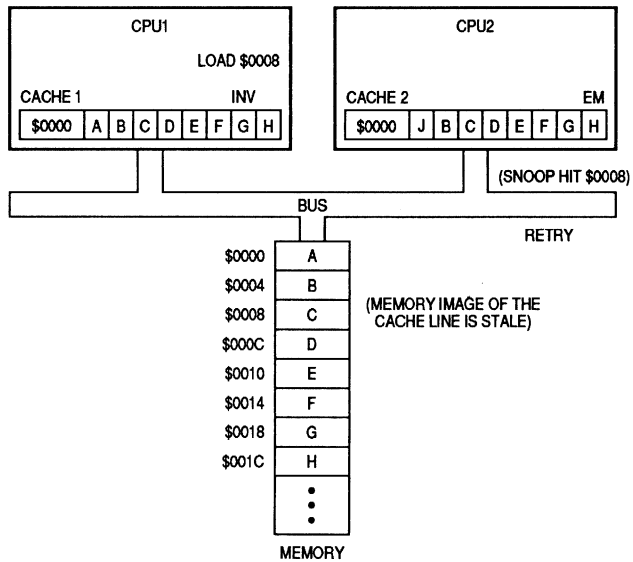
Figure 11-8. CPU1 Load, Cache Miss, Line Read Retried
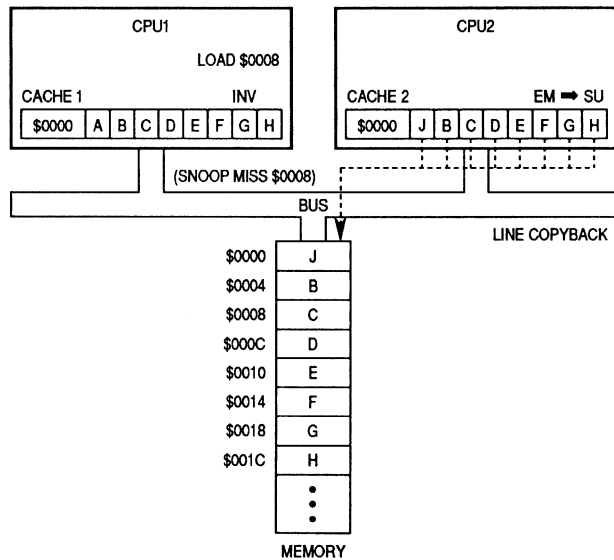
Figure 11-9. CPU2 Line Copyback
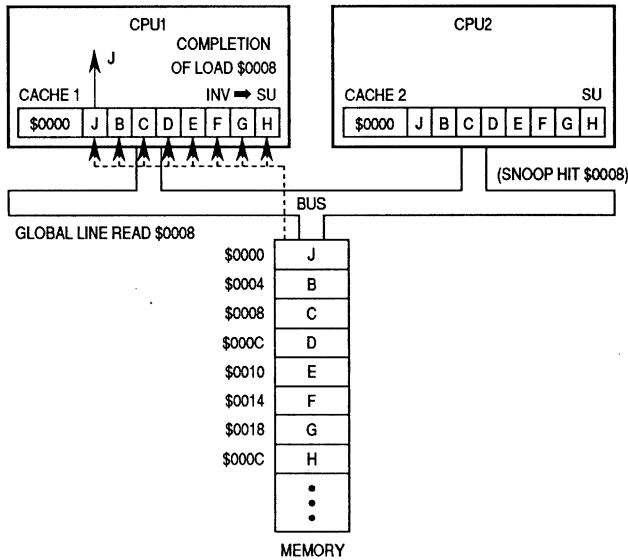
Figure 11-10. Completion of CPU1 Load, Cache Miss

11.3.4 Data Cache State Transitions

The MC88110 cache state logic is implemented as a four state design, but also supports a three state model. The three state model includes all of the states except the exclusive-unmodified state. When operating in the three state model, all internal cache state transitions are visible on the external signals of the MC88110 to allow for the construction of coherent external secondary caches. In the four state model, the transition from the exclusive-unmodified state to the exclusive-modified state for a write hit is not broadcasted on the bus. The distinction of whether the three or four state model is in use is determined by the status of the SHD input signal.

State transition diagrams for the data cache in the four state model are shown in Figures 11-11 and 11-12 and described in the following paragraphs. Figure 11-11 shows the state transition diagram for the cache operating in write-back mode, and Figure 11-12 shows the state transition diagram for the cache operating in write-through mode. State transitions for the cache in the three state model are shown in Figure 11-13. All other operations that are not explicitly shown in these diagrams do not affect the cache state.

11

In the following diagrams, state transitions labeled as "shared..." (e.g., shared read miss) imply that the $\overline{SHD}$ input signal to the MC88110 was asserted during the line fill operation. Transitions labeled as "Exclusive..." imply that the $\overline{SHD}$ input signal was negated during the line fill. Other snooping MC88110s on the bus should drive the $\overline{SHD}$ signal with their snoop hit status output ($\overline{SSTAT0}$). Thus, when the MC88110 reads a line into its cache with a normal line fill (not read-with-intent-to-modify), the line is marked as either shared- or exclusive-unmodified, depending on whether or not other processors on the bus have copies of the line in their caches. Systems implementing a three state cache model simply keep the $\overline{SHD}$ signal asserted and force all line fills to be marked as shared-unmodified. Note that during line fills for write misses in write-back mode, the $\overline{SHD}$ signal is ignored (i.e., write miss line fills are always marked as exclusive-modified).

Figure 11-11 shows all state transitions possible for the data cache in write-back mode for the four state model. A line can change state due to a cache miss. On a cache miss, the address of the missed data is used to select two cache lines. If one of the lines is invalid then it is selected to receive the data. If both lines are valid, then a pseudorandom algorithm is used to select one of the two lines. If both lines are invalid, then line 0 is selected. Replacing a cache line with a line from main memory is referred to as replacement. For any initial state, an exclusive read miss with replacement will change the line state to exclusive-unmodified, a shared read miss with replacement will change the line state to shared-unmodified, and a write hit will change the line state to exclusive-modified. A write miss with replacement of an invalid line will change the line state to exclusive-modified. In a multi-processor system a snoop hit on a read will change the snooping processor's line state to shared-unmodified. A snoop hit on a write or read with intent-to-modify will change the snooping processor's line state to invalid (after a copyback of the data, if modified).

Write operations in write-through mode leave the cache state unaffected. Figure 11-12 shows all state transitions possible for the cache when in write-through mode. The exclusive-unmodified state cannot be reached in write-through mode. If a cache line is already in either of the exclusive states when write-through mode is selected, the line will not change state while in write-through mode. This does not cause coherency problems, but if the mode is changed back to write-back, some data may be copied back to memory which is already consistent with the line in the data cache.

11
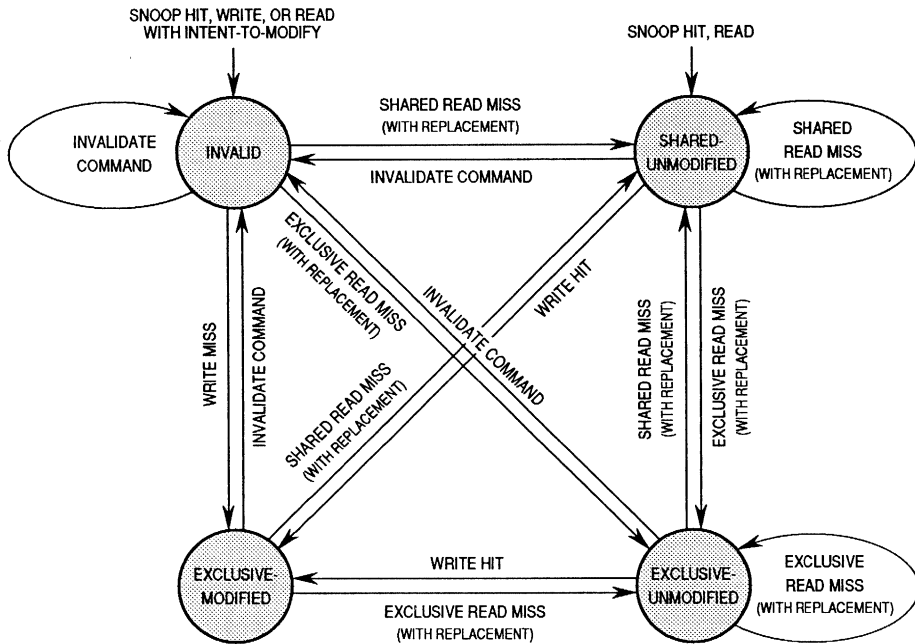
**Figure 11-11. Data Cache in Write-Back Mode
State Diagram (Four State)**
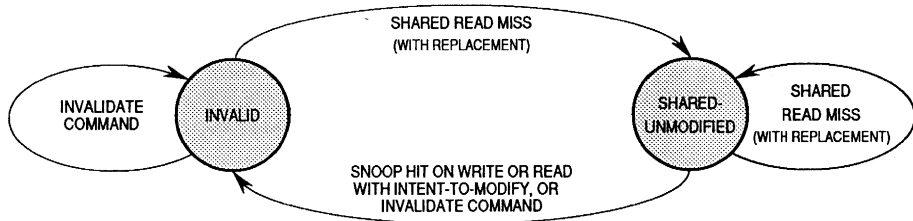
**Figure 11-12. Data Cache in Write-Through Mode
State Diagram (Four State)**

Figure 11-13 shows all possible state transitions for the data cache in the three state model. The three state model does not include the exclusive-unmodified state.
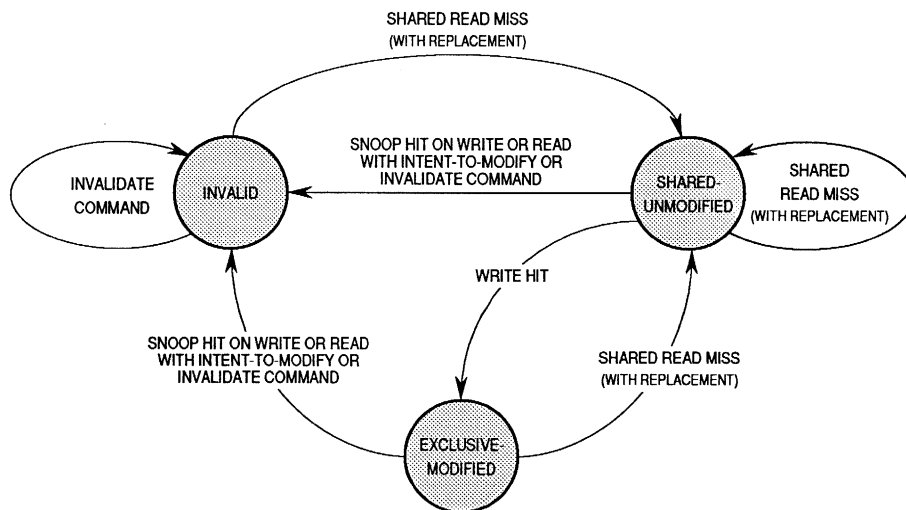
Figure 11-13. State Diagram for Data Cache in the Three-State Model

If a snooping processor performs a snoop copyback operation, then the snooping processor's cache line will change state to either shared-unmodified (for a read) or invalid (for a write or a read with intent to modify) depending on what caused the snoop copyback. In this case, in order to maintain a completely coherent secondary cache, the external logic must track the operation that caused the copyback to determine what the resulting state change from the snoop copyback will be within the on-chip data cache.

There are potential benefits to both the three and four state models. The three state model is useful for maintaining coherency with secondary caches. However, the three state implementation can cause lower performance than the four state implementation. In the three state implementation, the exclusive-unmodified state does not exist; therefore, all data is read in as shared-unmodified. A write hit to shared-unmodified data causes the snooping processor to perform an invalidation transaction on the bus. If the data had been read in as exclusive-unmodified (as in the four-state model), then a write hit would simply change the state of the data to be exclusive-modified, and no bus traffic would occur.

In the preceding explanations, two alternatives were given for secondary cache support: write-through mode and the three state model. The three state model requires more external logic to implement a secondary cache but provides higher performance than write-through mode. The only coherency operations that are required for the secondary cache are invalidation transactions. However, in write-through mode, every write operation creates bus traffic and therefore may cause lower performance than the three state model.

11.4 BUS ARBITRATION

Arbitration for bus mastership in a multi-master system is performed by external arbitration logic and the arbitration signals of the MC88110. Table 11-11 lists the arbitration signals for the MC88110. Note that address bus busy ($\overline{ABB}$) and data bus busy ($\overline{DBB}$) are I/O signals. These signals are inputs while the MC88110 is arbitrating for the respective buses and outputs while the MC88110 has mastership of each of the buses.

Table 11-11. Bus Arbitration Signals

Signal Name	Mnemonic	Type
Bus Request	$\overline{BR}$	Output
Bus Grant	$\overline{BG}$	Input
Address Bus Busy	$\overline{ABB}$	I/O
Data Bus Grant	$\overline{DBG}$	Input
Data Bus Busy	$\overline{DBB}$	I/O

The following paragraphs describe the arbitration protocol used by the MC88110 for systems with and without split data and address buses. This section also discusses the concept of parking, where the arbitration overhead can be eliminated.

11.4.1 Address Bus Arbitration

When the MC88110 needs to perform an external bus access and it is not parked ($\overline{BG}$ is negated), it asserts $\overline{BR}$ and continues to assert $\overline{BR}$ until it has been granted mastership of the bus and the bus is available. The external arbiter grants mastership of the bus to the potential master by asserting the bus grant signal $\overline{BG}$. Because the $\overline{ABB}$ signal is asserted by the current master to indicate address bus mastership, the potential master determines that the bus is available when the $\overline{ABB}$ signal is negated. A qualified bus grant is defined as $\overline{BG}$ asserted and $\overline{ABB}$ negated (as an input). The potential master does not assume address bus mastership until it receives a qualified bus grant.

When a parked MC88110 needs to perform an external bus access, it qualifies its bus grant with $\overline{ABB}$. If $\overline{ABB}$ is negated, then the MC88110 has a qualified bus grant and it can assume address bus mastership.

When the MC88110 receives a qualified bus grant, it assumes address bus mastership by asserting the $\overline{ABB}$ signal and negates the $\overline{BR}$ output signal (unless the transaction is the first half of an **xmem** operation). At the same time, the MC88110 drives the address for the requested access onto the address bus and asserts the transfer start ($\overline{TS}$) signal to indicate the start of a new transaction.

11

When designing external bus arbitration logic, it is important to note that the MC88110 may assert $\overline{BR}$ but never use the bus after it receives the qualified bus grant. One example of this is in a system using snooping. If the MC88110 asserts $\overline{BR}$ in order to perform a replacement copyback operation, it is possible for another device to invalidate that line before the MC88110 is granted the bus. Then, once the MC88110 is granted the bus, it no longer needs to perform the copyback, and it never asserts $\overline{ABB}$ for this case.

11.4.2 Data Bus Arbitration

In addition to signaling the start of a new transaction, the assertion of the $\overline{TS}$ output signal implies a data bus request. The arbitration for the data bus is very similar to the arbitration for the address bus. The $\overline{TS}$ signal serves the same function for the data bus as the $\overline{BR}$ signal does for the address bus; however, $\overline{TS}$ is asserted for only a single clock cycle. As with the address bus, the MC88110 only assumes data bus mastership when it has been granted the data bus and the data bus is available.

The external arbiter grants data bus mastership by asserting the data bus grant ($\overline{DBG}$) signal. The potential data bus master determines that the bus is available when the data bus busy ($\overline{DBB}$) signal is negated. A qualified data bus grant is defined as $\overline{DBG}$ asserted and $\overline{DBB}$ negated (as an input).

When the processor receives a qualified data bus grant, the MC88110 asserts $\overline{DBB}$ and data transfers may begin on the next rising clock edge. A design alternative for nonsplit bus systems is to ground the $\overline{DBG}$ signals for all CPUs, as both address and data bus arbitration can be controlled by $\overline{ABB}$ alone.

Note that the data handshake must occur for all transfers except transfers in split-bus systems which are terminated with $\overline{ARTRY}$. Therefore, even for invalidate cycles in which $\overline{MC}$ is negated and no data must be transferred, the memory system must assert the $\overline{DBG}$ signal for the transaction to terminate properly.

11.4.3 Bus Arbitration Timing Examples

Figures 11-14 and 11-15 show the relative timing of the bus arbitration signals for some simple cases of bus arbitration. Note that there are separate signals shown for $\overline{ABB}$ and $\overline{DBB}$ as inputs and as outputs (even though there is only one $\overline{ABB}$ and one $\overline{DBB}$ signal on the MC88110). This is to clarify when these signals are monitored as inputs, when they are driven as outputs, and when they are ignored. In systems with multiple MC88110s, the multiple $\overline{ABB}$ signals can be tied together, as can the multiple $\overline{DBB}$ signals. The combined $\overline{ABB}$ and $\overline{DBB}$ signals should be tied to pull-up resistors to keep the signal negated when no devices are driving the signals. For all timing diagrams that follow Figure 11-15, the combined $\overline{ABB}$ and $\overline{DBB}$ signals are shown with the assumption that pull-up resistors are being used.
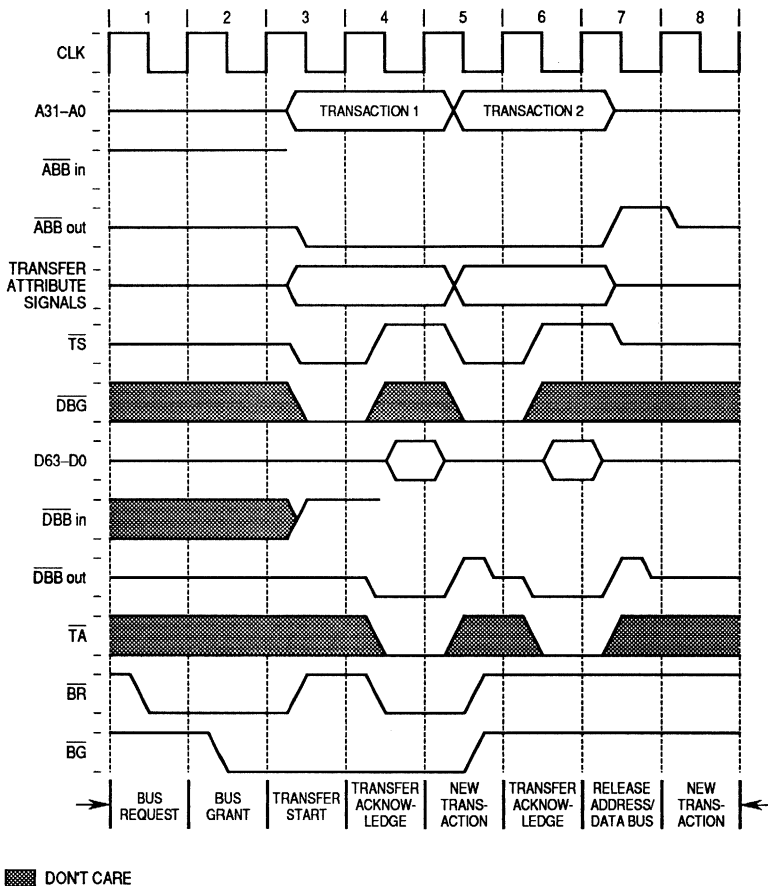
Figure 11-14. Bus Arbitration Example Timing

In clock cycle one of Figure 11-14, the MC88110 asserts $\overline{BR}$ and monitors $\overline{BG}$ and $\overline{ABB}$. Note that all of the MC88110 output signals except those used for arbitration are three-stated during clock cycles one and two because the MC88110 is not the current bus master. However, it is likely that these signals are driven by other bus masters in the system during that time. Since it receives a qualified bus grant on the rising edge of clock 3, the MC88110 asserts $\overline{ABB}$ and $\overline{TS}$, negates $\overline{BR}$, and drives the appropriate values onto the address bus and transfer attribute signals. On the following clock cycle, the MC88110 receives a qualified data bus grant, so it asserts $\overline{DBB}$ and completes the transaction.

The MC88110 re-asserts $\overline{BR}$ in clock 4 because it has another transaction to perform ($\overline{BR}$ may be asserted any time during an ongoing transfer except at the same time as $\overline{TS}$ of a non-**xmem** transaction). Since the arbiter kept $\overline{BG}$ asserted, the MC88110 is parked and can skip the clock cycle needed for address bus arbitration, keep $\overline{ABB}$ asserted, and start the second transaction as soon as the first transaction is terminated. If $\overline{BG}$ had not

been asserted on the rising edge of clock 5, then the MC88110 would have negated $\overline{ABB}$ and waited for a qualified bus grant before beginning the second transaction. Also, note that $\overline{DBB}$ was negated between the two transactions regardless of the state of $\overline{DBG}$. The MC88110 must re-arbitrate for the data bus for each transaction; however, in many cases (including this one) it does not cause any delay in the memory transaction. This protocol enforces at least one clock cycle for data bus turnaround.

In Figure 11-14, many of the input signals are ignored a majority of the time. The $\overline{DBG}$ signal is only monitored between the time when $\overline{TS}$ is asserted and when the MC88110 assumes mastership of the data bus, which in this case was one clock cycle for each memory transaction. The transfer acknowledge signal ($\overline{TA}$) is only monitored when the MC88110 has taken mastership of the data bus. $\overline{TA}$ is used to signify when and if the transfer has been successfully completed (the function of the $\overline{TA}$ signal is described in detail in **11.6 Termination of Bus Transactions**).

Note that in Figure 11-14, when the MC88110 is no longer using the address and data buses, it negates $\overline{ABB}$ and $\overline{DBB}$ before three-stating the signals. As mentioned previously, these signals should be tied to pull-up resistors. The MC88110 negates the signal before three-stating so that the signals meet the setup time for the next clock edge. The step in $\overline{DBB}$ in clock 6 of Figure 11-14 indicates how the MC88110 negates the signal, three-states it, and then immediately asserts it again.

Figure 11-15 shows an example of bus arbitration in which the data bus is not immediately available for the MC88110. This case is identical to the previous example until clock 3, where $\overline{DBG}$ is negated rather than asserted. Therefore, the MC88110 does not assume data bus mastership until $\overline{DBG}$ is asserted, and the transaction completes as before.

11.4.4 Bus Parking

To avoid the latency overhead of arbitration, it may be desirable to park the MC88110 on the system address bus. The MC88110 is parked when $\overline{BG}$ is asserted whether or not the processor is requesting bus mastership. If $\overline{BG}$ remains asserted until an internal bus request occurs, the MC88110 completes the arbitration sequence without any overhead and can begin the transaction without even asserting $\overline{BR}$. Thus, bus parking provides a performance advantage in that bus accesses occur without any delay for the arbitration protocol.

11
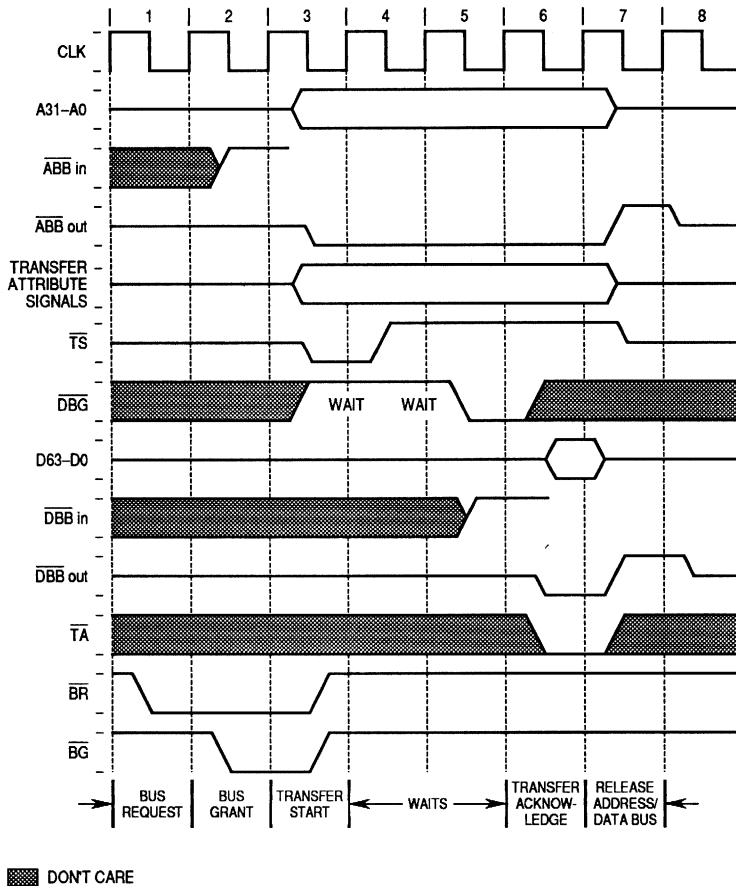
Figure 11-15. Data Bus Arbitration Example Timing

Figure 11-16 shows an example of the arbitration protocol when bus parking is used. Initially, an alternate master is the bus master and performs a data transaction. At the end of this transaction, the arbitration logic parks the MC88110 on the address bus by asserting the MC88110 $\overline{BG}$ input. Clock cycles 4 and 5 show that no device is using the bus but the MC88110 is parked on the address bus.

11
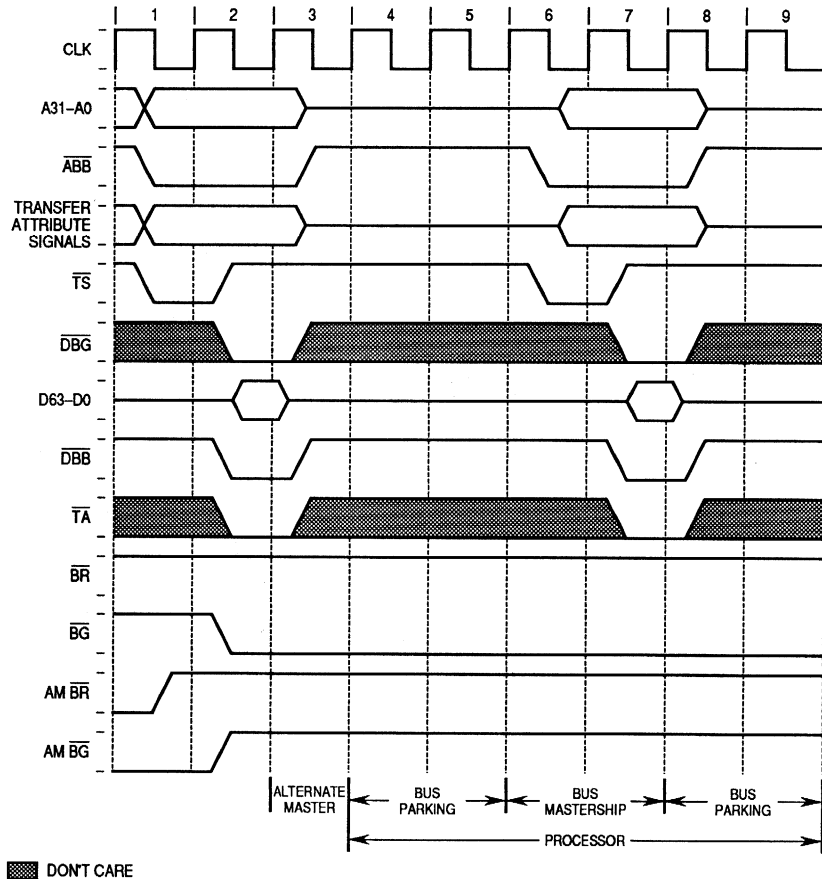
Figure 11-16. Bus Parking

In clock 6, the MC88110 initiates a transaction by driving the address and control information and asserting $\overline{TS}$ and $\overline{ABB}$. $\overline{ABB}$ is asserted to indicate that the address bus is in use (slow masters may assert $\overline{ABB}$ without driving a valid address). $\overline{ABB}$ only remains asserted after the transaction is terminated if the MC88110 is immediately initiating another transaction and the device is parked at the time that the initial transaction is normally terminated. Otherwise, $\overline{ABB}$ negates as usual. $\overline{DBB}$, however, always negates after the transaction is complete.

At the end of the transaction shown in Figure 11-16, $\overline{BG}$ for the MC88110 remains asserted, so the MC88110 remains parked on the address bus.

Caution should be taken when negating $\overline{BG}$ to a parked MC88110, because it is possible for the parked MC88110 to assert $\overline{ABB}$ and start a transfer in the same clock cycle that $\overline{BG}$ is negated. Figure 11-17 shows an example of this scenario.

Figure 11-17 shows $\overline{BG}$, $\overline{BR}$, and the bus busy signals for an MC88110 and an alternate master. In this figure, the $\overline{IBR}$ signal is the internal bus request for the MC88110. In clock 1, the MC88110 is parked on the bus. In clock 2, the MC88110 has an internal bus request, and the alternate master asserts its $\overline{BR}$ signal at the same time. Then, in clock 3, the MC88110 still has a qualified bus grant, assumes address bus mastership, and starts a new transaction. However, during that same clock cycle, the arbiter negates the $\overline{BG}$ to the MC88110 and asserts AM-$\overline{BG}$ to the alternate master. The alternate master then assumes address bus mastership in clock 4, which causes contention on the address bus.
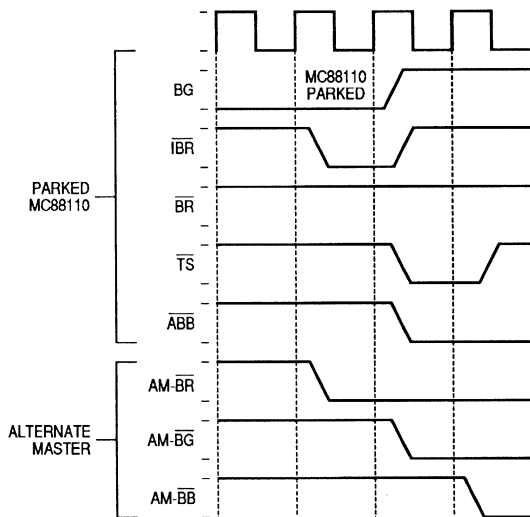
Figure 11-17. Address Bus Contention

There are two ways to prevent this type of contention. If the alternate master is another MC88110, the address bus busy signals should be tied together so the second MC88110 will not assume address bus mastership until the $\overline{ABB}$ signal is negated. If the alternate master does not qualify its $\overline{BG}$ with $\overline{ABB}$, then the arbitration circuitry should wait one clock cycle after unparking the MC88110 on the bus to be sure the MC88110 has not assumed mastership of the bus before granting the bus to the alternate master.

11.4.5 Arbitration for Split Bus Transactions

The MC88110 has the capability to split the address and data buses so that they operate completely independently from one another. For example, in a multiprocessor configuration, the address bus master is the processor driving the address and the data bus master is the processor that drove the address of the current data transfer. The separate control for this arbitration is controlled by the $\overline{AACK}$ input signal. The assertion of this signal by a memory system indicates that the current address has been latched and that the address bus master can relinquish mastership of the address bus.

11

The address bus master begins sampling the $\overline{\text{AACK}}$ input during the clock after $\overline{\text{TS}}$ is asserted. When the master detects that $\overline{\text{AACK}}$ is asserted, it releases the address bus by negating $\overline{\text{ABB}}$ so that another master can acquire the bus. The $\overline{\text{AACK}}$ signal is ignored on any clock that results in the termination of the transaction (e.g., on the last $\overline{\text{TA}}$, $\overline{\text{TEA}}$, or $\overline{\text{TRTRY}}$).

Figure 11-18 shows the relative timing for a split bus transaction. The MC88110 drives an address onto the address bus and then detects the assertion of $\overline{\text{AACK}}$. It then releases the address bus by three-stating it and negating $\overline{\text{ABB}}$, but continues to transfer data onto the data bus. The data transfer proceeds and terminates as in other normal transactions.
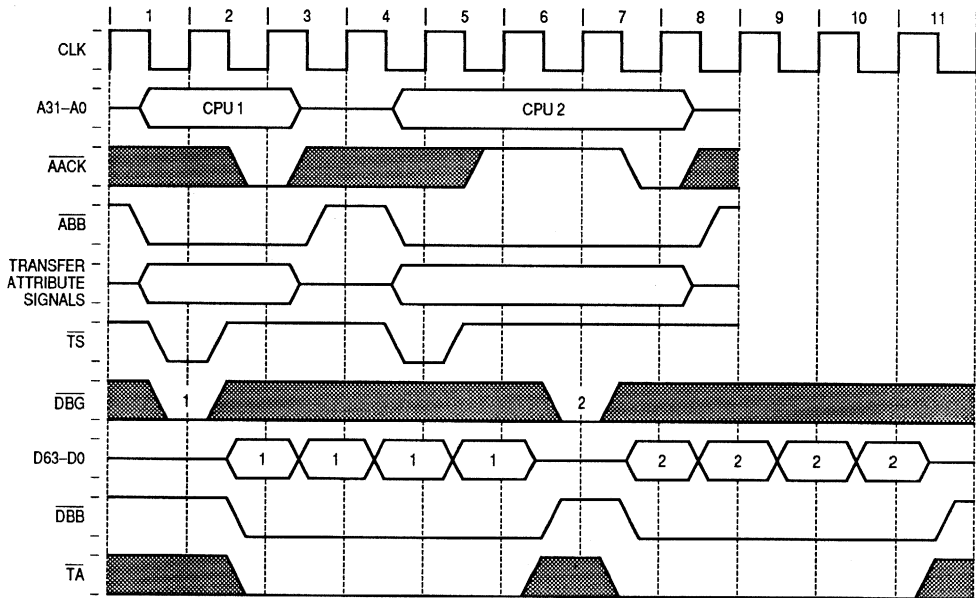
Figure 11-18. Split Bus Transactions Using $\overline{\text{AACK}}$ (One-Level)

Figure 11-18 shows a one-level split transaction. The one-level transaction is characterized by the mastership of the address bus being maintained until the mastership of data bus is acquired. The memory system accomplishes this by asserting $\overline{\text{DBB}}$ to qualify the memory system's assertion of $\overline{\text{AACK}}$. In a one-level pipeline, $\overline{\text{DBG}}$ can be grounded for all CPUs.

As shown in the figure, CPU1 begins a transaction and drives an address on the address bus. CPU1 begins the data transfer on the data bus and receives an $\overline{\text{AACK}}$ on the rising edge of clock 3. Therefore, CPU1 releases the address bus, which allows CPU2 to begin to drive a new address onto the address bus before CPU1 has completed the data transfer. A responding device can latch the new address from CPU2

and begin the data access before the transaction for CPU1 has completed. This feature increases the efficiency of the system because the time that it takes to access the data for the new address can be overlapped with that of a previous transaction.

Note that in this case, $\overline{\text{AACK}}$ is not asserted to CPU2 before CPU1 has completed its data transfer. This is what characterizes this transaction as a one-level split bus transaction. The advantage of implementing a one-level split bus is that the $\overline{\text{DBG}}$ signals to all CPUs can be tied low, which simplifies the data bus arbitration circuitry. After CPU1 completes its data transfer in clock cycle 6, $\overline{\text{DBB}}$ is negated and sampled by CPU2. One clock cycle later, $\overline{\text{AACK}}$ is asserted for CPU2 and the address bus is released for another address bus master.

Multi-level split bus systems can be designed where there are no limitations on how many addresses can be outstanding. Note, however, that for each MC88110 processor, only one outstanding transaction exists at any time. For example, it is possible to have four outstanding transactions at one time for a four-processor system, which corresponds to a three-level split bus system.

Multi-level split bus systems require that the memory system generate the correct $\overline{\text{DBG}}$ (and data) for the correct processor. Figure 11-19 illustrates the timing for a multi-level split bus transaction example. Note that in this case, CPU2 gains mastership and releases it before the data is returned for CPU1's transaction.
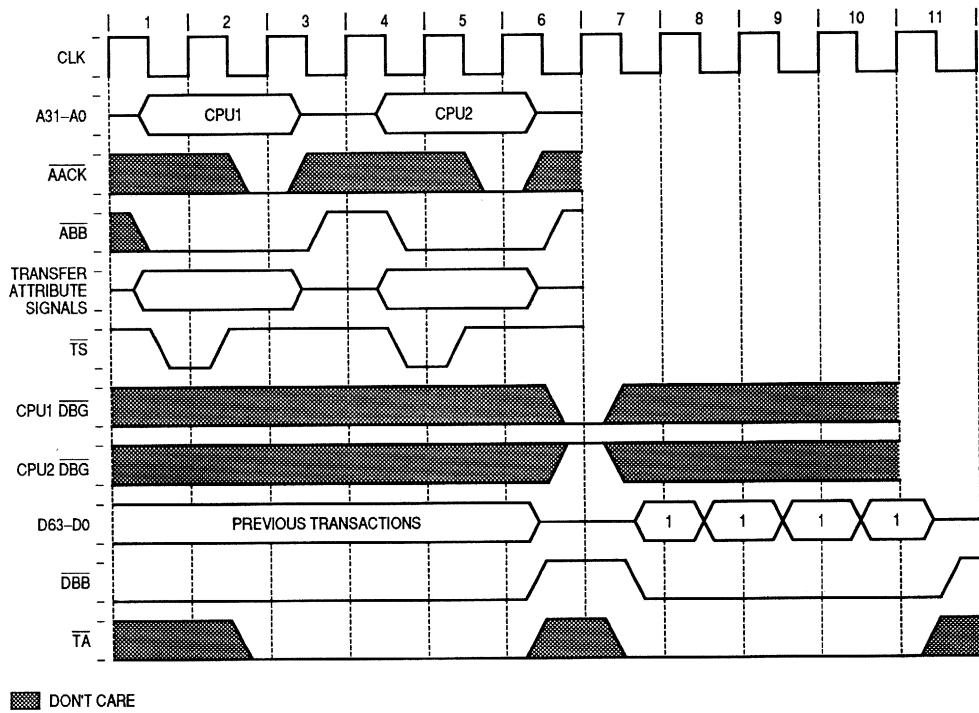
Figure 11-19. Split Bus (Full) Transactions

11.5 DATA TRANSFER MECHANISM

The following paragraphs describe the signals used in the transfer of data between the processor and external devices. The data transfer protocol is described in detail, and examples of the relative signal relationships for the different types of transactions are described. All of the transactions in the timing diagrams for this section are terminated normally. For more information on termination see **11.6 Termination of Bus Transactions**.

11.5.1 Data Transfer Mechanism Signal Overview

The signals that implement the data transfer mechanism for the MC88110 are classified as data transfer signals, transfer attribute signals, and transfer control signals. The transfer attribute signals are summarized in Table 11-12.

Table 11-12. Transfer Attribute Signal Summary

Signal Name	Signal	Asserted	Negated
Read/Write	R/$\overline{W}$	Read	Write
Lock	$\overline{LK}$	Transaction is one of two or more atomic transactions	Transaction is not part of an atomic sequence
Cache Inhibit	$\overline{CI}$	Cache inhibited access	Not a cache inhibited access
Write-Through*	$\overline{WT}$	Write-through memory update mode	Write-back memory update mode
User Programmable Attributes	UPA1–UPA0	UPA bit in ATC entry or area descriptor is set	UPA bit in ATC entry or area descriptor is clear
Transfer Burst	$\overline{TBST}$	Burst transaction	Single-beat transaction
Transfer Size**	TSIZ1–TSIZ0	See Table 11-6	See Table 11-6
Transfer Code	TC3–TC0	See Table 11-7	See Table 11-7
Invalidate	$\overline{INV}$	This signal is broadcast to snooping processors to invalidate the cache line	No need to have snooping processors invalidate the cache line
Memory Cycle	$\overline{MC}$	Data is transferred from processor to an external device	No data transfer to occur (invalidate cycle or allocate load)
Global	$\overline{GBL}$	Data being transferred is global data	Data being transferred is local data
Cache Line ***	CLINE	Transaction involves cache line 1	Transaction involves cache line 0

*   For cache inhibited/disabled accesses, the $\overline{WT}$ signal reflects the WT bit in the ATC entry or area descriptor for that access.

**   Should be ignored for burst transactions which are not touch, flush or allocate transactions.

*** Only valid for burst and invalidate transactions.

11.5.2 Data Byte Lanes and Multiplexing

Data can be transferred on the external bus in either single-beat transactions or burst transactions. The transfer burst ($\overline{\text{TBST}}$) output signal of the MC88110 indicates the type of transaction. For instruction accesses, the TSIZ1–TSIZ0 signals always indicate a double-word transfer. For single-beat data transactions, the transfer size (TSIZ1–TSIZ0) output signals indicate the size of the transaction (see Table 11-13). For burst data transactions, although eight words are always transferred, the TSIZ1–TSIZ0 signals indicate the size of the cache access which caused the burst (i.e., for a burst transaction caused by a cache miss due to a **ld.h**, TSIZ1 = 1 and TSIZ0 = 0 to indicate a half-word read operation even though four double words are transferred).

Table 11-13. Memory Transfer Size and Type

$\overline{\text{TBST}}$	TSIZ1	TSIZ0	Transfer Size
A	x	x	8 Word Burst
N	1	1	Byte (8 Bits)
N	1	0	Half-Word (16 Bits)
N	0	1	Word (32 Bits)
N	0	0	Double Word (64 Bits)

A = Asserted
N = Negated
x = Don't Care

The MC88110 drives the full 32-bit address of the requested data on the address bus. Address bus signals A2–A0 are then used in conjunction with $\overline{\text{TBST}}$ and TSIZ1–TSIZ0 to determine the positioning of valid bytes on the data bus. Table 11-14 lists the valid bytes on the data bus for read and write transactions corresponding to the various encodings of TSIZ1–TSIZ0 and A2–A0. (If $\overline{\text{TBST}}$ is asserted, double words must be transferred.) The entries labeled "A" are byte portions of the requested operand that are read or written during that bus transaction. The entries labeled "—" are not required and are ignored during read transactions and driven with undefined data during write transactions.

11

Table 11-14. Data Bus Requirements for Read and Write Cycles

Transfer Size	TSIZ1	TSIZ0	A2–A0	Byte Lane							
				0	1	2	3	4	5	6	7
Byte	1	1	0 0 0	A	—	—	—	—	—	—	—
	1	1	0 0 1	—	A	—	—	—	—	—	—
	1	1	0 1 0	—	—	A	—	—	—	—	—
	1	1	0 1 1	—	—	—	A	—	—	—	—
	1	1	1 0 0	—	—	—	—	A	—	—	—
	1	1	1 0 1	—	—	—	—	—	A	—	—
	1	1	1 1 0	—	—	—	—	—	—	A	—
	1	1	1 1 1	—	—	—	—	—	—	—	A
Half-Word	1	0	0 0 x	A	A	—	—	—	—	—	—
	1	0	0 1 x	—	—	A	A	—	—	—	—
	1	0	1 0 x	—	—	—	—	A	A	—	—
	1	0	1 1 x	—	—	—	—	—	—	A	A
Word	0	1	0 x x	A	A	A	A	—	—	—	—
	0	1	1 x x	—	—	—	—	A	A	A	A
Double Word	0	0	x x x	A	A	A	A	A	A	A	A

x = Don't care

For double-word accesses, TSIZ1 = 0, TSIZ0 = 0, and all bytes are labeled "A". For word accesses, TSIZ1 = 0 and TSIZ0 = 1. A2 is used to decode which of the two (32-bit) words is needed by the processor. If A2 = 0, then the upper 4 bytes of the data bus are marked "A". If A2 = 1, then the lower 4 bytes of the data bus are marked "A".

Similarly, for half-word accesses, TSIZ1 = 1, TSIZ0 = 0. A2–A1 are used to determine which of the four half-words is required by the processor. Finally, for byte accesses, TSIZ1 = 1, TSIZ0 = 1 and A2–A0 are decoded to determine which of the 8 bytes is required by the processor. Figure 11-20 illustrates how to decode A2–A0, TSIZ1–TSIZ0, and TBST to generate 8 byte strobe signals.

```
BS0 = (!A0 & !A1 & !A2
   # !A1 & !A2 & !TSIZ0
   # !A2 & !TSIZ1
   # !TSIZ0 & !TSIZ1
   # !~TBST);


BS1 = (A0 & !A1 & !A2
   # !A1 & !A2 & !TSIZ0
   # !A2 & !TSIZ1
   # !TSIZ0 & !TSIZ1
   # !~TBST);


BS2 = (!A0 & A1 & !A2 & TSIZ0
   # !A1 & !A2 & !TSIZ0
   # !A2 & !TSIZ1
   # !TSIZ0 & !TSIZ1
   # !~TBST);


BS3 = (A0 & A1 & !A2
   # A1 & !A2 & !TSIZ0
   # !A2 & !TSIZ1
   # !TSIZ0 & !TSIZ1
   # !~TBST);


BS4 = (!A0 & !A1 & A2
   # !A1 & A2 & !TSIZ0
   # A2 & !TSIZ1
   # !TSIZ0 & !TSIZ1
   # !~TBST);


BS5 = (A0 & !A1 & A2
   # !A1 & A2 & !TSIZ0
   # A2 & !TSIZ1
   # !TSIZ0 & !TSIZ1
   # !~TBST);


BS6 = (!A0 & A1 & A2 & TSIZ0
   # !A1 & A2 & !TSIZ0
   # A2 & !TSIZ1
   # !TSIZ0 & !TSIZ1
   # !~TBST);


BS7 = (A0 & A1 & A2
   # A1 & A2 & !TSIZ0
   # A2 & !TSIZ1
   # !TSIZ0 & !TSIZ1
   # !~TBST);
```

Figure 11-20. Byte Strobe Generation

Figure 11-21 shows the general form of the multiplexing between the external bus and an internal register. The four bytes shown in Figure 11-21 are connected through the internal data bus and data multiplexer to the external data bus. The data multiplexer establishes the necessary connections for different combinations of address and data sizes.

The multiplexer takes the eight bytes of the 64-bit bus and routes them to their required positions. For example, OP7 can be routed to D7–D0, as would be the case for a double-word transfer, or it can be routed to any other byte position in order to support a byte access.

11
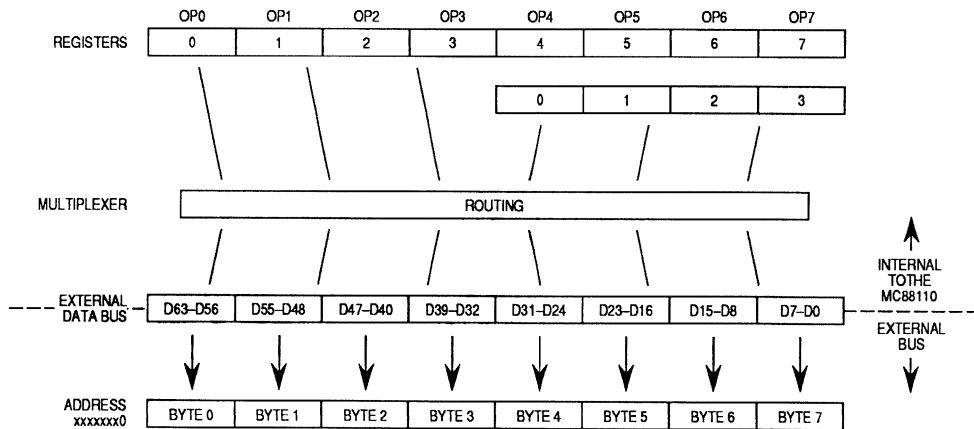
Figure 11-21. Data Multiplexing

11.5.3 Single-Beat Transactions

Accesses that occur directly on the external bus independently of the data cache (regardless of a cache hit) cause single-beat transactions to occur. These accesses include cache-inhibited accesses, invalidation cycles, **xmem** transactions, write transactions that occur in write-through mode, store-through accesses, table search transactions, and allocate load operations.

11.5.3.1 SINGLE-BEAT TRANSACTION TIMING EXAMPLE. Figure 11-22 shows the relative timing of the data transfer signals during a single-beat transaction. Before a single-beat transaction begins, the BIU arbitrates for the address bus, and the MC88110 becomes the address bus master.

As shown in Figure 11-22, the processor drives the address signals with the physical address of the access off the rising edge of clock 1 and at the same time asserts the appropriate attribute and control signals for the type of single-beat transaction being performed (see Table 11-15). The responding memory system can sample the address as early as the next rising clock edge (clock 2).

The MC88110 also asserts the transfer start ($\overline{\text{TS}}$) signal off the rising edge of clock 1 for one clock cycle. The memory system should then interpret the assertion of $\overline{\text{TS}}$ as a data bus request. Once the MC88110 becomes data bus master, either the MC88110 or the memory system places data on the data bus depending on the type of transaction (write or read).
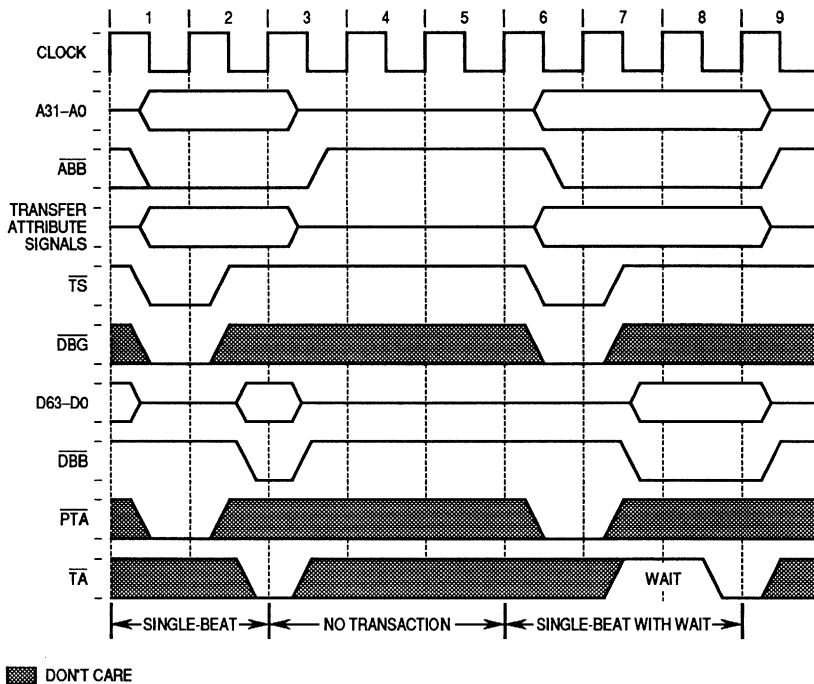
Figure 11-22. Single-Beat Transaction Timing Example

To indicate the status of the transaction to the processor, the memory system then either asserts or negates the $\overline{TA}$ signal. When the data is guaranteed to meet the appropriate setup and hold times with respect to the rising edge of the clock, the memory system should assert $\overline{TA}$ to terminate the transaction. In the fastest case, $\overline{TA}$ is asserted in the clock cycle after the address is sampled (clock 2 in Figure 11-22). If the data cannot be supplied (for reads) or latched (for writes) in time during the clock cycle after the address is sampled, $\overline{TA}$ must be explicitly negated until the appropriate setup and hold times are met.

While $\overline{TA}$ is negated, the processor waits, and the BIU continuously drives the address (and data for writes) on the address bus until $\overline{TA}$ is asserted. The memory system can insert as many wait cycles as necessary until the appropriate data setup and hold times are met. The fastest case and a one wait state case are both shown in Figure 11-22.

During the clock cycle after the assertion of $\overline{TA}$, the address lines are three-stated and (in this case) both $\overline{ABB}$ and $\overline{DBB}$ are negated.

The memory system should assert the transfer error acknowledge ($\overline{TEA}$) signal for a bus error, the transfer retry ($\overline{TRTRY}$) signal for a transfer retry, or the address retry ($\overline{ARTRY}$) signal for an address retry. For more information on $\overline{TA}$ and other termination signals, refer to **11.6 Termination of Bus Transactions**.

11.5.3.2 SINGLE-BEAT TRANSACTION TYPES.
Table 11-15 provides a list of the eight types of single-beat transactions and the state of the transfer attribute signals and some snoop control signals for each of these transactions. All single-beat transactions have similar timing characteristics; the differences between the transactions are determined by the transfer attribute signals that are asserted/negated.

Table 11-15. Single-Beat Transaction Transfer Attribute Signal States

Transaction	R/$\overline{\text{W}}$	$\overline{\text{LK}}$	$\overline{\text{CI}}$	$\overline{\text{WT}}$	UPA	$\overline{\text{TBST}}$	TSIZ1–TSIZ0	TC3–TC0	$\overline{\text{INV}}$	$\overline{\text{MC}}$	$\overline{\text{GBL}}$	$\overline{\text{CLINE}}$
Read	R	N	MMU	MMU	MMU	N	b,h,w,d	C/D,U/S	N	A	MMU	Invalid
Write	W	N	MMU	MMU	MMU	N	b,h,w,d	D,U/S	A	A	MMU	Invalid
Invalidate	W	N	N	N	MMU	N	b,h,w,d	D,U/S	A	N	A	Valid
xmem Read	R	A	MMU	MMU	MMU	N	b,w	D,U/S	A	A	MMU	Invalid
xmem Write	W	A	MMU	MMU	MMU	N	b,w	D,U/S	A	A	MMU	Invalid
Table Search	R	N	N	N	MMU	N	w	C/D,TSO	N	A	N	Invalid
Store-Through	W	N	MMU	A	MMU	N	b,h,w,d	D,U/S	A	A	MMU	Invalid
Allocate Load	R	N	MMU	MMU	MMU	N	h	TFA	A	N	MMU	Valid

Legend: A = Asserted
N = Negated
MMU = Value of bit in ATC entry or area descriptor
b = Byte
h = Half-Word
w = Word
d = Double Word
C = Code Access
D = Data Access
S = Supervisor
U = User Access
TSO = Table Search Operation
TFA = Touch, Flush, or Allocate Access

Since all transactions in Table 11-15 are single-beat, $\overline{\text{TBST}}$ is always negated. Note that during all types of transactions except for invalidate and allocate load, the memory cycle ($\overline{\text{MC}}$) signal is asserted. The $\overline{\text{MC}}$ signal is asserted when data must be transferred between the processor and an external device. Note that the $\overline{\text{INV}}$ signal is asserted for all write transactions, both portions of a **xmem** operation, and allocate load transactions. The $\overline{\text{INV}}$ signal is asserted to notify snooping processors to invalidate its corresponding cache line if necessary.

The following paragraphs describe each type of transaction in detail.

11.5.3.3 SINGLE-BEAT READ TRANSACTION. During single-beat read transactions, the MC88110 reads a byte, half-word, word, or double word from an external device.

To perform a single-beat read transaction, the MC88110 first arbitrates for mastership of the address bus. The MC88110 then asserts $\overline{TS}$, drives the address onto the address bus, and asserts or negates the appropriate transfer attribute signals (see Table 11-15) as described in Figure 11-23.

PROCESSOR   EXTERNAL DEVICE

ADDRESS DEVICE

1) SET R/$\overline{W}$ TO READ
2) DRIVE ADDRESS ON A31–A0
3) DRIVE SIZE ON TSIZ1–TSIZ0 (BYTE, HALF-WORD,
   WORD, OR DOUBLE WORD)
4) NEGATE $\overline{TBST}$
5) DRIVE TRANSFER ATTRIBUTE SIGNALS
   ($\overline{LK}$, $\overline{INV}$, $\overline{MC}$, $\overline{WT}$, $\overline{CI}$, UPA1–UPA0, TC3–TC0, $\overline{GBL}$)
6) ASSERT TRANSFER START ($\overline{TS}$) FOR ONE CLOCK

PRESENT DATA

1) DECODE ADDRESS
2) ASSERT $\overline{DBG}$ AND $\overline{PTA}$
3) PLACE DATA ON APPROPRIATE BYTES OF
   D63–D0
4) ASSERT TRANSFER ACKNOWLEDGE ($\overline{TA}$)

ACQUIRE DATA

1) LATCH DATA

TERMINATE CYCLE

1) REMOVE DATA FROM D63–D0
2) NEGATE $\overline{TA}$

START NEXT CYCLE

Figure 11-23. Single-Beat Read Transaction Flow

At the beginning of each transaction, $\overline{TS}$ is asserted for one clock cycle. The external arbiter should interpret the assertion of $\overline{TS}$ as a data bus request. Once the MC88110 becomes the data bus master, the memory system should supply the requested data on the appropriate D63–D0 signals within the required setup and hold times with respect to the rising edge of the clock, while asserting $\overline{TA}$. If the memory system is unable to supply the data within the appropriate setup and hold times, the memory system should insert wait states by negating $\overline{TA}$ until the data is available.

Figure 11-24 shows the relative timing for single-beat read transactions with and without a wait state.

11
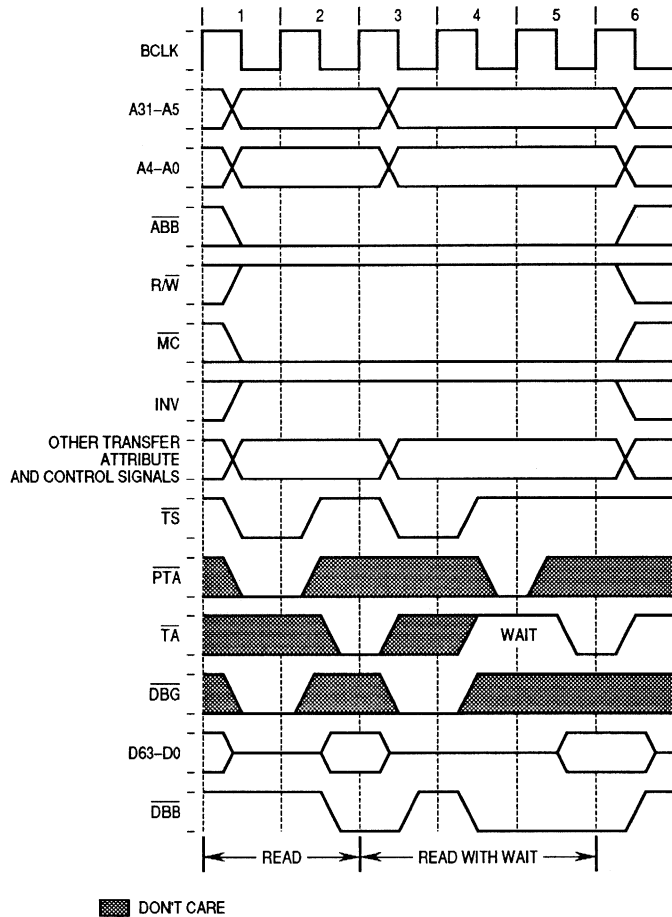
Figure 11-24. Single-Beat Read Transaction Timing

11.5.3.4 SINGLE-BEAT WRITE TRANSACTION. During single-beat write transactions, the MC88110 transfers a byte, half-word, word, or double word to an external device.

To perform a single-beat write transaction, the MC88110 first becomes the address bus master. The MC88110 then asserts TS, drives the address of the data onto the address bus, and asserts or negates the appropriate attribute and control signals (see Table 11-15) as described in Figure 11-25. All writes from the MC88110 cause the invalidate (INV) signal to be asserted so that snooping processors can invalidate their cached versions of the data.

EXTERNAL DEVICE

ADDRESS DEVICE
1) DRIVE R/W̄ AS WRITE
2) DRIVE ADDRESS ON A31–A0
3) DRIVE SIZE ON TSIZ1–TSIZ0 (BYTE, HALF-WORD, WORD, OR DOUBLE WORD)
4) NEGATE T̄B̄S̄T̄
5) DRIVE TRANSFER ATTRIBUTE SIGNALS (L̄K̄, ĪN̄V̄, M̄C̄, W̄T̄, C̄Ī, ŪP̄Ā1̄–ŪP̄Ā0̄, TC3–TC0, ḠB̄L̄)
6) ASSERT TRANSFER START (T̄S̄) FOR ONE CLOCK

LATCH ADDRESS
1) LATCH AND DECODE ADDRESS
2) ASSERT DATA BUS GRANT (D̄B̄Ḡ) AND P̄T̄Ā

PRESENT DATA
1) DRIVE DATA ON D63–D0

TERMINATE CYCLE
1) LATCH DATA FROM D63–D0
2) ASSERT T̄Ā

START NEXT CYCLE

Figure 11-25. Single-Beat Write Transaction Flow

At the beginning of each transaction, T̄S̄ is asserted for one clock cycle. The external arbiter should interpret the assertion of T̄S̄ as a data bus request. Once the MC88110 becomes the data bus master, the MC88110 immediately drives the data onto the data bus. The memory system should latch the data while asserting T̄Ā. If the memory system is unable to latch the data, it should insert wait states by negating T̄Ā until the data is latched.

Figure 11-26 shows the relative timing for single-beat write transactions with and without a wait state.

11

DON'T CARE
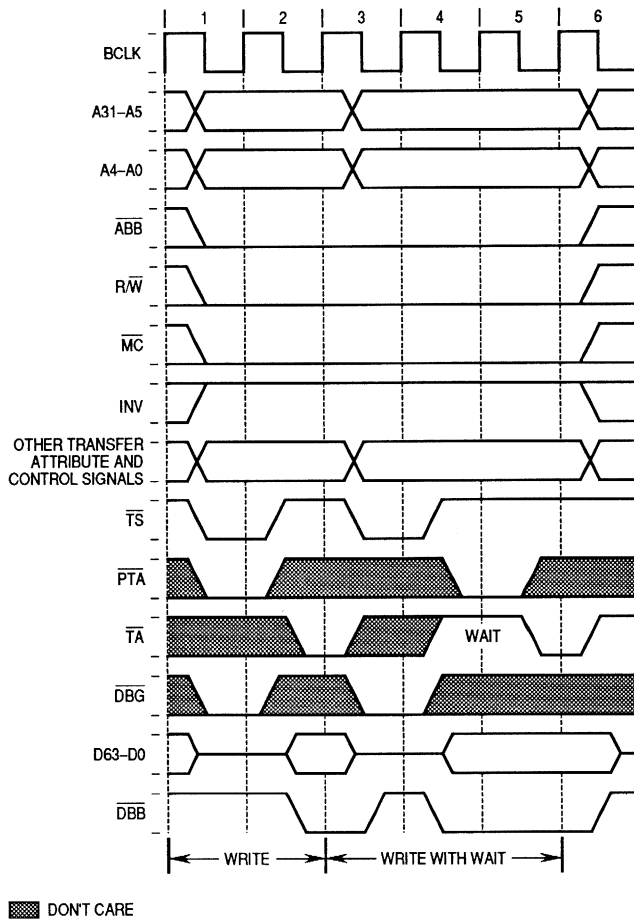
Figure 11-26. Single-Beat Write Transaction Timing

11.5.3.5 INVALIDATE TRANSACTION. Invalidate transactions are single-beat transactions used by the MC88110 to maintain cache coherency among multiple MC88110 processors. Invalidate transactions broadcast to snooping processors that a line in memory will be modified; thus, snooping processors should invalidate their cached versions of the line. See **11.3.3 Data Cache Coherency** for more information on snooping and cache coherency.

An invalidate transaction is an address-only transaction; although valid data is driven on the data bus, no data is transferred. Invalidate transactions use the protocol defined for single-beat write transactions. The only difference between an invalidate transaction and a normal single-beat write transaction is that for an invalidate transaction, $\overline{\text{MC}}$ is negated since no data must be transferred. For both invalidate and normal single-beat write transactions, R/$\overline{\text{W}}$ is low, signalling a write, and $\overline{\text{INV}}$ is asserted to notify snooping processors to invalidate their cached versions of the line.

Even though no data is transferred during an invalidate transaction, the MC88110 must still request and be granted the data bus. Unless a transaction is abnormally terminated with an address retry (see **11.7.3 Address Retry Transaction Termination**), the transaction cannot be completed until the arbiter asserts DBG.

Figure 11-27 shows the timing diagram for a read followed by a write followed by an invalidate transaction. The three types of transactions are differentiated by the state of the R/W, MC, and INV signals.
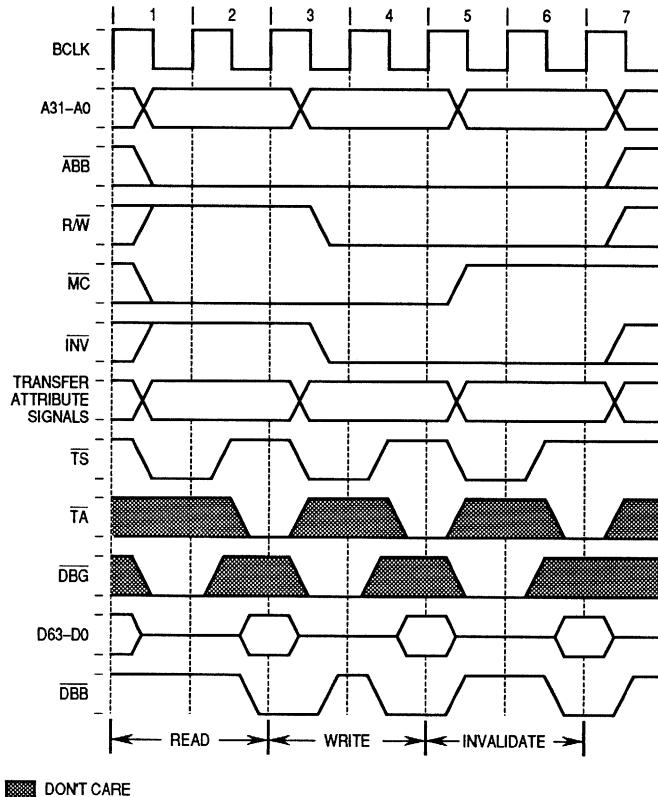
Figure 11-27. Single-Beat Read, Single-Beat Write, and Invalidate Transactions Timing

11.5.3.6 xmem TRANSACTION. The **xmem** instruction is a multiprocessor synchronization instruction that uses a single-beat read and a single-beat write transaction to exchange the contents of a general register with that of an addressed memory location. The **xmem** instruction is normally used to implement semaphores or resource locks in multiprocessor or multitasking systems.

The **xmem** instruction is effectively a locked combination of a load and store instruction. The MC88110 implements the **xmem** instruction in one of two ways based on the value of the XMEM bit in the DCTL. If the XMEM bit is clear (the default case), the **xmem** instruction causes a single-beat read followed by a single-beat write transaction; otherwise, the **xmem** instruction causes a single-beat write followed by a single-beat read transaction. If the **xmem** instruction causes a cache hit to a modified line, then a copyback is performed before the two single-beat transactions.

During the execution of the **xmem** transaction, the bus lock signal ($\overline{\text{LK}}$) is asserted for both the read and write portions of the **xmem** transaction. The $\overline{\text{LK}}$ signal is asserted to indicate that the bus arbitration circuitry should not allow another bus master to alter the data being accessed by the **xmem** transaction between the read and the write. One way that the arbitration circuit can ensure this is by locking the bus throughout the read and write portions of the **xmem** transaction.

The $\overline{\text{BR}}$ signal operates slightly differently for **xmem** operations than for all other transactions. For the first transaction in the **xmem** operation, $\overline{\text{BR}}$ remains asserted while $\overline{\text{TS}}$ is asserted. In all other cases, including the second transaction in the **xmem** operation, the $\overline{\text{BR}}$ signal is negated when $\overline{\text{TS}}$ is asserted. The arbitration circuit can use this feature to easily lock the bus between the two transactions by not negating $\overline{\text{BG}}$ (once it is asserted) until the MC88110 negates $\overline{\text{BR}}$. Another advantage to keeping $\overline{\text{BG}}$ asserted throughout the two transactions is that the transfer attribute signals will remain valid.

Figure 11-28 shows the timing of a read followed by a write **xmem** operation for the unparked case, and Figure 11-29 shows the timing for the parked case. Note that in both cases $\overline{\text{BR}}$ is asserted until the MC88110 initiates the write portion of the transaction. The transfer attribute and $\overline{\text{ABB}}$ signals, however, are asserted for the duration of the locked read/write sequence only in the parked case. **xmem**

The $\overline{\text{INV}}$ signal is also asserted for both of the **xmem** transactions to signal to snooping processors that a write to memory will occur, which may require them to invalidate a line in their cache.

11
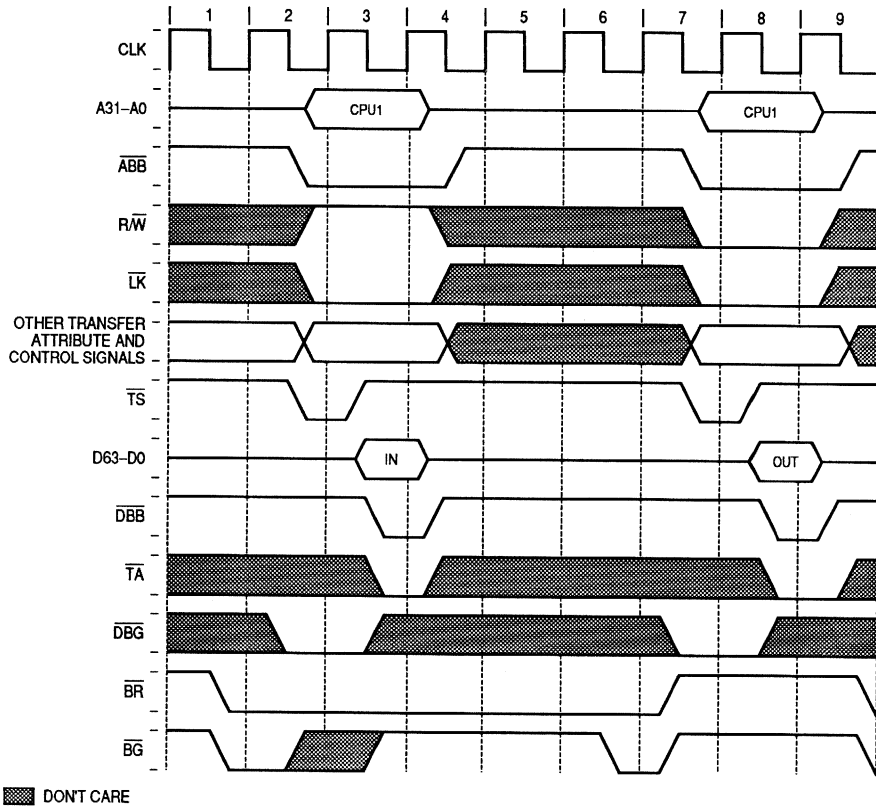
Figure 11-28. xmem Transaction Timing—Unparked Case
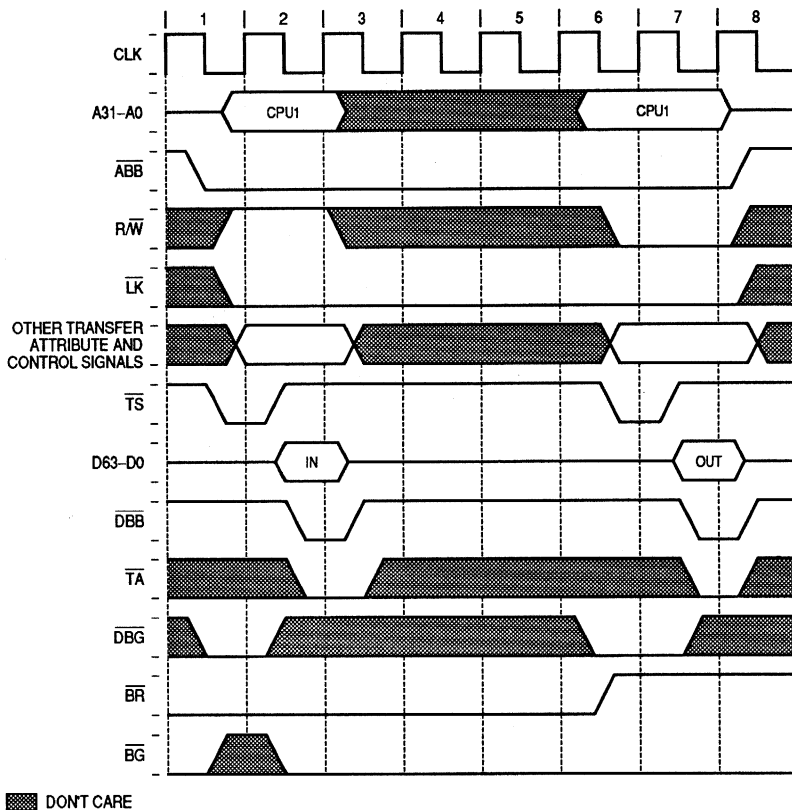
Figure 11-29. xmem Transaction Timing—Parked Case

11

11.5.3.7 TABLE SEARCH TRANSACTIONS. A table search operation is a series of single-beat transactions performed by the MC88110 when a logical address misses in the block address translation cache (BATC) and page address translation cache (PATC) with address translation enabled (MMU enabled) (see **Section 8 Memory Management Units** for more detailed information on the causes of table search operations). During a table search operation, the physical address for the missed logical address is obtained by progressing through the memory mapping tables.

The timing for a table search transaction is identical to the timing for a single-beat read transaction; however, the $\overline{CI}$, $\overline{WT}$, and $\overline{GBL}$ attribute signals are never asserted. (For a single-beat read, these signals can be asserted.) Also, TSIZ1–TSIZ0 always indicate a word access (TSIZ1 = 0 and TSIZ0 = 1) for a table search transaction. For the detailed timing for table search operations, see **11.8 MMU Transactions**.

11.5.3.8 STORE-THROUGH TRANSACTION. The store-through option is a feature that unconditionally causes the store instructions to write-through the on-chip data cache directly to memory. If a store-through access hits in the cache, the data is written both to memory and the cache, but the state of the cache line is not changed. When a store-through access misses in the data cache, no line is allocated in the cache, and the access simply writes directly to memory, bypassing the cache completely. The store-through operation is identical to a cache access in write-through mode. See **Section 6 Instruction and Data Caches** for more information on the store-through feature of the MC88110.

The store-through option is specified by a **.wt** (for write-through) extension on any triadic register addressing form of the store instruction. The timing for a store-through transaction is the same as that for a single-beat write transaction; however, the $\overline{WT}$ signal is always asserted.

11.5.3.9 ALLOCATE LOAD TRANSACTION. The allocate load option is a cache control feature that allows the user to allocate a line in the data cache for a series of subsequent store operations while avoiding the normal line fill from memory. The allocate load option can improve performance by eliminating the overhead of reading a new line from memory that is going to be overwritten. The allocate load option is specified as a half-word load to **r0**. See **Section 6 Instruction And Data Caches** for more detailed information on the allocate load option.

The allocate load option allocates a line in the cache on a cache miss (as any normal load does), but only performs a single-beat bus transaction rather than a complete line fill burst transaction. For an allocate load transaction, the $\overline{INV}$ signal is asserted and the $\overline{MC}$ signal is negated. This timing is the same as that of an invalidate transaction; however, the transfer code signals indicate that it is a touch, flush, or allocate load transaction. If an allocate load is used to access cache inhibited memory, the single-beat bus transaction is still performed but no new line is allocated in the cache. In this case, the $\overline{INV}$ signal is asserted, the $\overline{MC}$ signal is negated, and the $\overline{CI}$ signal is asserted. The memory system does not have to provide valid parity on the BP7–BP0 signals for this transaction.

11

11.5.4 Burst Transactions

Burst transactions perform the transfer of four double words between the processor and an external device. Cache maintenance operations that require four double words to be read from or written to memory (e.g., cache line fill and copyback operations) are performed as burst transactions.

For most burst transactions, the MC88110 uses a critical-word-first convention to determine the double word in the cache line that is accessed first. The critical-word-first convention means that the cache line fill (or copyback) operation always begins with the evenly aligned double word containing the missed word (i.e., critical-word-first), followed by the subsequent double word(s) in the line, if any. If the double word containing the missed instruction (or data) does not correspond to the first double word in the cache line, the fill operation wraps around and then fills the double word(s) at the beginning of the line.

Figure 11-30 illustrates an example of the critical-word-first operation. The example shows the result of a byte load from the address $0B. Note that the full byte address is driven on the address bus for the first two clock cycles, even though it is a burst transaction and the full evenly aligned double word must be transferred by the memory system to the processor. Also, note that in the subsequent clock cycles, address bits A2–A0 remain the same, and address bits A4–A3 are changed to reflect the address of the double word that must be transferred.
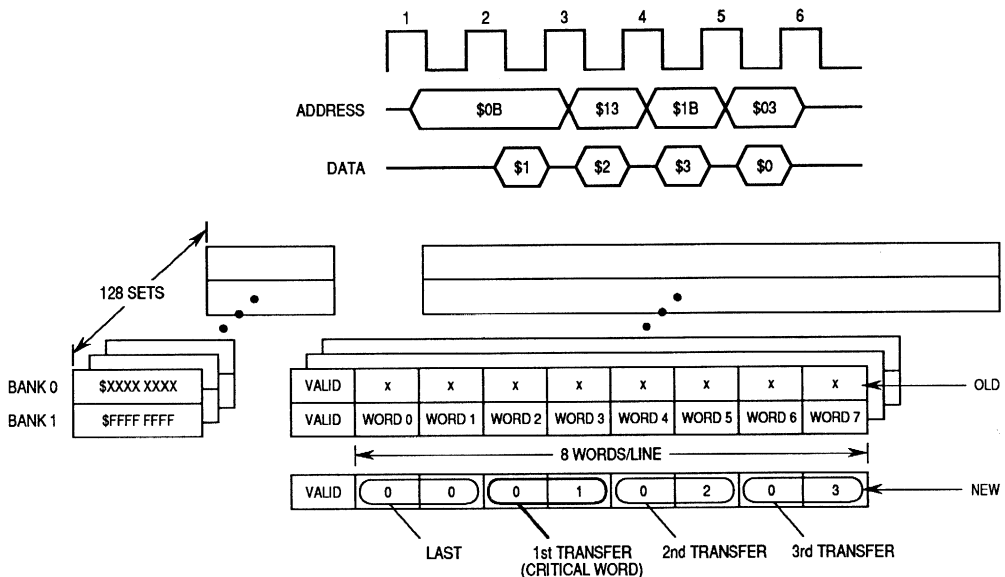
Figure 11-30. Critical-Word-First Operation Example

If the address bus is not released by the assertion of $\overline{\text{AACK}}$, the MC88110 drives the address of the missed word and then steps through the addresses required for the remainder of the cache line fill (or copyback) operation. If $\overline{\text{AACK}}$ is asserted, the MC88110 releases the address bus and the memory system is responsible for incrementing the remaining addresses for the remainder of the cache line fill (or copyback) operation. For the flush copyback operation, the transfer of data begins with the first double word of the line. The following paragraphs describe the timing of the MC88110 signals for burst transactions and the operation of the various types of burst transactions.

11.5.4.1 BURST TRANSACTION TIMING EXAMPLES. Figure 11-31 shows the relative timing of the data transfer signals during a burst transaction. Before a burst transaction begins, the BIU arbitrates for the address bus, and the MC88110 becomes the address bus master.

■■■ DON'T CARE
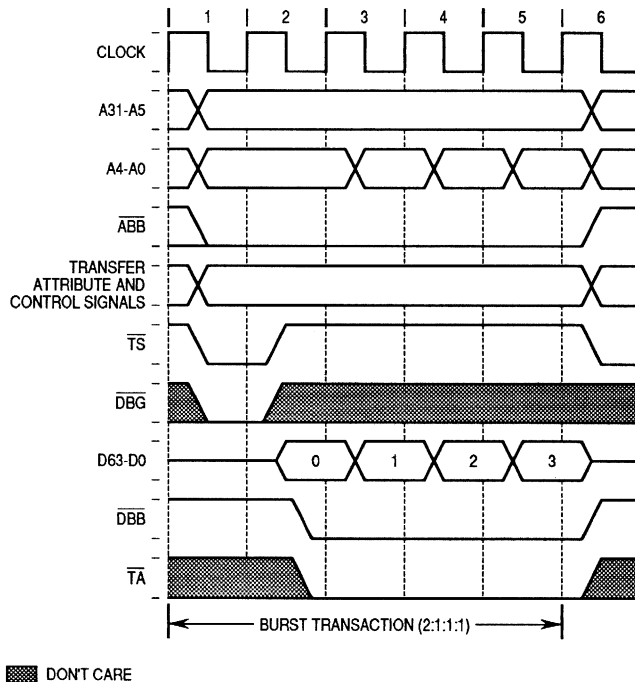
Figure 11-31. General Burst Transaction Timing

11

As shown in Figure 11-31, the processor then drives the address signals with the physical address of the access off the rising edge of clock 1 and at the same time asserts the appropriate attribute and control signals for the type of burst transaction being performed.

At the same time that the attribute and control signals are asserted, the transfer start ($\overline{\text{TS}}$) signal is also asserted for one clock cycle. The external arbiter should then interpret the assertion of $\overline{\text{TS}}$ as a data bus request. Once the MC88110 becomes the data bus master, either the MC88110 or the memory system places the instruction/data for the first beat of the burst on the data bus. The next three beats of the burst occur during subsequent clock cycles.

To indicate the status of each of the four beats of the burst transaction to the processor, the memory system then either asserts or negates the $\overline{\text{TA}}$ signal. When the double-word data is guaranteed to meet the appropriate setup and hold times, the memory system should assert $\overline{\text{TA}}$ to terminate the beat. At this time, either the address is incremented to be the address for the next beat of the burst, or, if all four beats have completed successfully, the burst transaction is terminated.

The fastest case burst transaction occurs when no wait cycles are inserted by the memory system. In this case (as shown in Figure 11-31), $\overline{\text{TA}}$ is asserted during the first beat of the burst transaction and remains asserted during all four beats of the burst. In this case specifically, the memory system places the first aligned double word on the data bus during clock 2 and asserts $\overline{\text{TA}}$. During each of the following three clock cycles, the address is incremented to reflect the address of the appropriate double word. The memory system continues to supply the MC88110 with the appropriate double words on the data bus. The address, data, and control signals are three-stated in clock 6, and $\overline{\text{TA}}$ is negated to signal the end of the transaction.

If the data/instruction cannot be supplied in time during the clock cycle after the address is sampled, $\overline{\text{TA}}$ should be explicitly negated until the setup and hold times are met. While $\overline{\text{TA}}$ is negated, the processor waits and the BIU continuously drives the address on the address bus until $\overline{\text{TA}}$ is asserted. The memory system can insert as many wait cycles as necessary until the setup and hold times are met for each beat. For more information on $\overline{\text{TA}}$ and the termination of bus transactions, refer to **11.6 Termination of Bus Transactions**.

An example of a burst transaction with wait cycles is shown in Figure 11-32. During clock 1, the full 32-bit address of the requested data is driven on the address bus, but the data is not immediately available. Thus, the memory system negates the $\overline{\text{TA}}$ signal and the processor interprets this as a wait response. In this example, the memory system inserts two wait cycles before the data is available. During clock 4, the first double word is driven on the data bus and $\overline{\text{TA}}$ is asserted. During each of the following three clock cycles, the address is incremented to reflect the address of the appropriate double word. The memory system continues to supply the MC88110 with the appropriate double words on the data bus. In clock 9, the transaction is terminated and $\overline{\text{TA}}$ is negated.

11
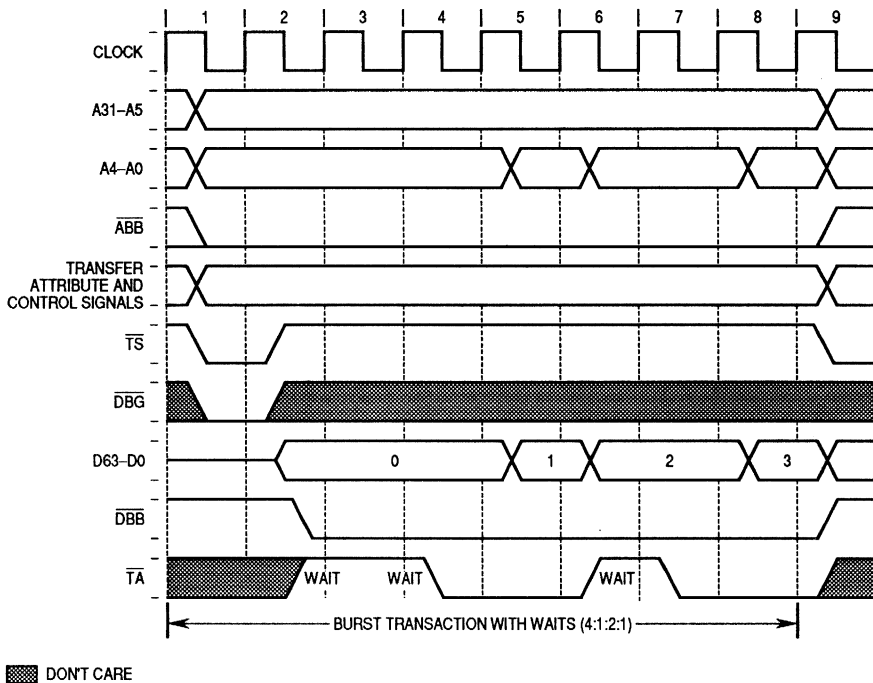
Figure 11-32. Burst Transaction with Wait Cycles

The memory system should assert the transfer error acknowledge ($\overline{\text{TEA}}$) signal for a bus error, the transfer retry ($\overline{\text{TRTRY}}$) signal for a transfer retry, or the address retry ($\overline{\text{ARTRY}}$) signal for an address retry.

If a bus error is encountered during the access to the critical word, then a data or instruction access exception occurs and the cache line is not updated. If a bus error is encountered at any time during a read-with-intent-to-modify cycle, then a data access exception occurs. For more information on MC88110 exceptions and exception processing, see **Section 7 Exceptions**. If a bus error occurs during any other beat of the transaction, then the corresponding cache line is marked as invalid. If no bus error is encountered, the line is marked as valid when the transaction completes.

11

11.5.4.2 BURST TRANSACTION TYPES. There are eight types of burst transactions performed by the MC88110 bus. Table 11-16 lists the types of burst transactions and the double word in the cache line that is transferred first for each type of transaction.

**Table 11-16. Burst Transaction Types
and First Double Word Transferred**

Type of Burst Transaction	Double Word Transferred First
Instruction Cache Read Miss Line Fill	Evenly Aligned Double Word Containing the Critical Word
Data Cache Read Miss Line Fill	Evenly Aligned Double Word Containing the Critical Word
Data Cache Write Miss Line Fill—Read-with Intent-to-Modify Cycle	Evenly Aligned Double Word Containing the Critical Word
Touch Load	Evenly Aligned Double Word Containing the Critical Word
Replacement Copyback	Evenly Aligned Double Word Containing the Critical Word
Snoop Copyback	Evenly Aligned Double Word Containing the Critical Word
Flush Copyback	First Double Word in the Cache Line
Flush Load	Evenly Aligned Double Word Containing the Critical Word

Table 11-17 lists the eight types of burst transactions and the state of the transfer attribute signals for each type of burst transaction.

11

Table 11-17. Burst Transaction Transfer Attribute Signal States

Transaction	R/$\overline{W}$	$\overline{LK}$	$\overline{CI}$	$\overline{WT}$	UPA	$\overline{TBST}$	TSIZ1–TSIZ0	TC3–TC0	$\overline{INV}$	$\overline{MC}$	GBL	$\overline{CLINE}$
Read Transactions												
Instruction Cache Read Miss Line Fill	R	N	N	MMU	MMU	A	d	C, U/S	N	A	MMU	Valid
Data Cache Read Miss Line Fill	R	N	N	MMU	MMU	A	b,h,w,d	D, U/S	N	A	MMU	Valid
Data Cache Read-with Intent-to-Modify Cycle	R	N	N	N	MMU	A	b,h,w,d	D, U/S	A	A	MMU	Valid
Touch Load*	R	N	MMU	MMU	MMU	A	b	TFA	N	A	MMU	Valid
Write Transactions												
Replacement Copyback	W	N	N	N	N	A	d	D, S	A	A	N	Valid
Snoop Copyback	W	N	N	N	N	A	d	SCB	A	A	N	Valid
Flush Copyback	W	N	N	N	N	A	d	D, S	A	A	N	Valid
Flush Load	W	N	N	N	N	A	w	TFA	A	A	N	Valid

* If the CI bit is set in the ATC entry, then the flush load causes a single-beat access

Legend: A = Asserted
N = Negated
MMU = Value of bit in ATC entry or area descriptor
b = Byte
h = Half-Word
w = Word
d = Double Word
C = Code Access
D = Data Access
S = Supervisor
U = User Access
SCB = Snoop Copyback Operation
TFA = Touch, Flush, or Allocate Access

11.5.4.3 BURST READ TRANSACTIONS. During a burst read transaction, the MC88110 reads four double-words from memory to fill a cache line. As instructions/data are latched from the bus, they are written into the appropriate cache and simultaneously streamed directly to the instruction/data unit, at which time any pending data dependencies are resolved. All subsequent instructions/data read from the bus are also streamed to the instruction/data unit in parallel with the reading of the remaining words on the bus.

11

Figure 11-33 shows a general flow diagram for a burst read transaction (see Figures 11-31 and 11-32 for illustrations of the relative timing for some burst read transactions).

PROCESSOR   EXTERNAL DEVICE

ADDRESS DEVICE

1) DRIVE R/W̄ AS READ
2) DRIVE ADDRESS ON A31–A0
3) DRIVE SIZE ON TSIZ1–TSIZ0 (BYTE, HALF-WORD,
   WORD, OR DOUBLE WORD)
4) ASSERT T̄B̄S̄T̄
5) DRIVE TRANSFER ATTRIBUTE SIGNALS
   (L̄K̄, ĪNV̄, M̄C̄, W̄T̄, C̄Ī, ŪPA1–ŪPA0, TC3–TC0, GBL)
6) ASSERT TRANSFER START (T̄S̄) FOR ONE CLOCK

LATCH ADDRESS

1) DECODE ADDRESS
2) ASSERT D̄B̄Ḡ AND P̄T̄Ā
2) PLACE DATA ON D63–D0
3) ASSERT TRANSFER ACKNOWLEDGE (T̄Ā)

ACQUIRE DATA

1) LATCH DATA
2) INCREMENT ADDRESS BITS A4–A3 (IF STILL
   ADDRESS BUS MASTER)

TERMINATE CYCLE

1) REMOVE DATA FROM D63–D0
2) NEGATE T̄Ā (IF NECESSARY)

START NEXT CYCLE

WHEN 4 DOUBLE WORDS   UNTIL 4 DOUBLE WORDS
    TRANSFERED   TRANSFERED

Figure 11-33. Burst Read (Cache Line Fill) Transaction Flow

There are three types of burst read transactions: cache line fill operations, touch load operations and read-with-intent-to-modify cycles. The burst read operations are described in the following paragraphs.

11.5.4.3.1 Cache Line Fill Operation—Read Miss. A cache miss occurs when caching is enabled and the instruction/data required by the processor is not resident in the appropriate cache. A processor read access that misses in the cache causes a bus transaction to occur. This operation is called a cache line fill operation.

Several conditions contribute to the actions taken by the processor for a read cache miss. For this section, it is assumed that the transaction results in an ATC hit (or address translations are disabled) and no table search is necessary. See **11.8 MMU Transactions** for a detailed description of the bus transactions that occur when an ATC miss occurs and a hardware table search operation is performed.

If the cache line that is selected for replacement is marked as modified, the cache line is flushed before the cache line fill operation occurs. This operation is called a replacement copyback. If the cache line selected for replacement is marked as unmodified or invalid, no memory update is necessary and the cache line fill operation proceeds as a burst read transaction. The replacement copyback operation is described in **11.5.4.4.1 Replacement Copyback Transaction**.

Figure 11-31 shows the timing for a cache line fill operation. The full 32-bit address of the critical double word and the appropriate control and transfer attribute signals are asserted by the processor off the rising edge of clock 1. Because this is a cache line fill operation, the $\overline{\text{INV}}$ signal is negated, R/$\overline{\text{W}}$ is driven high, $\overline{\text{CI}}$ is negated, $\overline{\text{MC}}$ is asserted, and $\overline{\text{LK}}$ is negated (as shown in Table 11-17).

11.5.4.3.2 Touch Load Burst Read Transaction. The touch load option is a user-mode cache control feature that allows data to be loaded into the data cache under user program control. Normally, data is brought into the cache only when it is needed. This can lead to instruction execution stalls due to dependencies on data that must be read from main memory. In many cases, however, the need for data can be predicted. By forcing certain data be read into the cache ahead of its actual use, the latency of the memory system can be overlapped with useful work, and stalls due to long latency cache misses can be minimized. The touch load option is specified as a byte load instruction to **r0**. See **Section 6 Instruction and Data Caches** for more detailed information on the use of the touch load option.

The timing for touch load read transactions is the same as that for a burst read operation. However, if the CI bit in the ATC is set for a touch load operation, then a single-beat read transaction is performed instead of a burst read. For touch load operations, the $\overline{\text{INV}}$ signal is negated, R/$\overline{\text{W}}$ is driven high, $\overline{\text{MC}}$ is asserted, and $\overline{\text{LK}}$ is negated. The $\overline{\text{CI}}$ and $\overline{\text{WT}}$ signals reflect the value of the corresponding bits in the ATC entry in the appropriate MMU for the respective signal (see Table 11-17).

11.5.4.3.3 Read-with-Intent-to-Modify Burst Transaction. A read-with-intent-to-modify transaction is caused by a write access that misses in the data cache in write-back mode. A read-with-intent-to-modify transaction operates like a burst read transaction for a cache line fill but has the side effect of broadcasting to other processors on the bus that the cache line being read will be modified; thus, the other processors should invalidate any resident local copy of the cache line.

To notify the other processors on the bus that the cache line being read will be modified, the $\overline{\text{INV}}$ signal is asserted. Also, like a burst read transactions for cache line fills, R/$\overline{\text{W}}$ is driven high, $\overline{\text{CI}}$ is negated, $\overline{\text{WT}}$ is negated, $\overline{\text{MC}}$ is asserted, and $\overline{\text{LK}}$ is negated.

11

11.5.4.4 BURST WRITE TRANSACTIONS. During a burst write transaction, the MC88110 transfers four double-words from a data cache line to memory. Figure 11-34 shows a general flow diagram for a burst write transaction. The timing for the burst writes is shown in Figure 11-31.

```
            PROCESSOR                                    EXTERNAL DEVICE

      ┌─────────────────────────────┐
      │       ADDRESS DEVICE         │
      ├─────────────────────────────┤
      │ 1) DRIVE R/W̄ AS WRITE        │
      │ 2) DRIVE ADDRESS ON A31–A0   │
      │ 3) DRIVE SIZE ON TSIZ1–TSIZ0 │
      │    (BYTE, HALF-WORD,         │
      │    WORD, OR DOUBLE WORD)     │
      │ 4) ASSERT T̄B̄S̄T̄                │
      │ 5) DRIVE TRANSFER ATTRIBUTE  │       ┌─────────────────────────────┐
      │    SIGNALS                   │       │        LATCH ADDRESS         │
      │    (L̄K̄, ĪN̄V̄, M̄C̄, W̄T̄, C̄Ī,    │──────▶├─────────────────────────────┤
      │    ŪP̄Ā1̄–ŪP̄Ā0̄, TC3–TC0, ḠB̄L̄)  │       │ 1) LATCH AND DECODE ADDRESS, │
      │ 6) ASSERT TRANSFER START     │       │    TSIZ1–TSIZ0, AND TRANSFER │
      │    (T̄S̄) FOR ONE CLOCK        │       │    ATTRIBUTE SIGNALS         │
      └─────────────────────────────┘       │ 2) ASSERT D̄B̄Ḡ AND P̄T̄Ā         │
                                            └─────────────────────────────┘
      ┌─────────────────────────────┐              │
      │        PRESENT DATA          │◀─────────────┘
      ├─────────────────────────────┤
      │ 1) DRIVE DATA ON D63–D0      │
      │ 2) INCREMENT ADDRESS BITS    │       ┌─────────────────────────────┐
      │    A4–A3 (IF STILL           │       │        TERMINATE CYCLE       │
      │    ADDRESS BUS MASTER)       │──────▶├─────────────────────────────┤
      └─────────────────────────────┘       │ 1) LATCH DATA FROM D63–D0    │
                                            │ 2) ASSERT T̄Ā                  │
                                            └─────────────────────────────┘
      ┌─────────────────────────────┐
      │       START NEXT CYCLE       │◀─────
      └─────────────────────────────┘      WHEN 4 DOUBLE WORDS    UNTIL 4 DOUBLE WORDS
                                              TRANSFERED            TRANSFERED
```

Figure 11-34. Burst Write Transaction Flow

Before a burst write transaction is performed, the BIU arbitrates for mastership of the address bus. When the MC88110 becomes the address bus master the burst write transaction begins. The MC88110 drives the physical address of the access onto the address bus and asserts/negates the appropriate control and attribute signals (see Table 11-17) (e.g., the M̄C̄ signal is asserted to indicate that the access transfers data and R/W̄ is driven high). All write transactions from the MC88110 cause the ĪN̄V̄ signal to be asserted so that snooping processors can invalidate resident copies of the cache line. The memory system decodes the address on the next rising clock edge after the address is driven.

At the same time that the attribute and control signals are asserted, the T̄S̄ signal is asserted by the MC88110 for one clock cycle. The arbiter should interpret the assertion of T̄S̄ as a data bus request. Once the MC88110 becomes data bus master, the MC88110 immediately drives the data on the data bus. The memory system should latch the data and then assert T̄Ā. If the memory system is unable to latch the data within the appropriate setup and hold times, the memory system should insert wait cycles by negating T̄Ā until the data is latched.

11

Once $\overline{TA}$ has been asserted for the first beat of the burst write, the MC88110 increments address lines A4–A3 to reflect the double-word address needed for the second beat of the burst. Also, during that clock cycle, the MC88110 drives the data corresponding to the new address, and, if possible, the memory system latches the data while asserting $\overline{TA}$. Again, if the memory system is unable to latch the data within the setup and hold times, it should insert wait cycles by negating $\overline{TA}$. The process described in this paragraph is repeated for the third and fourth beats of the burst write transaction. When all four beats have completed successfully, the memory system negates $\overline{TA}$.

There are four types of burst write transactions: replacement copyback operations, snoop copyback operations, flush copyback operations, and flush load operations. A copyback operation is the process of writing a modified cache line out to memory so that memory is updated. All of the four conditions that cause burst write transactions are described in the following paragraphs.

The MC88110 asserts the same transfer attribute and control signals, except for the transfer code (TC3–TC0) signals, for all of the burst write transactions. The particular burst write transaction can be determined by decoding the transfer code (TC3–TC0) signals (see Table 11-7). Note that the timing for TC3–TC0 coincides with the timing for the address signals.

11.5.4.4.1 Replacement Copyback Transaction. When a data cache miss which requires a cache line fill occurs and the corresponding cache set has two valid entries, the cache access algorithm selects one of the two lines in the corresponding cache set for replacement. The MC88110 checks the state of the line to be replaced, and if the line is modified, then the line is copied back to memory. This copyback operation is referenced as a replacement copyback.

The timing for the replacement copyback is the same as for the burst write; however, the transfer code signals always indicate that a supervisor data access is in progress.

11.5.4.4.2 Snoop Copyback Transaction. The MC88110 uses a bus snooping protocol to maintain cache coherency in systems where more than one processor is allowed to access shared memory. When a snooping MC88110 has a cache hit during a global write or global read-with-intent-to-modify transaction, the snooping MC88110 determines if the cache line is modified. If the line is modified, the line must be copied back to memory before the MC88110 performing the global access can complete its transaction. This copyback operation is referenced as a snoop copyback. For more information on snooping transactions, see **11.7 Data Cache Coherency Timing Considerations**.

The timing for the snoop copyback is the same as for the burst write; however, the transfer code signals indicate that a snoop copyback access is in progress.

11

11.5.4.4.3 Flush Copyback Transaction. The MC88110 has a supervisor mode cache control feature that causes either all modified lines or any individual modified line in the data cache to be transferred out to memory and causes the transferred line(s) to be marked as unmodified. Each line transferred to memory by this operation is transferred by way of a burst write transaction called a flush copyback. See **Section 6 Instruction and Data Caches** for more information about flushing data cache entries to memory.

The timing for the flush copyback is the same as for the burst write; however, the transfer code signals indicate that a supervisor data access is in progress.

11.5.4.4.4 Flush Load Transaction. The flush load option is a cache control feature that allows the user to force a modified (dirty) cache line to be written to memory. Normally, modified cache lines are copied back to memory only as a side effect of needing to allocate a new cache line. However, it is sometimes appropriate to be able to flush data in the cache in order to immediately update the memory image. For example, the user may store several data words to memory that are filtered by the cache and never actually update memory. In this case, the flush load option can be used to flush the data words from the cache to memory. See **Section 6 Instruction and Data Caches** for more detailed information on the flush load option.

The timing for the flush copyback is the same as for the burst write; however, the transfer code signals indicate that a touch, flush, or allocate load is in progress.

11.5.5 Back-to-Back Transfer Timing

Table 11-18 shows the number of clock cycles between the assertion of $\overline{\text{TA}}$ or $\overline{\text{TEA}}$ for one transaction and the assertion of $\overline{\text{TS}}$ for the second transaction (assuming the MC88110 is parked). The last column shows the number of clock cycles between the assertion of $\overline{\text{ARTRY}}$ or $\overline{\text{TRTRY}}$ for a transaction and the assertion of $\overline{\text{TS}}$ for the retried transaction. Note that the burst write transactions include the flush and replacement copybacks. Also, there are no dead cycles between a replacement copyback operation and the burst read which caused it.

11

Table 11-18. Back-to-Back Transfer Timing

First Transaction/ Second Transaction	Table Search	Single-Beat Read	Single-Beat Write	Burst Read	Burst Write	Snoop Copyback	Any Instruction Access	Retry
	2	3	3	3	3	2	0	2
Single-Beat Read	4	2	2	2	2	2	0	3
Single-Beat Write	4	2	2	2	2	2	0	3
Burst Read	4	2	2	2	2	2	0	3
Burst Write	4	0	0	0	2	2	0	3
Snoop Copyback	1	1	1	1	1	2	0	1
Any Instruction Access	0	0	0	0	0	0	2	3

11.6 TERMINATION OF BUS TRANSACTIONS

This section describes the different methods for terminating transactions on the MC88110 bus. Transactions may be terminated normally, indicating that the transfer was completed successfully, or terminated with an error or a retry indication. Two types of retry terminations are possible: transfer retry and address retry. The address retry terminates the transaction of the current address bus master. The transfer retry terminates the transaction of the current data bus master and is discussed in this section.

The state of several input signals determine the termination for each transaction on the MC88110 bus. These are the data bus busy ($\overline{DBB}$), transfer error ($\overline{TEA}$), transfer acknowledge ($\overline{TA}$), pretransfer acknowledge ($\overline{PTA}$), transfer retry ($\overline{TRTRY}$), and address retry ($\overline{ARTRY}$) signals. The operation of $\overline{ARTRY}$ is described in **11.7 Data Cache Coherency Timing Considerations**. Table 11-19 depicts the encodings of $\overline{DBB}$, $\overline{TA}$, $\overline{TEA}$, and $\overline{TRTRY}$ and the corresponding types of transaction termination.

Table 11-19. Transaction Termination Encodings

$\overline{DBB}$	$\overline{TA}$	$\overline{TEA}$	$\overline{TRTRY}$	Termination
A	A	N	N	Normal
A	x	A	x	Error
A	x	N	A	Transfer Retry

A = Asserted
N = Negated
x = Don't Care

Normal terminations, transfer retry terminations, and error terminations are described and the relative timing diagrams are explained in the following paragraphs.

11.6.1 Normal Transaction Termination with $\overline{TA}$

The assertion of $\overline{TA}$, while $\overline{DBB}$ is asserted and $\overline{TRTRY}$ and $\overline{TEA}$ are negated, signals a normal termination to the processor. The assertion of either $\overline{TRTRY}$ or $\overline{TEA}$ overrides the assertion/negation of $\overline{TA}$ and signals either a transfer retry or an error. For a transaction to terminate normally, the $\overline{PTA}$ signal must be asserted at least one clock cycle before $\overline{TA}$. A normal termination indicates to the MC88110 that the current data transfer has completed successfully. For a read transaction, the data is valid on the data bus and may be latched by the processor. For a write transaction, the data has been accepted by the memory system.

For single-beat transactions, the MC88110 ends the transaction after $\overline{TA}$ is asserted. To end the transaction, the MC88110 releases the data bus by negating $\overline{DBB}$. If it is also the current address bus master, it releases mastership of the address bus by negating $\overline{ABB}$ (unless it is parked and a new transaction is ready to begin). For burst transactions, each beat of the burst must be terminated by $\overline{TA}$ before the transaction is completed. Figure 11-35 shows both single-beat and burst transactions that are completed by normal transaction termination.

In the first clock cycle in Figure 11-35, the MC88110 starts a new transaction by asserting $\overline{TS}$ and $\overline{ABB}$. In the second clock, the MC88110 is granted the data bus and becomes the data bus master by asserting $\overline{DBB}$. Also, in clock 2, $\overline{PTA}$ is asserted by the memory system. In clock cycle 3, the MC88110 detects that $\overline{TA}$ is asserted while $\overline{TEA}$ and $\overline{TRTRY}$ are both negated, so it completes the transaction and relinquished data bus mastership. Since $\overline{BG}$ is asserted, the MC88110 can maintain mastership of the address bus and immediately begin a burst transaction. It becomes the data bus master in clock 4, and detects that $\overline{PTA}$ is asserted in clock 4, and $\overline{TA}$ is asserted in clock 5. This signals the end of the first double-word transfer of the burst. After three more clocks of $\overline{TA}$ asserted successfully (each signaling the end of another double-word transfer), the transaction is complete. Wait states may be added when the MC88110 is the data bus master by not asserting $\overline{TA}$. There is no limit to the number of wait states that may be inserted for any beat of a transaction.

11
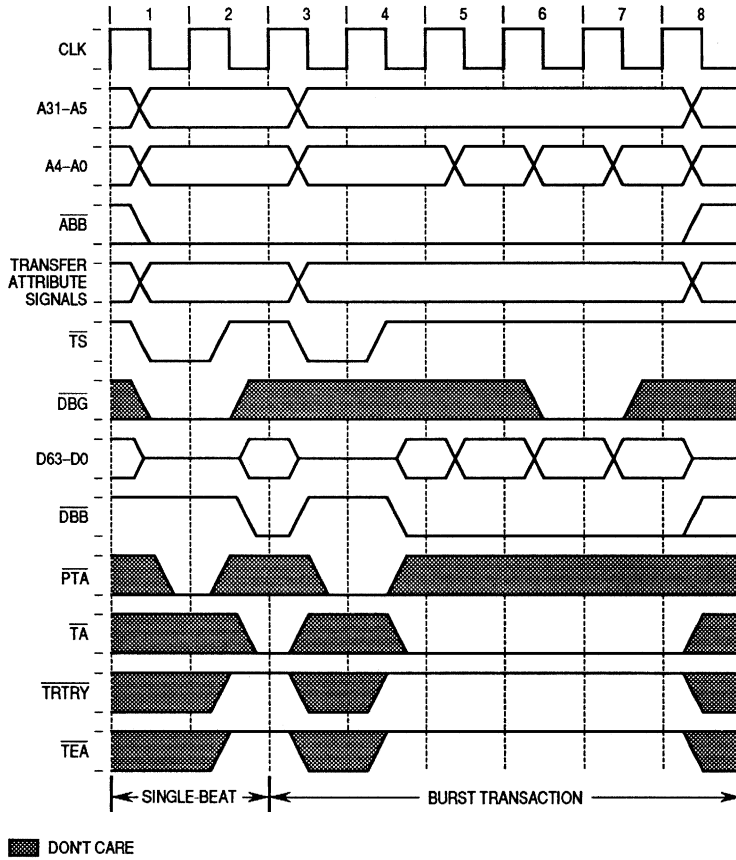
Figure 11-35. Normal Transaction Terminations with $\overline{\text{TA}}$

11.6.2 Decoupled Cache Accesses and $\overline{\text{PTA}}$

The MC88110 has the capability to decouple accesses to the on-chip data cache from bus transactions by setting the DEN bit in the DCTL. When the processor is operating with decoupled cache and bus accesses, the pretransfer acknowledge ($\overline{\text{PTA}}$) signal must be used to explicitly indicate when on-chip data cache accesses must be suspended in order to grant the bus access to the data cache. The $\overline{\text{PTA}}$ signal is used to inform the data cache that the initial assertion of $\overline{\text{TA}}$ may follow on the next rising edge. If decoupled cache accesses are not desired, the $\overline{\text{PTA}}$ signal can be tied to ground. Note that although the $\overline{\text{TA}}$, $\overline{\text{TEA}}$, and $\overline{\text{TRTRY}}$ signals are only sampled when the MC88110 is asserting $\overline{\text{DBB}}$, $\overline{\text{PTA}}$ is sampled independently of data bus mastership. For this reason, split-bus systems may not want to share a common $\overline{\text{PTA}}$ signal.

The window of time between the assertion of $\overline{\text{TS}}$ and $\overline{\text{PTA}}$ allows load and store hits to the data cache to occur without interrupting bus activity. Once $\overline{\text{PTA}}$ is asserted, $\overline{\text{TA}}$ may follow in the next clock, so on-chip data accesses are prevented from accessing the cache. The processor begins sampling $\overline{\text{PTA}}$ simultaneously with the assertion of $\overline{\text{TS}}$, . Once $\overline{\text{PTA}}$ is recognized as asserted by the processor, it is ignored for the duration of the transaction. Note that PTA only has to be asserted for one clock cycle. For more information of the use of decoupled cache/bus accesses see **Section 6 Instruction and Data Caches**.

Figure 11-36 shows a timing diagram of a single-beat transaction that explicitly uses $\overline{\text{PTA}}$. The transaction starts during the first clock, and the processor gains mastership of the data bus during the second clock. For each clock cycle that $\overline{\text{PTA}}$ is negated, the data cache operates independently, because there is guaranteed to be at least one more cycle before $\overline{\text{TA}}$ will be asserted. Therefore, load and store operations that are hits are decoupled from the bus and allowed to access the data cache at the maximum rate. On the rising edge of clock 8, $\overline{\text{PTA}}$ is asserted to inform the cache that $\overline{\text{TA}}$ may follow. The cache then prevents any more load or store operations from accessing the cache until the end of the transaction.

11
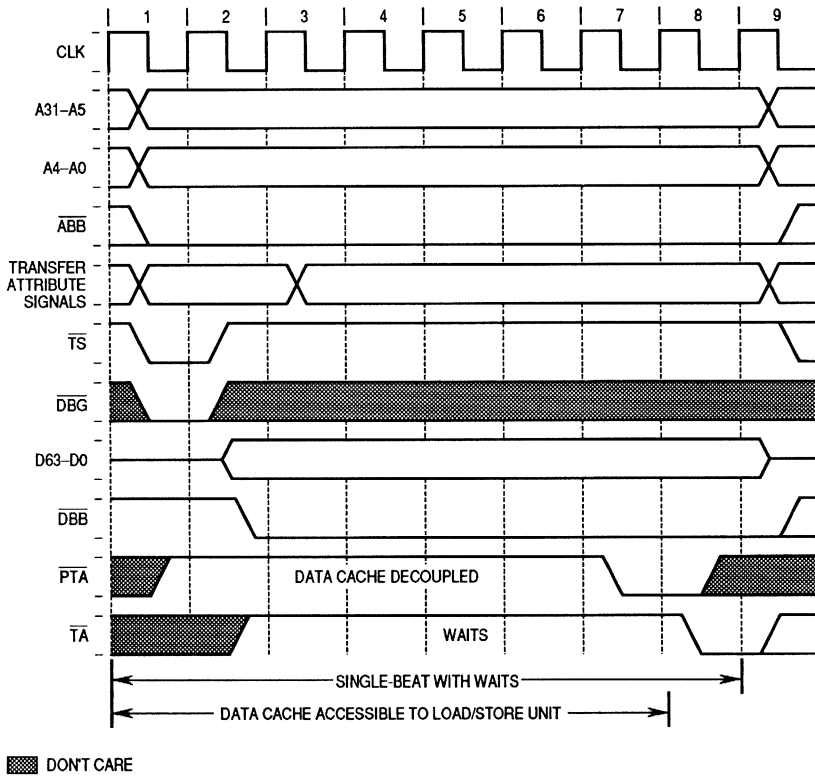
Figure 11-36. Normal Termination of a Single-Beat Transaction with $\overline{\text{PTA}}$ and $\overline{\text{TA}}$

Figure 11-37 shows a diagram for a burst transaction for the data cache that uses $\overline{\text{PTA}}$. For burst transactions, $\overline{\text{PTA}}$ must be asserted before the first time that $\overline{\text{TA}}$ is asserted in order to guarantee correct data cache operation. The data cache is then used only by the bus for the remainder of the burst transaction. The burst transaction begins in the first clock cycle, with $\overline{\text{PTA}}$ negated. The data cache operates decoupled from the bus until clock 8, when $\overline{\text{PTA}}$ asserts, preventing any other internal accesses to the data cache throughout the remainder of the burst transaction. The first beat of the burst is terminated with $\overline{\text{TA}}$ in clock 9, with each of the next three beats following. Note the insertion of a wait state during the third beat by the negation of $\overline{\text{TA}}$ on the rising edge of clock 11. However, load and store operations are not allowed access to the data cache from clock 8 through the end of the transaction.
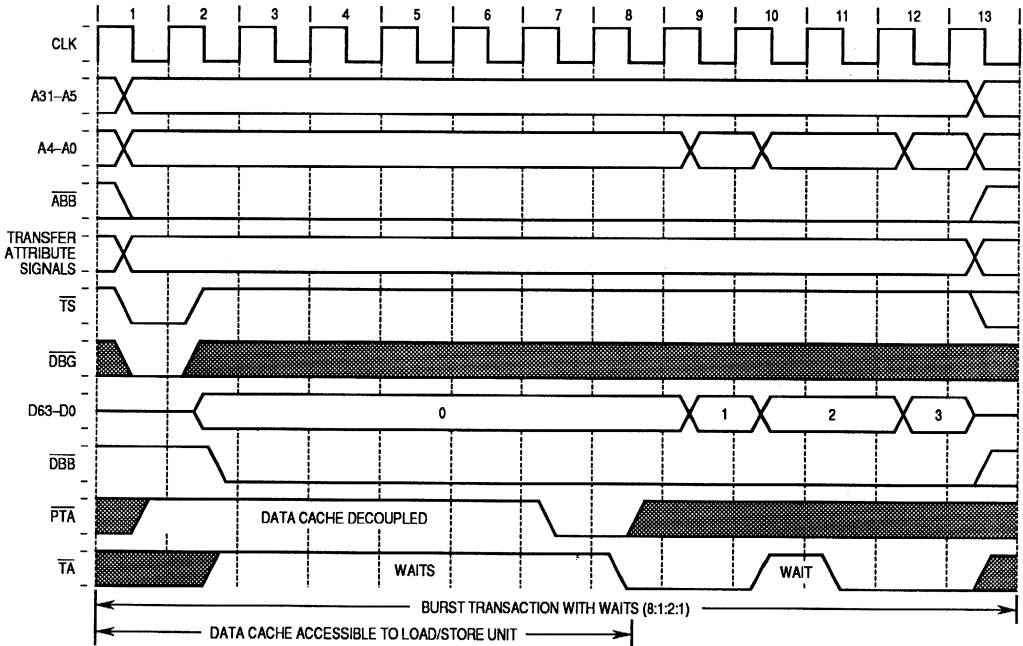
DON'T CARE

Figure 11-37. Normal Termination of a Burst Transaction with $\overline{\text{PTA}}$ and $\overline{\text{TA}}$

11

11.6.3 Transfer Retry Termination

The assertion of $\overline{\text{TRTRY}}$ and the negation of $\overline{\text{TEA}}$ during an MC88110 transaction causes a transfer retry termination of the transaction. If the MC88110 is the current address bus master, but not data bus master, then it does not recognize an assertion of $\overline{\text{TRTRY}}$. Also, the assertion of $\overline{\text{TEA}}$ has a higher priority than $\overline{\text{TRTRY}}$, so the processor detects an error termination if both signals are asserted during a transaction. Refer to **11.6.4 Transfer Error Termination** for more information on error terminations.

For single-beat transactions or the first beat of a burst, a transfer retry causes the processor to immediately terminate the transaction and release the data bus. If the processor is also the address bus master, then the address bus is released at the same time. The burst transaction is then re-initiated from the cache lookup (see **Section 6 Instruction and Data Caches**). For both read and write transactions that terminate with a transfer retry, the previous state of the cache line remains unchanged.

For a transfer retry that occurs on the second, third, or fourth beat of a burst, the processor immediately ends the transaction. If the transaction was a read burst, the burst is not re-initiated later (unless it is required by another instruction or data access), and the corresponding cache line is marked as invalid. If the transaction was a replacement or flush copyback, the state of the cache line is unchanged and the burst is re-initiated. If the transaction was a snoop copyback, it is not re-initiated and the state of the cache line is unchanged.

11

Figure 11-38 shows the timing for a single-beat transfer retry termination. The transaction begins in clock 1, with the processor acquiring data bus mastership in clock 2. On the rising edge of clock 3, TRTRY is asserted while TEA is negated. The processor detects this condition as a transfer retry and relinquishes mastership of both the address bus and the data bus. In clock 6, the transaction is initiated again (this example assumes that the MC88110 was parked). In clock 7, TA is asserted and the transaction is completed successfully.
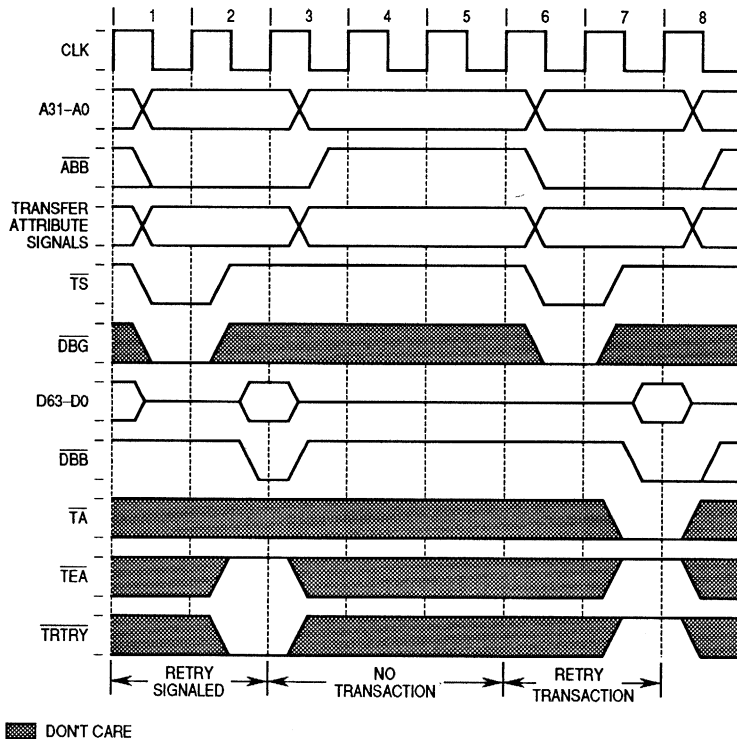
Figure 11-38. Single-Beat Transfer Retry Termination

Figure 11-39 shows the timing for a transfer retry termination that occurs during the first beat of a burst transaction. The transaction begins in clock 1 and the processor gains mastership of the data bus in clock 2. On the rising edge of clock 3 $\overline{\text{TRTRY}}$ is asserted, while $\overline{\text{TEA}}$ is negated. The processor detects a transfer retry and terminates the transaction before it begins the second beat of the burst. The transaction is initiated again from the cache lookup (see **Section 6 Instruction and Data Caches**). This example assumes that the MC88110 is parked, so the transfer retry begins in clock 6. Since each data beat is terminated normally during the retry, the transaction completes normally.
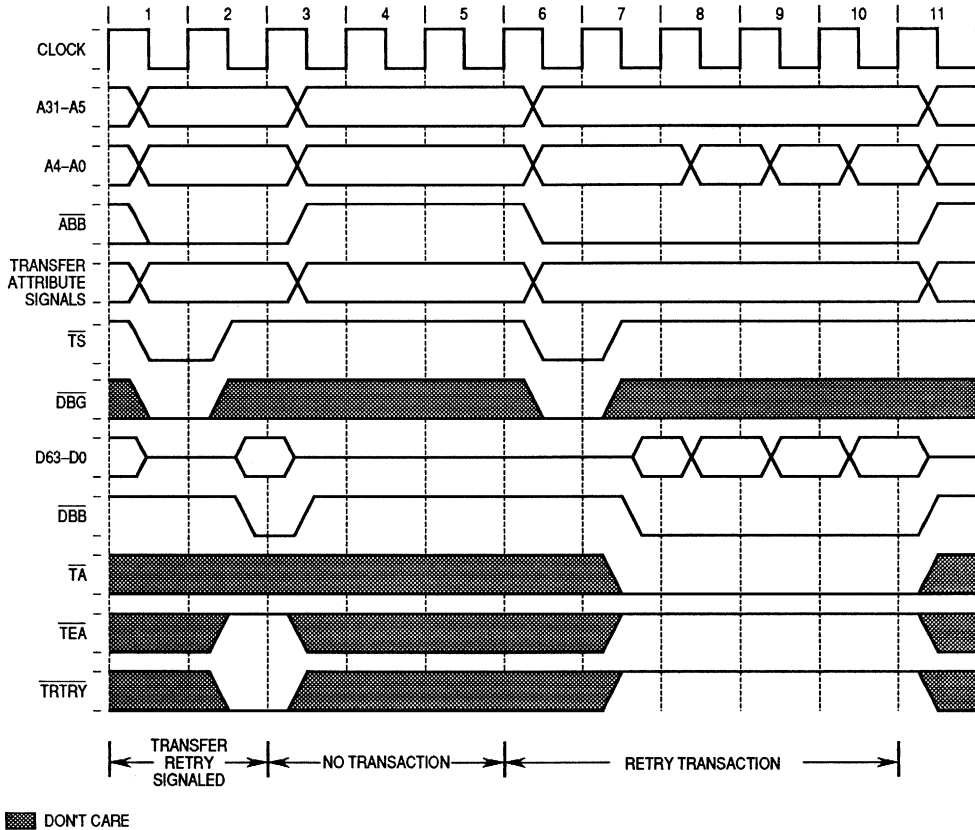
Figure 11-39. Transfer Retry Termination during Beat 0 of a Burst Transaction

Figure 11-40 shows the timing for a transfer retry termination that occurs after the first beat of a burst transaction. In this case, the transaction begins and the first beat terminates with a normal $\overline{TA}$ on the rising edge of clock 3. The second beat of the burst, however, detects the assertion of $\overline{TRTRY}$ on the rising edge of clock 4. The transaction is immediately terminated, but it is not re-initiated later, as it was in the previous two examples, because the critical word has already been received by the processor.
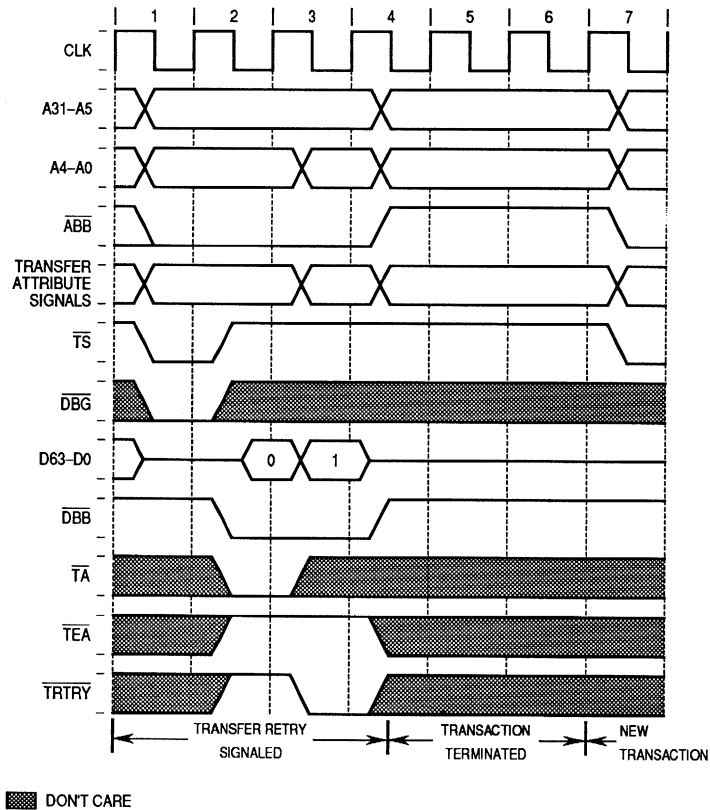
**Figure 11-40. Transfer Retry Termination
after Beat 0 of a Burst Transaction**

11.6.4 Transfer Error Termination

The assertion of $\overline{TEA}$ while the processor is the data bus master results in an error termination, and the processor immediately ends the transaction. The assertion of $\overline{TEA}$ overrides the assertion of either $\overline{TA}$ or $\overline{TRTRY}$ and results in an error termination. The processor relinquishes mastership of the data bus, and, if it is also the address bus master, it relinquishes mastership of the address bus. If there is a different address bus master, the address bus master ignores the assertion of $\overline{TEA}$.

Errors that occur during the first double-word beat of a burst cause the instruction/data access exception to occur. Refer to **Section 7 Exceptions** for a detailed description of exception processing for the MC88110.

Figure 11-41 shows the timing of a transfer error termination for either a single-beat transaction or the first beat of a burst transaction. The transaction begins in clock 1, with the processor becoming the data bus master in clock 2. On the rising edge of clock 3, TEA is asserted. The processor ends the transaction and releases mastership of both the data bus and the address bus.
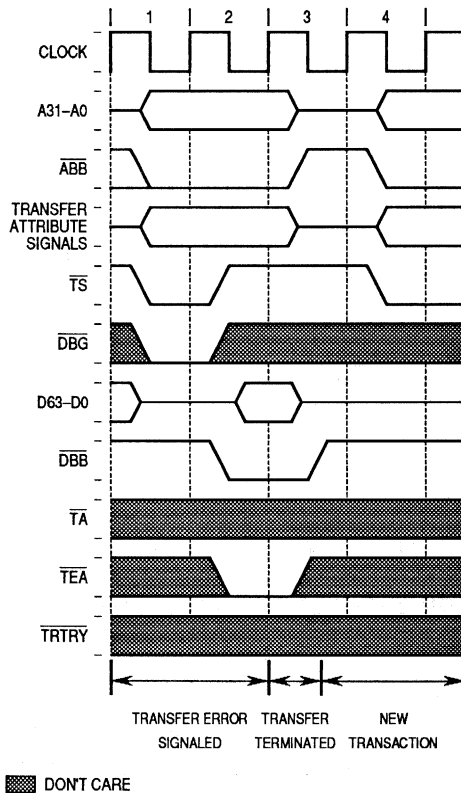
Figure 11-41. Transfer Error Termination

Figure 11-42 shows the timing for a transfer error termination that occurs during the second beat of a burst transaction. The transaction begins in clock 1, with the processor becoming the data bus master in clock 2. The first beat completes with a normal termination caused by the assertion of TA on the rising edge of clock 3. On the rising edge of clock 4, TEA is asserted, terminating the rest of the transaction. Errors in the second, third, or fourth beat of a burst do not cause an exception, unless the transaction is a replacement copyback or the data is being streamed through the cache and forwarded to an execution unit for immediate use.
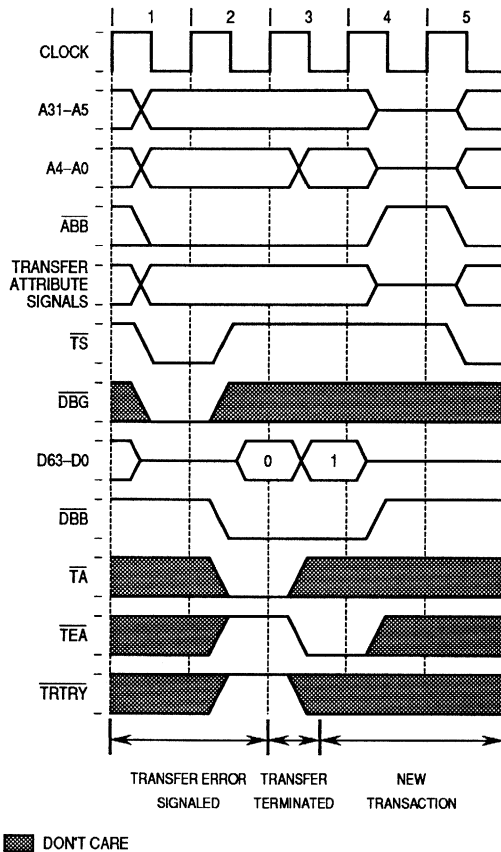
Figure 11-42. Transfer Error Termination
during Beat 1 of Burst Transaction

11.7 DATA CACHE COHERENCY TIMING CONSIDERATIONS

The MC88110 uses a bus snooping protocol to monitor bus transactions performed by other bus masters and to intervene in the access, when required, in order to maintain cache coherency. The following paragraphs describe the operation of the bus when snooping is enabled. For more information on coherency issues internal to the MC88110, refer to **11.3 Data Cache Operation**.

Throughout this discussion of data cache coherency the terms initiating CPU and snooping CPU are used. The initiating CPU is the processor that is the bus master at the beginning of a bus transaction. The snooping CPU is the processor that snoops this transaction.

11.7.1 Snoop Control Signal Overview

Table 11-20 lists the snoop control signals of the MC88110. The snoop request ($\overline{\text{SR}}$) signal is an input to all snooping CPUs that indicates that the current address should be latched because a snoop lookup may be required. $\overline{\text{SR}}$ may be tied to the $\overline{\text{TS}}$ signal of the initiating CPU. The $\overline{\text{SR}}$ signal must be negated and re-asserted between two accesses that need to be snooped, or it will be ignored on the second access. The global ($\overline{\text{GBL}}$) signal is an output when the MC88110 is the initiating CPU and an input when the MC88110 is snooping. The MC88110 only snoops transactions when both the $\overline{\text{SR}}$ and $\overline{\text{GBL}}$ signals are asserted.

Table 11-20. Snoop Control Signal Summary

Signal Name	Mnemonic	Type
Snoop Request	$\overline{\text{SR}}$	Input
Address Retry	$\overline{\text{ARTRY}}$	Input
Snoop Status	$\overline{\text{SSTAT1}}\text{–}\overline{\text{SSTAT0}}$	Output
Shared	$\overline{\text{SHD}}$	Input

When an MC88110 is snooping, its actions depend on the values of the $\overline{\text{GBL}}$, R/$\overline{\text{W}}$, $\overline{\text{INV}}$, and $\overline{\text{TBST}}$ signals. A snooping MC88110 does not snoop any transactions unless it detects both $\overline{\text{SR}}$ and $\overline{\text{GBL}}$ as asserted. When the $\overline{\text{SR}}$ and $\overline{\text{GBL}}$ signals are both asserted, the MC88110 determines whether or not it has a cache hit (see **11.3 Data Cache Operation**) or a collision (see **11.7.8 Split-Bus Snoop Collisions**). If there is a collision, the snooping CPU asserts $\overline{\text{SSTAT1}}$ but does not assert $\overline{\text{SSTAT0}}$. If there is a cache hit, the snooping CPU takes the action described in Table 11-21.

Table 11-21. MC88110 Actions for Snoop Hits

$\overline{\text{SR}}$	$\overline{\text{GBL}}$	R/$\overline{\text{W}}$	$\overline{\text{INV}}$	$\overline{\text{TBST}}$	Action on Snoop Hit
N	x	x	x	x	No action
A	N	x	x	x	No action
A	A	R	N	x	Assert $\overline{\text{SSTAT0}}$; if line was modified, assert $\overline{\text{SSTAT1}}$, perform copyback, and mark line shared-unmodified
A	A	R	A	x	Assert $\overline{\text{SSTAT0}}$ signal and invalidate cache line; if line was modified, assert $\overline{\text{SSTAT1}}$, perform copyback, and invalidate cache line
A	A	W	x	N	Assert $\overline{\text{SSTAT0}}$; if line was modified, assert $\overline{\text{SSTAT1}}$, perform copyback, and invalidate cache line
A	A	W	x	A	Assert $\overline{\text{SSTAT0}}$ and invalidate cache line

11

11.7.2 $\overline{\text{SSTAT1}}$–$\overline{\text{SSTAT0}}$ Timing

The MC88110 asserts $\overline{\text{SSTAT1}}$–$\overline{\text{SSTAT0}}$, if necessary, two clock cycles after the assertion of the $\overline{\text{SR}}$ and $\overline{\text{GBL}}$ inputs. If a snoop copyback must be performed, the MC88110 asserts the bus request one clock cycle after the assertion of $\overline{\text{SSTAT1}}$. The $\overline{\text{SSTAT1}}$–$\overline{\text{SSTAT0}}$ signals remain valid until $\overline{\text{ABB}}$ is negated. Note that if the initiating CPU is parked, $\overline{\text{ABB}}$ may not be negated between transactions. In this case, the snoop status signals are negated when $\overline{\text{SR}}$ is re-asserted.

Figure 11-43 shows the timing for the $\overline{\text{SSTAT1}}$–$\overline{\text{SSTAT0}}$ signals. The initiating CPU starts a global memory access in clock 1, as indicated by $\overline{\text{SR}}$ and $\overline{\text{GBL}}$ asserted. The snooping CPU latches the address and asserts the appropriate snoop status signals two clocks later (if necessary), in clock 3. If the snooping CPU determines that there is a snoop hit to a modified line, then the snooping CPU asserts its bus request one clock after $\overline{\text{SSTAT1}}$–$\overline{\text{SSTAT0}}$. The second transaction in Figure 11-43 shows an example when $\overline{\text{ABB}}$ stays asserted for several clock cycles after the snoop status signals are asserted.
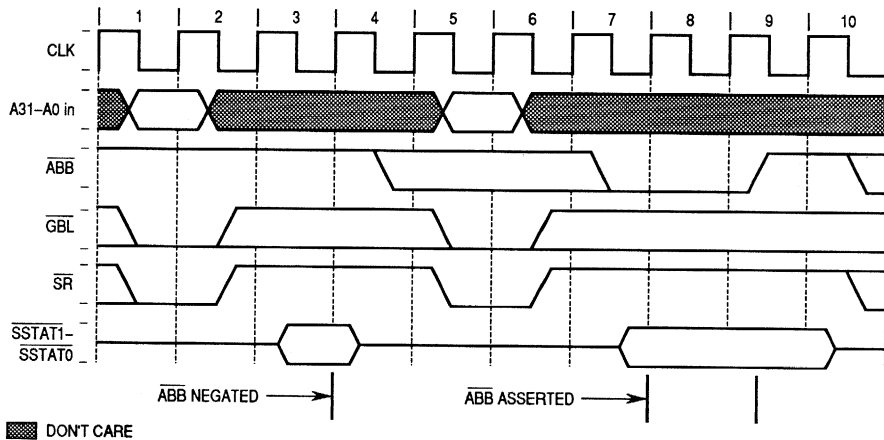
Figure 11-43. Snoop Hit/Miss Indication ($\overline{\text{SSTAT1}}$–$\overline{\text{SSTAT0}}$)

The $\overline{\text{SSTAT1}}$ outputs of the MC88110 can be tied together and the $\overline{\text{SSTAT0}}$ outputs can be tied together without concern about contention. These signals must be tied to pull-up resistors to keep them negated when no processor is driving them. Each time one of the snoop status signals is asserted, the MC88110 negates it before three-stating it. The snoop status signals must be negated in a unique way to avoid contention problems during the transition.

Figure 11-44 shows an example with two processors both driving the $\overline{\text{SSTAT0}}$ signals at the same time (labeled $\overline{\text{SSTAT0}}$-A and $\overline{\text{SSTAT0}}$-B in the diagram). The two $\overline{\text{SSTAT0}}$ signals are tied together and connected to V_{dd} through a pull-up resistor. The combined signal is called $\overline{\text{SSTAT0}}$. In clock 1, a third CPU starts a global transaction. Note that both $\overline{\text{SSTAT0}}$-A and $\overline{\text{SSTAT0}}$-B are three-stated because neither CPU is driving the

signal, but $\overline{\text{SSTAT0}}$ is negated because of the pull-up resistor. Two clocks later, both CPU1 and CPU2 have a cache hit and assert $\overline{\text{SSTAT0-A}}$ and $\overline{\text{SSTAT0-B}}$, respectively. When $\overline{\text{ABB}}$ is negated, the processors must negate $\overline{\text{SSTAT0}}$ to prepare for the next snoop cycle. However, if both CPUs transition from driving the signals low to driving them high, there is the possibility for bus contention for several nanoseconds during the transition. Therefore, $\overline{\text{SSTAT0-A}}$ and $\overline{\text{SSTAT0-B}}$ are each three-stated, then negated, and then three-stated again.
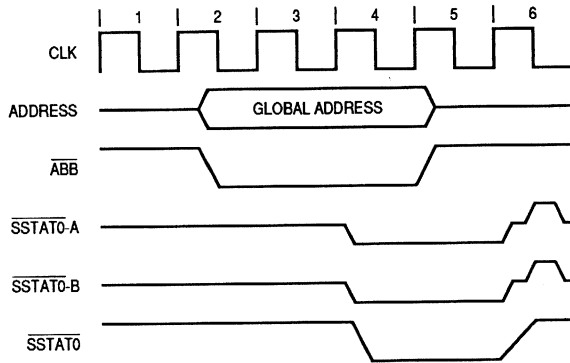
Figure 11-44. Snoop Status Negation Timing

11.7.3 Address Retry Transaction Termination

The $\overline{\text{ARTRY}}$ signal is an input that indicates to the initiating CPU that another device has requested that it terminate the transaction, relinquish mastership of the address and data buses, and retry the transaction at a later time. The timing for the $\overline{\text{SSTAT1}}$ and $\overline{\text{ARTRY}}$ signals allow the $\overline{\text{SSTAT1}}$ output to be directly or indirectly connected to the $\overline{\text{ARTRY}}$ input of other MC88110s. The MC88110, however, qualifies the $\overline{\text{ARTRY}}$ signal with either $\overline{\text{AACK}}$, the first qualified $\overline{\text{TA}}$, or a qualified $\overline{\text{TRTRY}}$ in order to terminate the transaction with an address retry.

When the $\overline{\text{AACK}}$ signal is asserted by the memory system to indicate that the current address has been latched, the processor relinquishes mastership of the address bus. In this way, an alternate bus master can initiate a transaction while the data from the previous transaction is still being transferred. In systems using using this protocol, $\overline{\text{AACK}}$ is also used to qualify $\overline{\text{ARTRY}}$. $\overline{\text{ARTRY}}$ may be asserted before $\overline{\text{AACK}}$ is asserted (but it must remain asserted until $\overline{\text{AACK}}$ is asserted), when $\overline{\text{AACK}}$ is first asserted, or during the first clock cycle after $\overline{\text{AACK}}$ is asserted.

If $\overline{\text{AACK}}$ is negated throughout the transaction, $\overline{\text{ARTRY}}$ is qualified with the first assertion of the $\overline{\text{TA}}$ and/or $\overline{\text{TRTRY}}$. In this case, $\overline{\text{ARTRY}}$ is ignored after $\overline{\text{ABB}}$ is negated.

Figure 11-45 shows the qualification window for $\overline{\text{ARTRY}}$ using $\overline{\text{AACK}}$. Note that the figure shows $\overline{\text{ARTRY}}$ asserted one clock cycle after $\overline{\text{TS}}$. This would not be possible if the snooping processor was an MC88110 because it takes 2 clock cycles for the MC88110

to determine whether or not there was a snoop hit; however, the MC88110 may be connected to a device that can assert $\overline{\text{ARTRY}}$ in one clock.
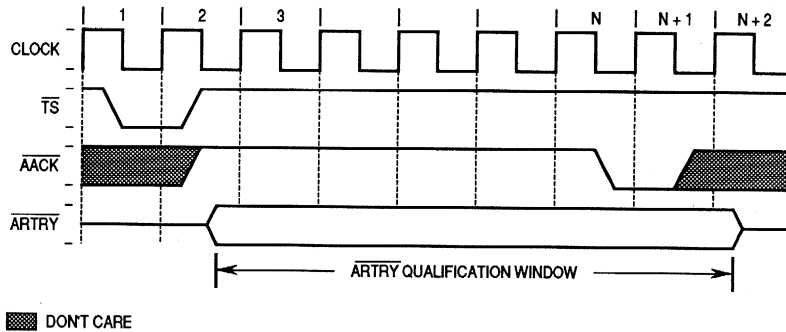
Figure 11-45. $\overline{\text{ARTRY}}$ Qualification with $\overline{\text{AACK}}$

When the initiating CPU detects the qualified assertion of $\overline{\text{ARTRY}}$, it terminates the transaction, releases mastership of the address bus, and re-initiates the transaction from the cache lookup. If a qualified $\overline{\text{ARTRY}}$ occurs before or coincident with a qualified data bus grant, the initiating CPU does not assume data bus mastership. When an MC88110 that is requesting the bus detects that $\overline{\text{ARTRY}}$ is asserted and that $\overline{\text{ABB}}$ was asserted on the previous clock cycle, it removes its bus request and ignores any bus grant. The MC88110 then blocks its bus requests by not asserting $\overline{\text{BR}}$ until $\overline{\text{ARTRY}}$ is negated. Note that the MC88110 does not block $\overline{\text{BR}}$ due to $\overline{\text{ARTRY}}$ if $\overline{\text{ABB}}$ was negated on the previous clock cycle. This blocking protocol, shown in Figure 11-46, allows the snooping CPU an opportunity to acquire mastership of the address bus.
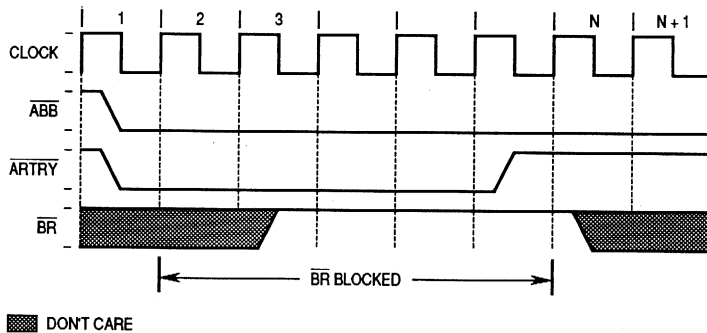
Figure 11-46. $\overline{\text{BR}}$ Blocking Protocol

Note that the memory system can control the length of time that the bus requests will be blocked by controlling when ARTRY is negated. Assuming that the ARTRY signal is controlled by the SSTAT1 signal of the snooping CPU, the memory system can control when ARTRY is negated via the AACK signal. This is because the SSTAT1/ ARTRY signal remains asserted as long as ABB is asserted, and the initiating CPU keeps ABB asserted until the AACK signal is asserted (or the transaction is terminated).

11.7.4 Snoop Miss Timing Example

When the MC88110 is snooping, it takes two clock cycles from the assertion of SR to assert the snoop status signals; therefore, if the MC88110 is initiating a transaction that another MC88110 will be snooping, there must be a minimum of a one clock wait inserted into the transaction to allow for the snooping CPU to assert the snoop status signals. This can be done by delaying TA or DBG by at least one clock. Otherwise, data may be transferred to the MC88110 and forwarded to the register file before the snoop status is known.

Figure 11-47 shows some example snoop transactions from the perspective of the initiating CPU. In clock cycle 1, the first transaction begins, but GBL is negated so this transaction is not snooped. The first transaction is terminated with the TA in clock 2 and a new one begins. In the second transaction, GBL is asserted, so snooping occurs and a wait cycle must be inserted to allow snooping CPUs to assert SSTAT1–SSTAT0. ARTRY is negated, so transaction 2 is terminated with the TA in clock 6 and a new one begins.

The third transaction uses the split bus feature. The address and control information is driven in clock cycle 6. There is a wait reply during the next clock (the TA is negated) and AACK is asserted to signify that the address has been latched and is no longer needed on the bus. In this example, ARTRY is negated, so the transaction is allowed to complete.

11.7.5 Snoop Hit Timing—No Split Bus Example

Figure 11-48 shows an example of a snoop hit protocol. First, CPU1 initiates a global read of a cache line that corresponds to an exclusive-modified copy resident in CPU2. CPU2 asserts SSTAT1, which is coupled to ARTRY. When CPU1 detects that ARTRY is asserted, it qualifies it with AACK, terminates its transaction, and negates the bus request (if it was asserted). In clock 5, CPU2 detects a qualified bus grant and begins its snoop copyback operation. Since ABB was negated by CPU1, SSTAT1 and ARTRY are negated, and CPU1 re-asserts BR. When CPU2 completes the snoop copyback, the arbiter grants CPU1 mastership of the address bus, and CPU1 successfully completes the global read transaction.

11
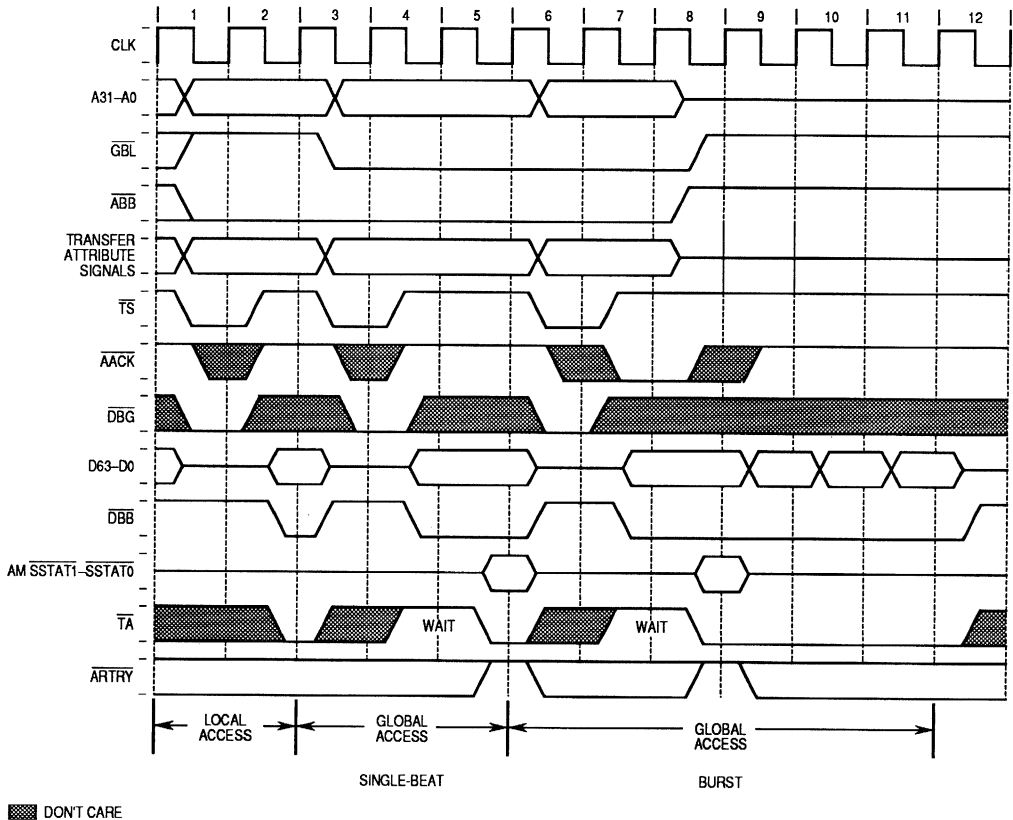
Figure 11-47. Snoop Miss Transactions

11.7.6 Snoop Hit Timing—Split Bus (One-Level) Example

Figure 11-49 shows an example of a snoop hit with a one-level split bus. In clock 1, CPU1 initiates a local read transaction. In clock 3, $\overline{\text{AACK}}$ is asserted, so CPU1 relinquishes address bus mastership while the rest of the line is being transferred. CPU2 then receives a qualified bus grant in clock 4 and initiates a global read transaction. Two clocks later, CPU1 asserts $\overline{\text{SSTAT1}}$, which is tied to $\overline{\text{ARTRY}}$. Note that $\overline{\text{ARTRY}}$ is not acknowledged in clock 7 because $\overline{\text{AACK}}$ is negated (by definition, when using the one-level split bus protocol, $\overline{\text{AACK}}$ cannot be asserted until $\overline{\text{DBB}}$ is negated).

In clock 8, $\overline{\text{DBB}}$ is asserted, so $\overline{\text{AACK}}$ is asserted and it serves to qualify $\overline{\text{ARTRY}}$. Therefore, CPU2 responds to the qualified $\overline{\text{ARTRY}}$ by relinquishing mastership of both buses, thus terminating its transaction. In clock 9, CPU1 receives a qualified bus grant and begins the snoop copyback operation. Note that if a global read corresponding to the address of the snoop copyback is attempted during the copyback, a snoop hit CPU1 signals until the copyback is complete. Therefore, even though $\overline{\text{AACK}}$ is asserted in clock 11 and CPU1 relinquishes mastership of the address bus, CPU2 is prevented from

retrying the global read until clock 14. The arbiter can control this by artificially negating BG until the snoop copyback is complete.

11.7.7 Snoop Hit Timing—Split Bus (Full) Example

Figure 11-50 shows an example of a snoop hit with a full split bus protocol. In clock one, CPU1 initiates a global read transaction. On the rising edge of clock 3, $\overline{\text{AACK}}$ is asserted, so CPU1 relinquishes mastership of the address bus. The memory system must also add one wait cycle before the initial data transfer in order to give the snooping processors enough time to assert the snoop status signals. In this case, the one clock cycle wait cycle was added by delaying the data bus grant to CPU1. In clock 4, CPU1 recognizes the qualified $\overline{\text{ARTRY}}$ and terminates the read. CPU2 then performs the snoop copyback operation and CPU1 retries the global read in the same way as in the previous example.

11.7.8 Split-Bus Snoop Collisions

An MC88110 may initiate a global transaction and receive an $\overline{\text{AACK}}$ before the transaction is completed, thus allowing another processor to initiate a transaction. Therefore, another processor may attempt a global access to the same cache line before the data transfer is complete. This condition is defined as a snoop collision.

To prevent any coherency problems in this case, each CPU maintains an address latch (and comparator) for detection of collision data accesses. This latch is loaded by the CPU when it detects an $\overline{\text{AACK}}$ in response to a global data address and it is cleared when the data transfer is complete. If another CPU initiates a global access to the same cache line, a snoop collision occurs. The snooping CPU asserts $\overline{\text{ARTRY}}$ (via $\overline{\text{SSTAT1}}$), causing the initiating CPU to retry its transaction when the snooping CPU has completed its transaction.

The collision latch is implemented as an additional snoop tag that forces an address retry on all hits, clean or modified. Collision detection occurs for global accesses when the cache tags have not yet been loaded (transaction still in progress).

Figure 11-51 shows a timing example of a snoop collision. CPU1 begins a global transaction in clock cycle 1. $\overline{\text{AACK}}$ is asserted at the end of clock 2 to signal that the address has been latched. CPU1 relinquishes mastership of the address bus and internally latches the address it had been driving. In clock 5, CPU2 begins a global transaction for the same address. At the end of clock 6, $\overline{\text{AACK}}$ is asserted for CPU2. When CPU1 checks the address of the global transaction initiated by CPU2 and detects that it is the same as the address for its transaction still in progress, it asserts $\overline{\text{SSTAT1}}$ (which is connected to $\overline{\text{ARTRY}}$) but does not assert bus request. CPU2 then recognizes that it has received a qualified $\overline{\text{ARTRY}}$ and terminates its transaction.

During this time, data is being transferred for CPU1. Note that if CPU2 asserts $\overline{\text{TS}}$ to retry the transaction before the collision is resolved by the data transfer in progress, then another collision occurs. In this example, the external arbitration circuit avoids this condition by waiting to assert the bus grant for CPU2 until the last clock cycle of data transfer by CPU1.

11

Figure 11-48. Snoop Hit Using $\overline{\text{ARTRY}}$—No Split Bus (Sheet 1 of 2)
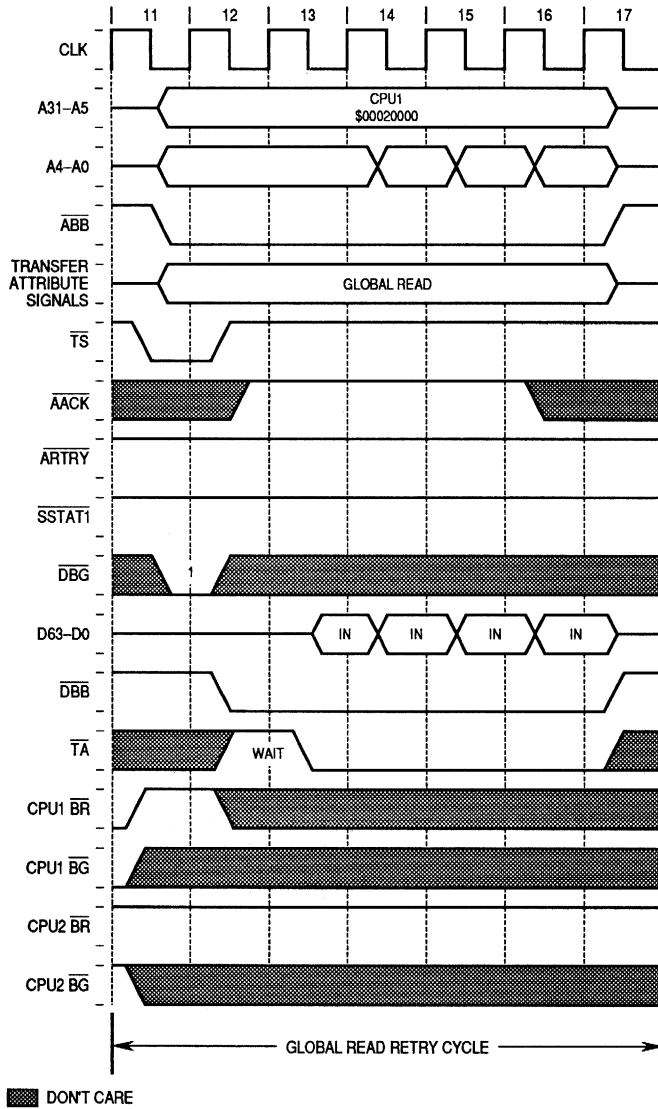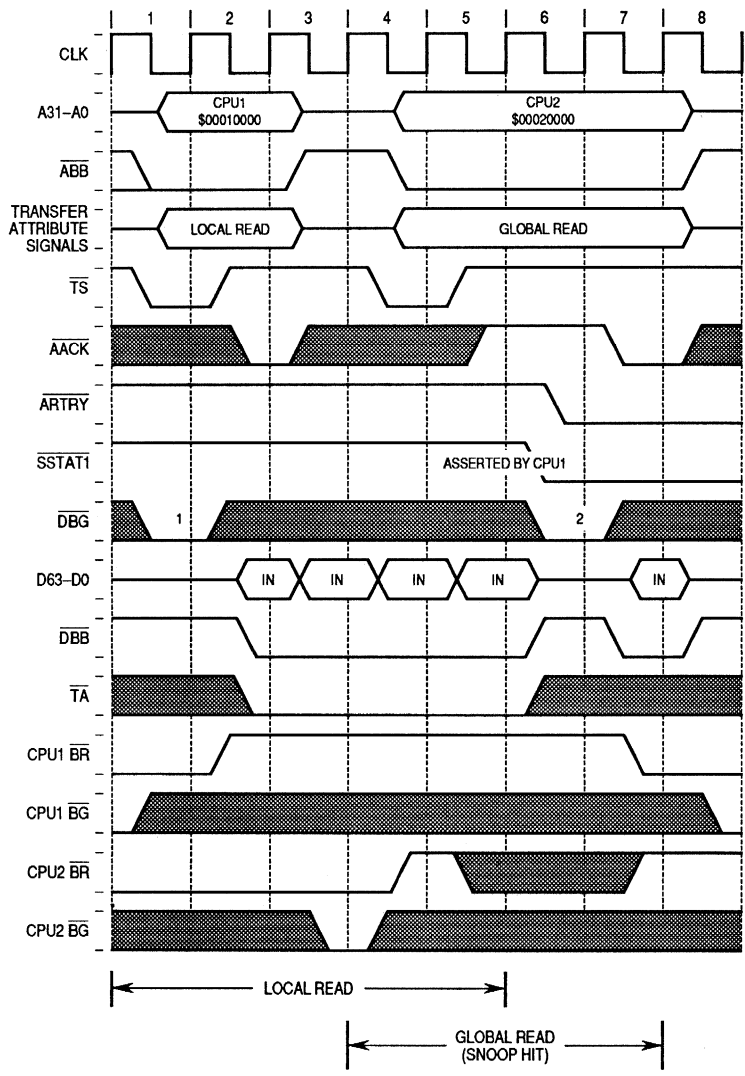
Figure 11-48. Snoop Hit Using $\overline{\text{ARTRY}}$—No Split Bus (Sheet 2 of 2)

DON'T CARE

Figure 11-49. Split Bus (One-Level) Snoop Hit with ARTRY (Sheet 1 of 2)
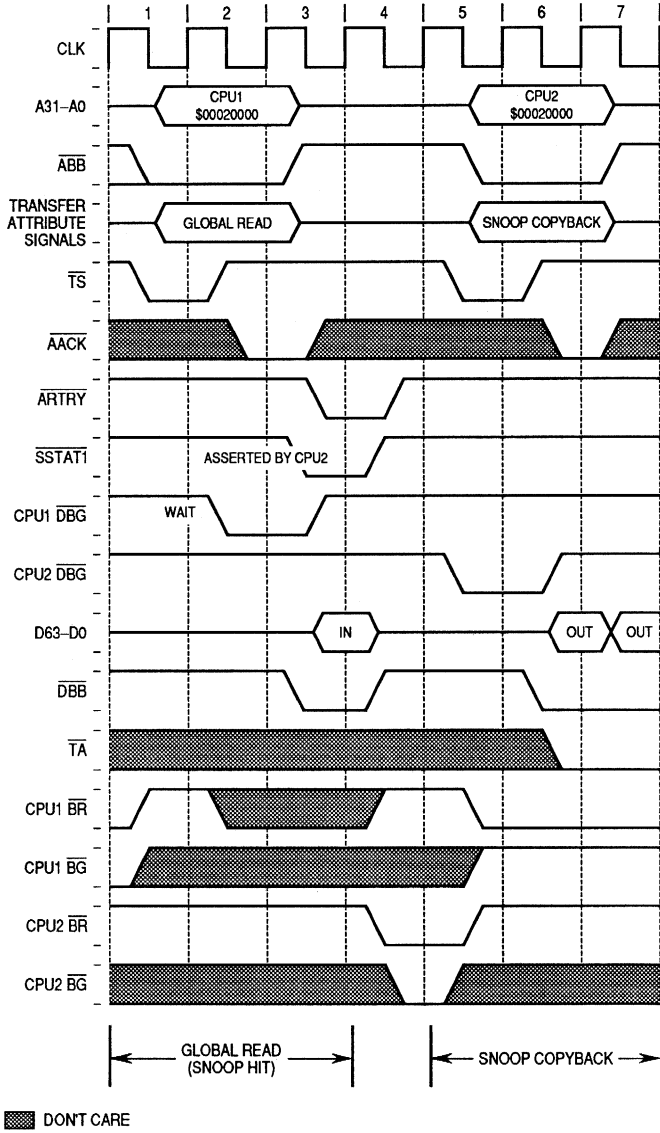
11

DON'T CARE

**Figure 11-49. Split Bus (One-Level) Snoop Hit
with ARTRY (Sheet 2 of 2)**

Figure 11-50. Split Bus (Full) Snoop Hit with $\overline{\text{ARTRY}}$ (Sheet 1 of 2)

11
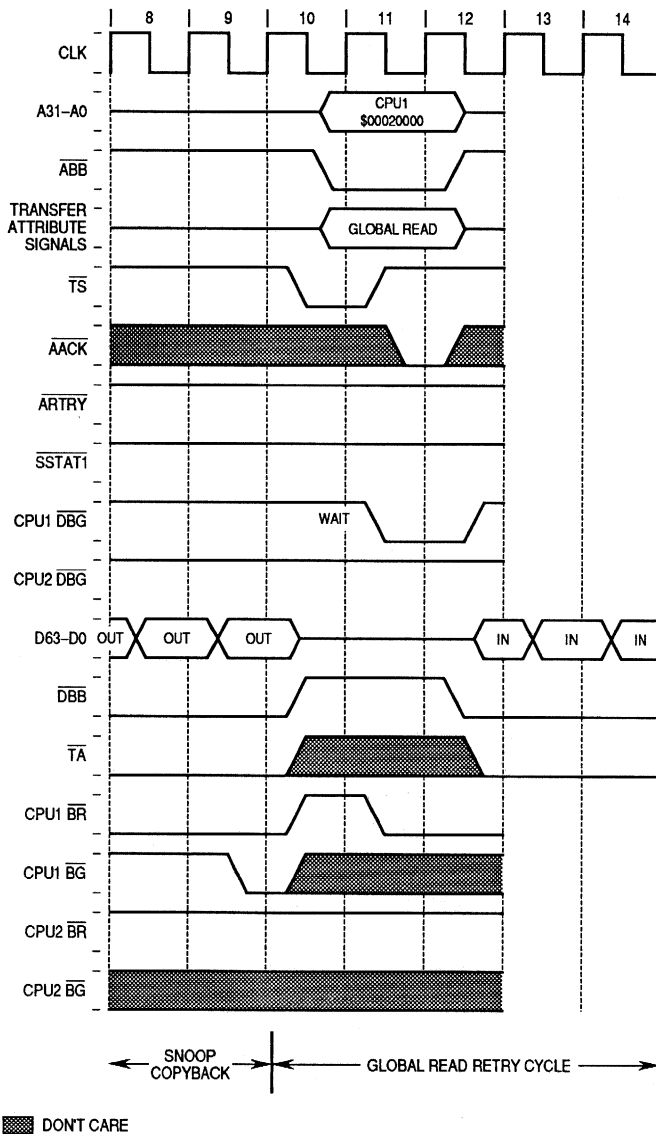
Figure 11-50. Split Bus (Full) Snoop Hit with $\overline{\text{ARTRY}}$ (Sheet 2 of 2)
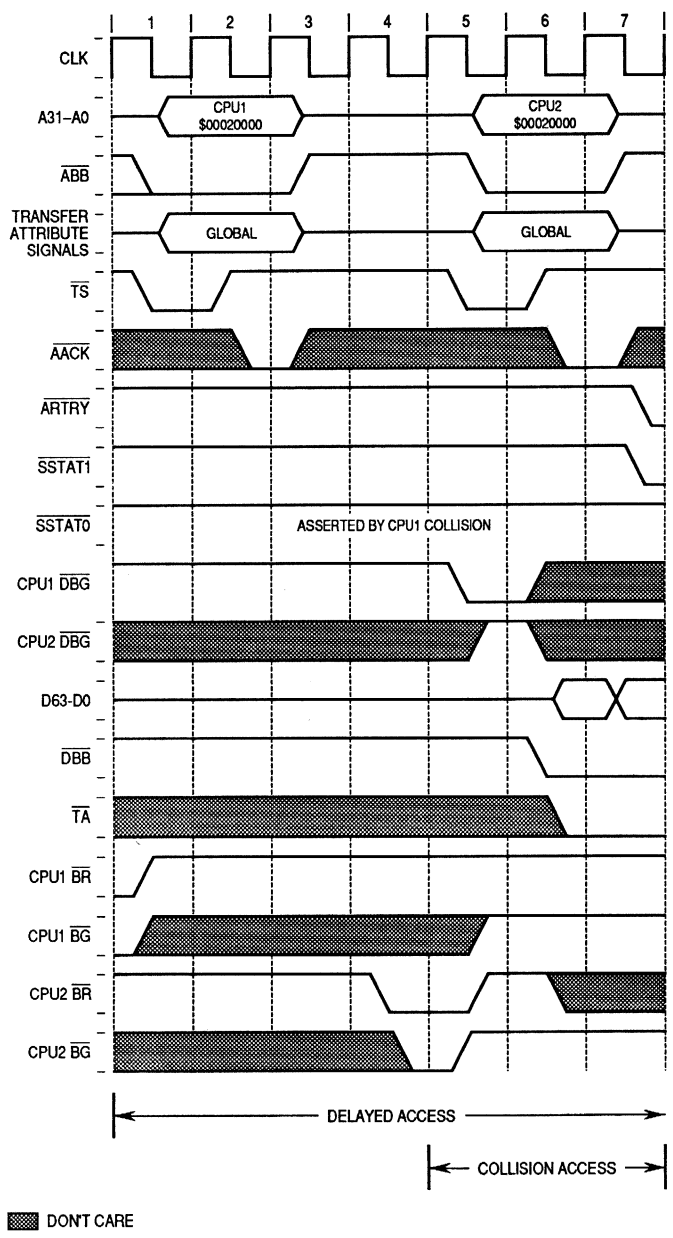
Figure 11-51. Snoop Collision Detection (Sheet 1 of 2)
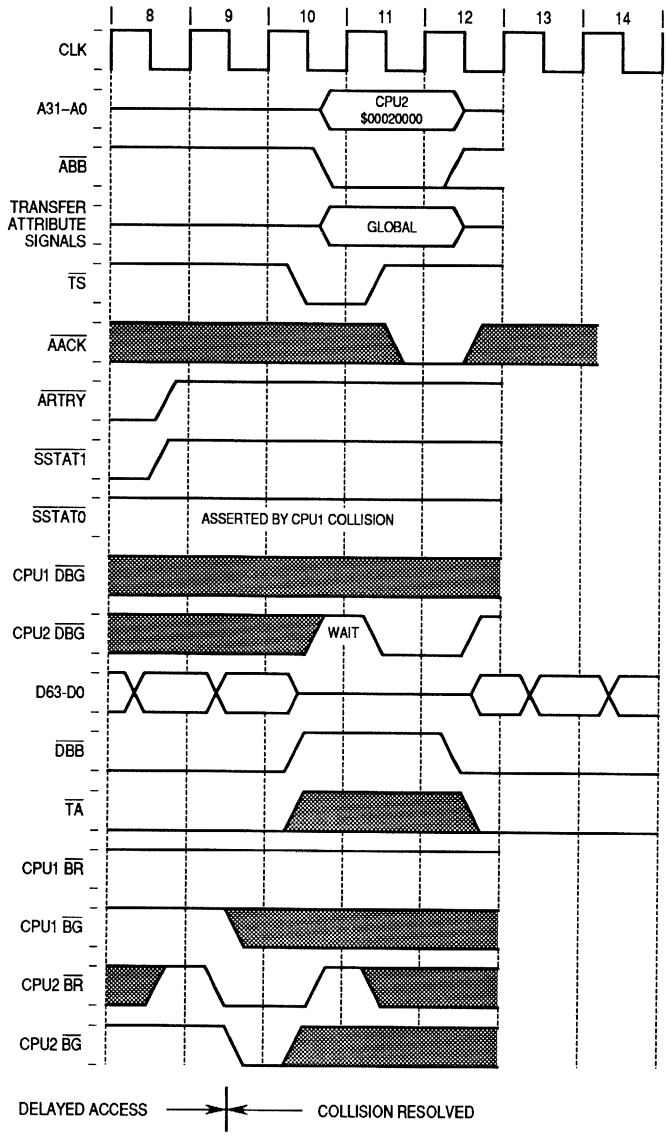
Figure 11-51. Snoop Collision Detection (Sheet 2 of 2)

11.7.9 Snoop Copyback Details

Any retry on a snoop copyback terminates the copyback completely, so the copyback is not retried at a later time. If the MC88110 has a snoop hit to a modified line while it has a transaction outstanding, the snoop copyback is performed when the previous transaction is normally terminated. If the previous transaction is terminated with an error, the snoop copyback is not performed. If the previous transaction is terminated with a retry, then the snoop copyback occurs first, followed by the retried transaction. If there are multiple snoop hits to modified lines while a transaction is outstanding (or while waiting for address bus mastership), then only the last snoop copyback is performed.

11.8 MMU TRANSACTIONS

The MC88110 performs a table search operation when a logical address misses in the PATC when address translation is enabled. The table search operation loads the PATC with a new entry so that a logical-to-physical address translation may be performed. For more information on the MC88110 MMUs, including how the segment descriptor addresses are formed, refer to **Section 8 Memory Management Units**. Table 11-22 shows the state of the transfer attribute signals during table search transactions.

Table 11-22. Transfer Attribute Signals during Table Search

Function	Mnemonic	State
Transfer Burst	$\overline{\text{TBST}}$	Negated
Read/Write	$\text{R/}\overline{\text{W}}$	Read
Cache Inhibit	$\overline{\text{CI}}$	Negated
Write-Through	$\overline{\text{WT}}$	Negated
Memory Cycle	$\overline{\text{MC}}$	Asserted
Invalidate	$\overline{\text{INV}}$	Negated
Global	$\overline{\text{GBL}}$	Negated
Lock	$\overline{\text{LK}}$	Negated
User Page Attributes	$\overline{\text{UPA1}}$–$\overline{\text{UPA0}}$	Asserted if U1 and U0 bits from the area descriptor are set
Transfer Size	TSIZ1–TSIZ0	Word
Transfer Code	TC3–TC0	Code or Data Table Search

11

11.8.1 Hardware Table Search Operation

When a PATC miss occurs, the MC88110 selects a PATC entry for replacement using a first-in-first-out (FIFO) algorithm and then performs a two-level table search to fetch the page descriptor for the referenced address. Figure 11-52 illustrates the relative timing for a table search operation. In the first clock cycle, the MC88110 drives the segment descriptor address. The segment descriptor is read with a single-beat transaction, and it completes in clock 3. The MC88110 then takes two clock cycles to calculate the page descriptor address and drives that address in clock 5. The page descriptor fetch is also a single-beat transaction, and it completes successfully in clock 7. The MC88110 then takes three clock cycles to load the page descriptor and retry the cache access. When the cache access is retried, there is a PATC hit, and the required transaction begins in clock 10. Note that the MC88110 must re-arbitrate for mastership of the bus between each of these transactions.
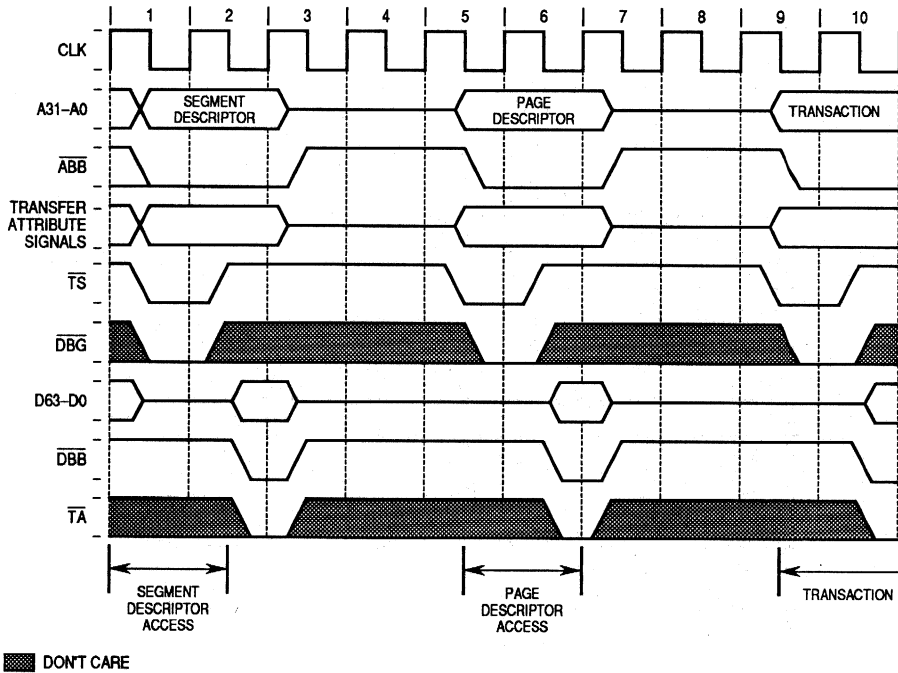
Figure 11-52. Hardware Table Search Operation Timing

11.8.2 Hardware Table Search Operation with Indirection

It is sometimes desirable for multiple pages to be mapped with a common page descriptor. This is supported in the MC88110 by indirection descriptors. When using indirection descriptors, the table search operation is a three-level search. The first access is to the segment descriptor, the second to the indirection descriptor, and the third to the page descriptor. The timing for this type of table search operation is shown in Figure 11-53. Note that the MC88110 takes two clock cycles between accesses to calculate the descriptor addresses and that there are three clock cycles between the successful completion of the page descriptor access and the initiation of the required transaction. Also, note that the MC88110 must re-arbitrate for mastership of the bus between each transaction.
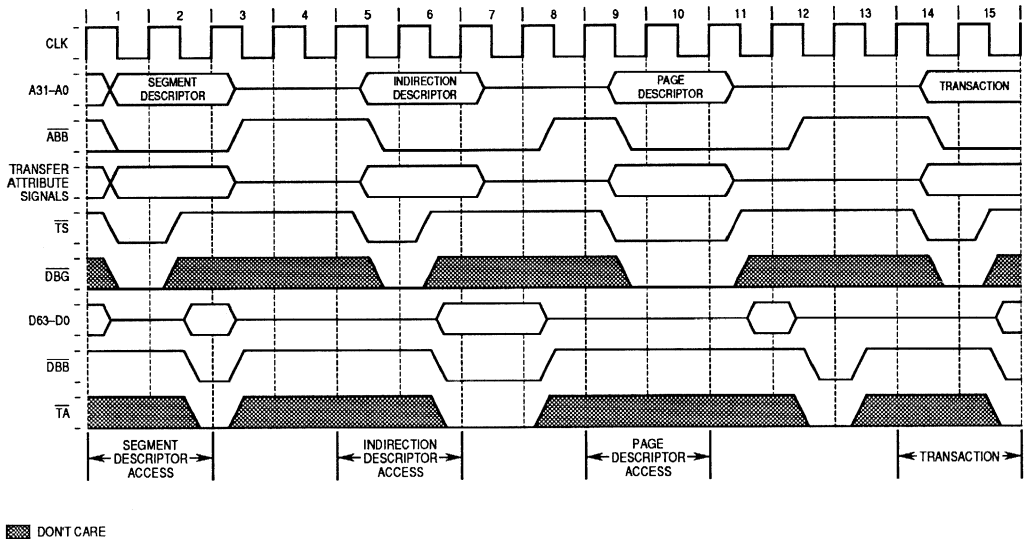
Figure 11-53. Hardware Table Search with Indirection

11.8.3 Hardware Table Search Operation with $\overline{\text{TRTRY}}$

Figure 11-54 shows an example of a hardware table search operation in which the indirection descriptor access terminates with a transfer retry. In this example, the first access completes successfully, as in the previous two examples. In the second access, however, the $\overline{\text{TRTRY}}$ signal is asserted and the $\overline{\text{TEA}}$ signal is negated, indicating a transfer retry. The MC88110 relinquishes mastership of both the address and data bus, waits one clock cycle, and retries the indirection descriptor access. On the second attempt, the indirection descriptor access completes successfully, and the table search continues as in the previous example.

11.8.4 Hardware Table Search with Snoop Copyback

Because the MC88110 does not retain bus mastership between the transactions performed for a hardware table search, it is possible for alternate masters to gain control of the bus before the table search is complete. If another CPU initiates a global transaction with snooping enabled, the MC88110 may have to interrupt the table search operation in order to perform a snoop copyback operation. If this occurs, the MC88110 must fetch the last descriptor when the snoop copyback is complete.

Figure 11-55 shows an example of a hardware table search operation that is interrupted by the global read initiated by another processor. In the first clock cycle, CPU2 initiates a single-beat access to the segment descriptor, which completes successfully in clock 2. This is followed by the indirection descriptor access which also completes successfully; however, in clock 8, the address bus is granted to CPU1, which initiates a global read transaction. Since CPU2 has a modified copy of the corresponding cache line that CPU1 is accessing, CPU2 asserts $\overline{\text{SSTAT1}}$, causing the $\overline{\text{ARTRY}}$ input to CPU1 to be asserted. CPU1 then relinquishes mastership of the address bus, and CPU2 performs the snoop copyback. When the copyback is complete, the arbiter grants the bus to CPU1, which must restart its table search with the indirection descriptor.
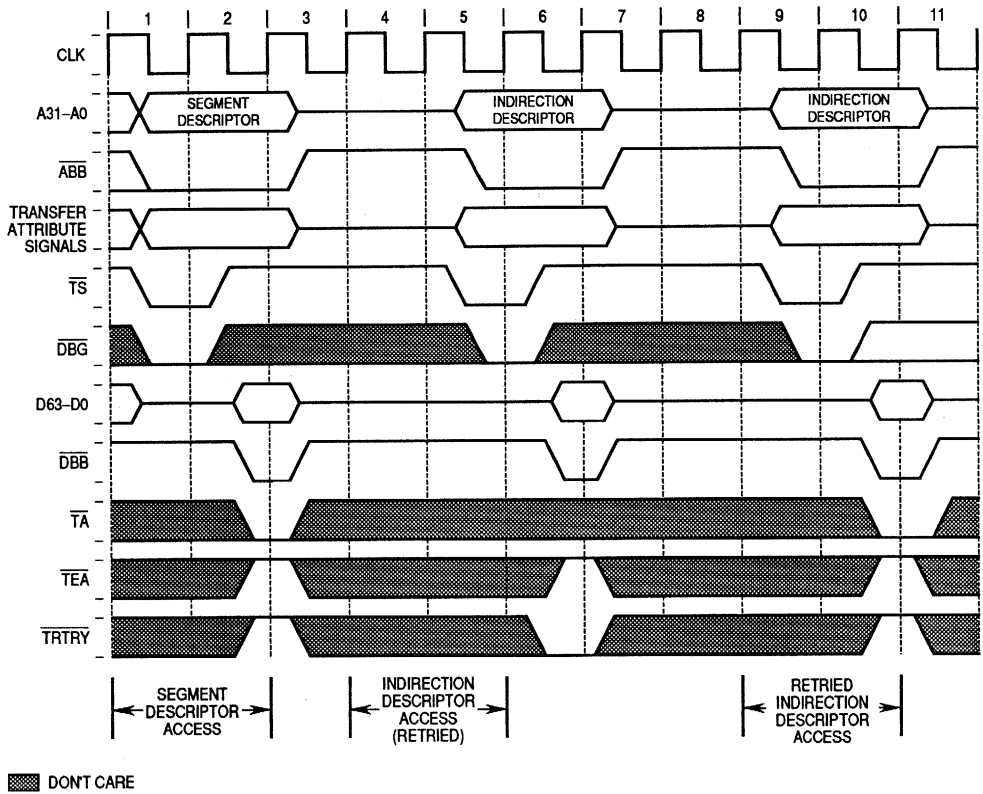
Figure 11-54. Hardware Table Search with $\overline{\text{TRTRY}}$ (Sheet 1 of 2)

11
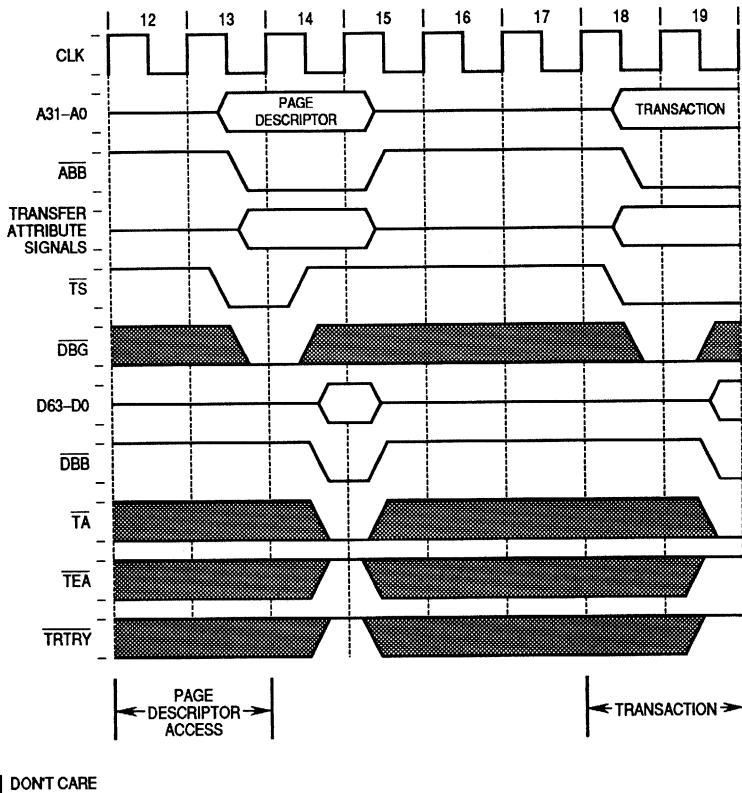
Figure 11-54. Hardware Table Search with $\overline{\text{TRTRY}}$ (Sheet 2 of 2)
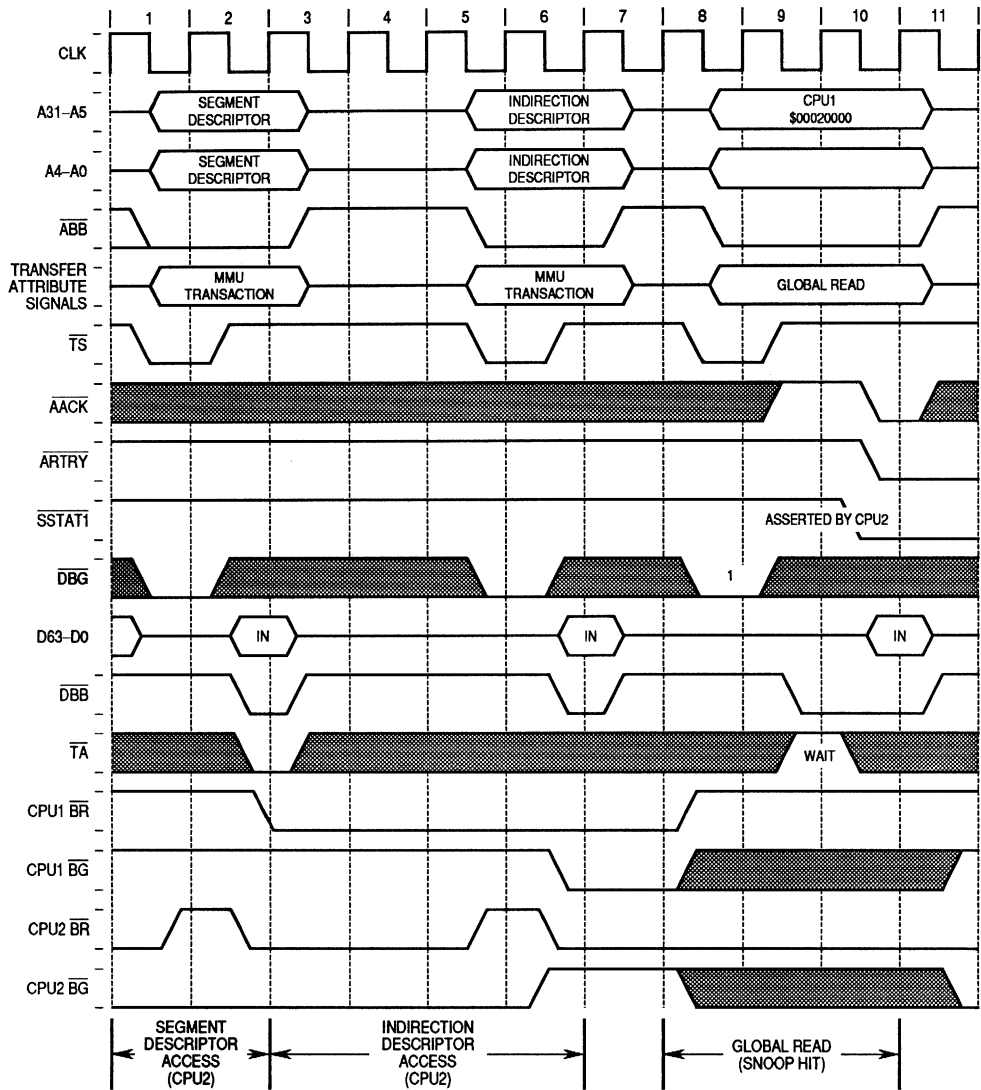
Figure 11-55. Hardware Table Search with Snoop Copyback (Sheet 1 of 2)
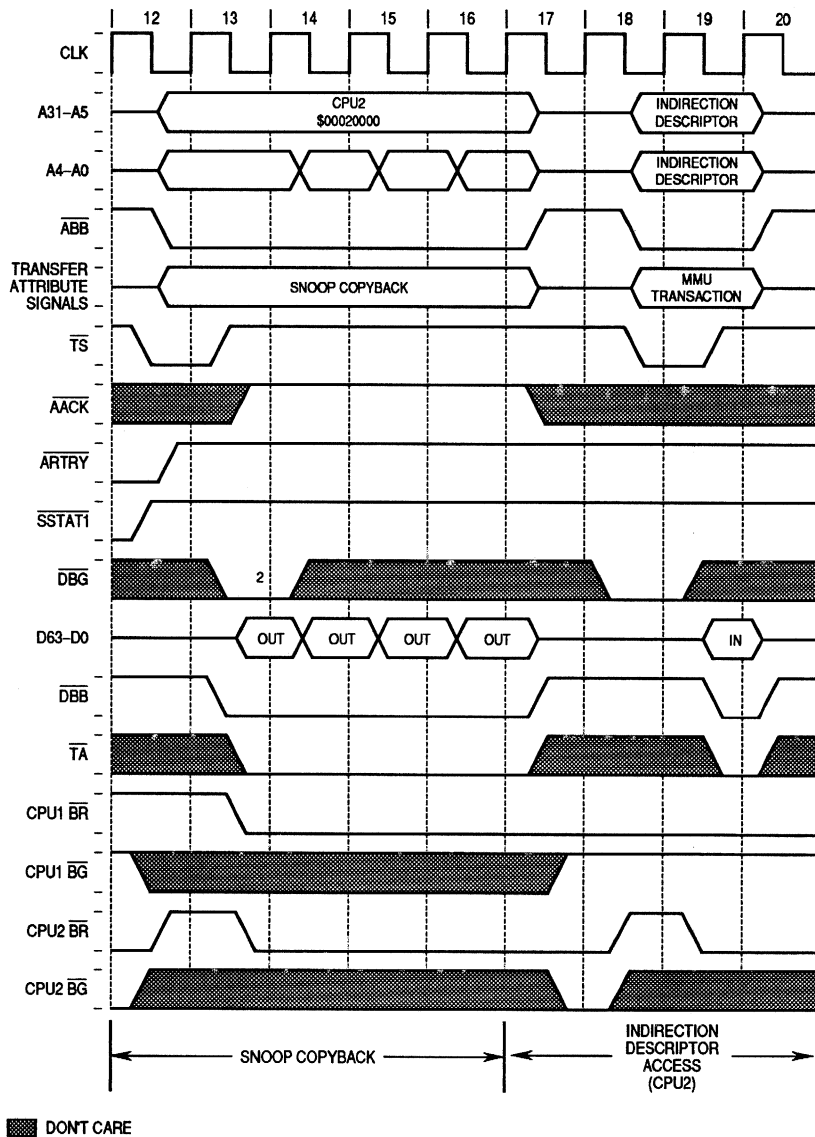
Figure 11-55. Hardware Table Search with Snoop Copyback (Sheet 2 of 2)

11.9 RESET OPERATION

The reset input signal ($\overline{RST}$) is asserted by an external device to reset the processor. When power is applied to the system, external circuitry should assert $\overline{RST}$ for a minimum of 200 ms after V_{CC} is within tolerance. Figure 11-56 is a timing diagram of the power-on reset operation, showing the relationships between V_{CC}, $\overline{RST}$, PSTAT2–PSTAT0, and the bus signals. The CLK signal is required to be stable by the time V_{CC} reaches the minimum operating specification.

Once $\overline{RST}$ negates, the processor is internally held in reset for another three clock cycles. During the reset period, $\overline{BR}$ and $\overline{TS}$ are negated, and all other three-statable signals are three-stated. Once the internal reset signal negates, the processor must be granted the bus. After this, the first bus transaction for reset exception processing begins. In Figure 11-56, the processor is parked on the bus and can start its first transaction immediately after the internal reset signal negates.

The processor status signals provide limited visibility of the CPU status. The three-bit value loaded through PSTAT2–PSTAT0 at reset determines the function of the signals during normal operation. The PSTAT2–PSTAT0 signals are sampled on every clock cycle in which reset is asserted. When reset is negated, the MC88110 waits a minimum of three clock cycles before driving the PSTAT2–PSTAT0 signals. This gives the off-chip driving logic time to go into a high-impedance state to avoid possible bus contention.
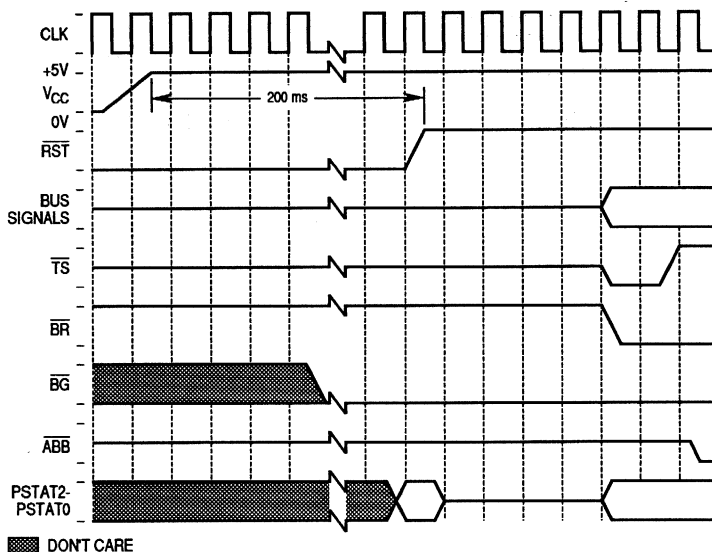
Figure 11-56. Initial Power-On Reset Timing

For processor resets after the initial power-on reset, $\overline{\text{RST}}$ should be asserted for at least 16 clock cycles. Figure 11-57 shows the timing associated with a reset when the processor is executing bus transactions.

Resetting the processor causes all output signals to three-state. In addition, the processor initializes control registers appropriately for a reset exception. Exception processing for a reset operation is described in **Section 7 Exceptions**.
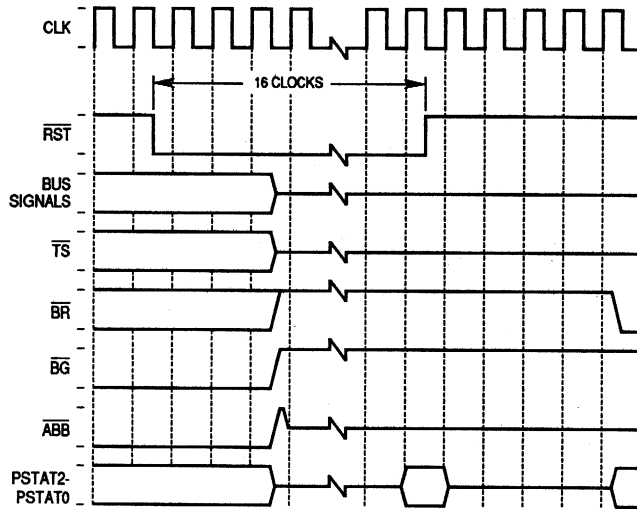
Figure 11-57. Normal Reset Timing

11

11.10 IEEE 1149.1 TEST ACCESS PORT

The MC88110 includes dedicated user-accessible test logic that is fully compatible with the *IEEE 1149.1 Standard Test Access Port and Boundary Scan Architecture*. Problems associated with testing high density circuit boards have led to development of this standard under the sponsorship of the Test Technology Technical Committee of IEEE Computer Society and the Joint Test Action Group (JTAG). The MC88110 implementation supports circuit board test strategies based on this standard.

The test logic implemented on the MC88110 includes a test access port (TAP) consisting of five dedicated signals, a 16-state controller, and two test data registers. A boundary scan register links all device signals into a single shift register. The test logic is implemented using static logic design and is independent of the system logic of the device. The MC88110 implementation provides capabilities to:

1. Perform boundary scan operations to test circuit board electrical continuity.
2. Bypass the MC88110 for a given circuit board test by effectively reducing the test data register to a single cell.
3. Sample the MC88110 system signals during operation and transparently shift out the result in the boundary scan register.
4. Statically control the output state (high, low, high-impedance) of all signals that can be outputs. The control state is latched or clamped within the MC88110 device even though the enabled test data register is the single-bit bypass register.
5. Quickly force all signals into the high-impedance state while enabling the single-bit bypass register as the test data register.
6. Enable a weak pull-up current device on all signals controlled by the boundary scan register while performing boundary scan operations to provide for a deterministic test result in the presence of a continuity fault.

NOTE

Certain precautions must be observed to ensure that the IEEE 1149.1 test logic does not interfere with nontest operation. See **11.10.4 Non-IEEE 1149.1 Operation** for details.

11

11.10.1 JTAG Overview

This document includes those aspects of the IEEE 1149.1 implementation that are specific to the MC88110 and is intended to be used in conjunction with the supporting IEEE document. The scope of this description includes those items required by the standard to be defined and, in certain cases, provides additional information specific to the MC88110 implementation. For internal details and applications of the standard, refer to the IEEE 1149.1 document.

A block diagram of the MC88110 implementation of IEEE 1149.1 test logic is shown in Figure 11-58. The MC88110 implementation includes a dedicated TAP consisting of the following signals:

TCK— a test clock input to synchronize the test logic

TMS— a test mode select input (with an internal pull-up resistor) sampled on the rising edge of TCK to sequence the test controller's state machine

TDI— a test data input (with an internal pull-up resistor) sampled on the rising edge of TCK

TDO— a three-statable test data output actively driven in the shift-IR and shift-DR controller states that changes on the falling edge of TCK

TRST— an asynchronous reset with an internal pull-up resistor which provides initialization of the TAP controller and other logic as required by the standard

NOTE

The pull-up resistor will pull TRST out of test reset.

Figure 11-58. IEEE 1149.1 Test Logic Block Diagram

11.10.2 Three-Bit Instruction Register

The MC88110 IEEE 1149.1 implementation includes the three mandatory public instructions (BYPASS, SAMPLE/PRELOAD, and EXTEST) and three optional public instructions (CLAMP, HI-Z, and EXTEST_PULLUP). The EXTEST_PULLUP instruction is very similar to the EXTEST instruction; however, in the EXTEST_PULLUP instruction, the DC parametric of each signal controlled by the boundary scan register is affected by the addition of a weak pull-up device. The MC88110 includes a three-bit instruction register without parity as shown in Figure 11-59. The register consists of an instruction shift register and a parallel output register. Data is transferred from the instruction shift register to the parallel output register during the update-IR controller state. The three bits are used to decode the six unique instructions as shown in Table 11-23.

LENGTH = 3 BITS; NO PARITY BIT

Figure 11-59. Instruction Register Implementation

The parallel output of the instruction register is preset to all ones in the test-logic-reset controller state. Note that this preset state is equivalent to the BYPASS instruction.

Table 11-23. Instruction Register Encodings

Code			Instruction
B2	B1	B0	
1	1	1	BYPASS
1	1	0	Reserved (BYPASS)
1	0	1	Reserved (BYPASS)
1	0	0	SAMPLE/PRELOAD
0	1	1	CLAMP
0	1	0	EXTEST_PULLUP
0	0	1	HI-Z
0	0	0	EXTEST

During the capture-IR controller state, the parallel inputs to the instruction shift register are loaded with the three-bit binary value, 001. The parallel outputs, however, remain unchanged by this action since an update-IR signal is required to modify them.

Note that skipping the shift-IR state allows the 001 value to be updated as the current instruction, therefore entering the HI-Z instruction. This is useful for the board test applications that are not utilizing the full integrated boundary scan test techniques, but would still like to use the HI-Z instruction for board test isolation purposes.

11.10.2.1 EXTEST (000). The external test (EXTEST) instruction selects the boundary scan register, including cells for all device, clock, and associated control signals. The resistor 1 and resistor 2 signals are associated with analog signals and are not included in the boundary scan register. EXTEST also asserts internal reset for the MC88110 system logic in order to force a predictable internal state while performing external boundary scan operations.

By using the TAP, the boundary scan register is capable of 1) scanning user-defined values into the output buffers, 2) capturing values presented to input signals, and 3) controlling the direction and value of bi-directional signals.

The boundary scan register has bit cells associated with 15 pure input signals, and the other 262 cells are associated with 131 bi-directional signals. All MC88110 bi-directional signals and output signals (treated as bi-directional), have both a boundary scan register bit for signal data and a boundary scan register bit for direction control. This allows the user great flexibility and control of the direction of every signal that can be an output. Due to the implementation of the individual direction control cell for each signal, some signals that are typically referenced as output-only can be programmed as input and have input data sampled into the boundary scan register.

The boundary scan bit definitions are shown in Table A-1. The first column in the table defines the ordinal position of the bits in the boundary scan resister. The shift resister cell nearest the TDO signal (i.e., first to be shifted out) is defined as bit zero while the last bit to be shifted out is 276.

11

The second column in Table A-1 references one of the three boundary scan cell types depicted in Figures 11-60 through 11-63. The cells are input-only (I.CELL), compound input and output cell (IO.CELL), and bi-directional control cell (IO.CTL1). All control bits use the same active level where logic 1 corresponds to output driver ON. The third column in Table A-1 lists the signal name for all signal related cells or defines the name of bi-directional control register bits. The fourth column lists the signal type (for convenience) where input indicates pure input signal and inout indicates a bi-directional signal. Note that when sampling the bi-directional data cells (IO.CELL), the cell data can be interpreted only after examining the IO.CTL1 cell to determine signal directionality.
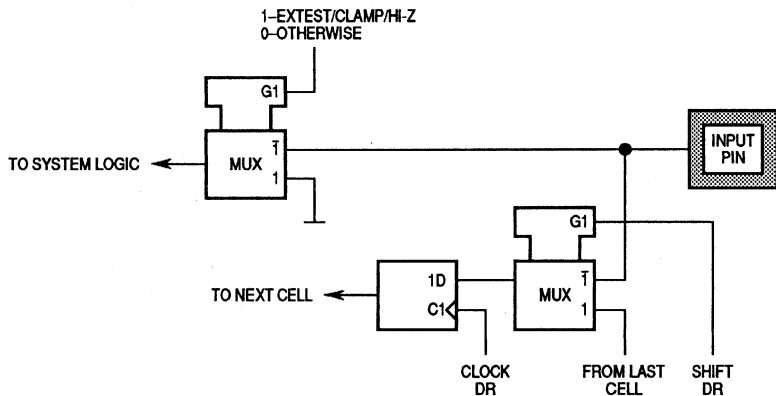
Figure 11-60. Input Signal Cell (I.Pin)
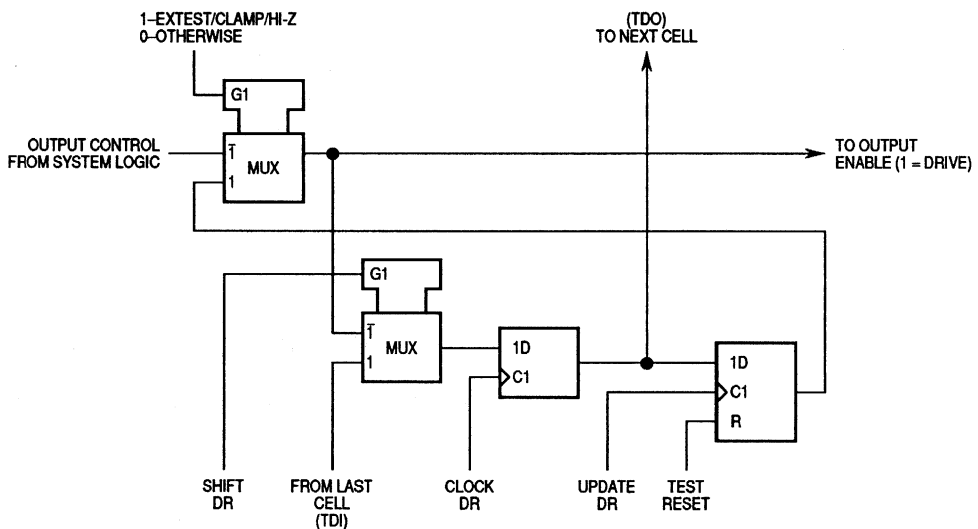
Figure 11-61. Active High Output Control Cell (IO.Ctl1)
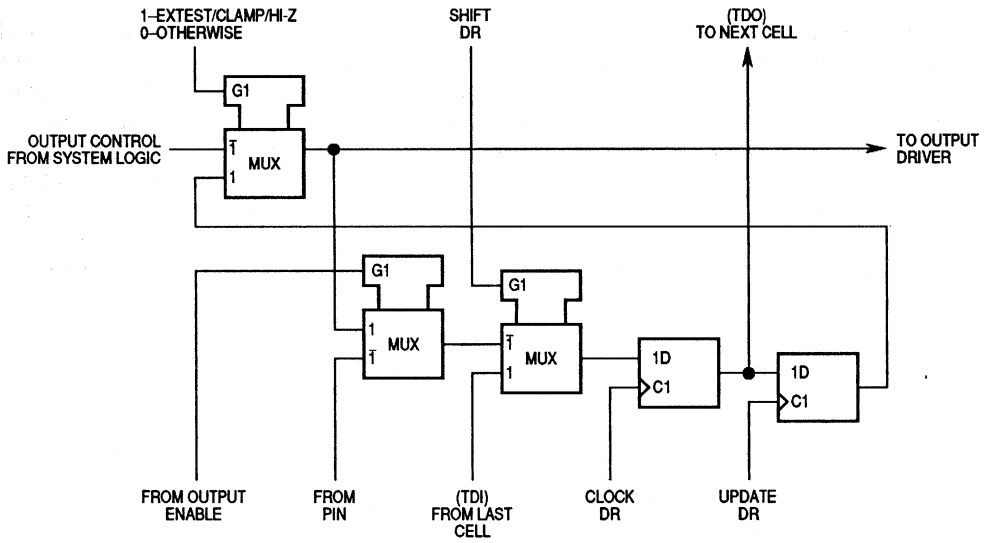
Figure 11-62. Bi-Directional Data Cell (IO.Cell)
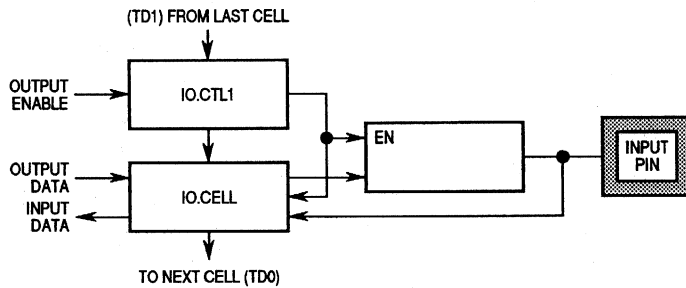
Figure 11-63. Bi-Directional Cell Arrangement

11.10.2.2 BYPASS (111). The BYPASS instruction selects the single-bit bypass register as shown in Figure 11-64. This creates a shift-register path from the TDI signal to the bypass register and finally to the TDO signal, circumventing the boundary scan register. This instruction is used to enhance test efficiency when a component other than the MC88110 becomes the device under test. In this instruction, the MC88110 system logic is independent of the test access port.
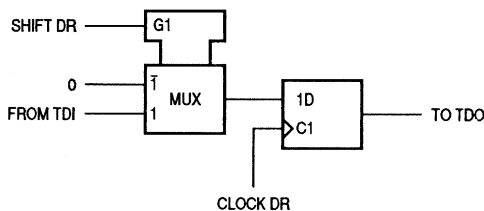
Figure 11-64. Bypass Register

When the bypass register is selected by the current instruction, the shift-register stage is set to a logic 0 on the rising edge of TCK following entry into the capture-DR controller state. Therefore, the first bit to be shifted out after selecting the bypass register is always a logic 0.

11.10.2.3 SAMPLE/PRELOAD (100). The SAMPLE/PRELOAD instruction provides two separate functions. First, it provides a means to obtain a snapshot of system data and control signals. The snapshot occurs on the rising edge of TCK in the capture-DR controller state. The data can be observed by shifting it transparently through the boundary scan register. In a normal system configuration many signals require external pull-ups to insure proper system operation. Consequently, the same is true for the SAMPLE/PRELOAD functionality. The data latched into the boundary scan register during capture-DR may not match the drive state of the package signal if the system-required pull-ups are not present within the test environment.

NOTE

Since there is no internal synchronization between the IEEE 1149.1 clock (TCK) and the system clock (CLK), the user must provide some form of external synchronization to achieve meaningful results.

The second function of the SAMPLE/PRELOAD instruction is to initialize the boundary scan register output cells prior to selection of the EXTEST instruction. This insures that known data will appear on the outputs when entering the EXTEST instruction. Since the MC88110 has only input-only and fully bi-directional signals, entering EXTEST does not require initialization of the boundary scan register. During TAP reset, bi-directional signals preload the output control cell with output disable. In the SAMPLE/PRELOAD instruction, system logic is independent of the TAP.

11.10.2.4 CLAMP (100). The CLAMP instruction is not included in the IEEE 1149.1 standard, but it is provided as a manufacturer's optional public instruction in order to prevent having to backdrive the output signals during some methods of circuit board testing. When the CLAMP instruction is invoked, the package signals will respond to the preconditioned values within the update latches of the boundary scan register, even though the bypass register is enabled as the test data register.

In-circuit testing can be facilitated by setting up the guarding signal conditions with use of the SAMPLE/PRELOAD or EXTEST instructions, and then as the MC88110 enters into the CLAMP instruction, the state and drive of all signals remain static until the instruction is disabled. A feature of the CLAMP instruction is that while the signals continue to supply the guarding inputs to the in-circuit test site, the bypass register is enabled and thus should minimize overall test time.

11.10.2.5 HI-Z (001). The HI-Z instruction is not included in the IEEE 1149.1 standard. It is provided as a manufacturer's optional public instruction in order to prevent having to backdrive the output signals during circuit board testing. When the HI-Z instruction is invoked, all output drivers are turned off (i.e., three-state). The instruction selects the bypass register.

11.10.2.6 EXTEST_PULLUP (010). The EXTEST_PULLUP instruction is not included in the IEEE 1149.1 standard, but is provided as a manufacturer's optional public instruction to aid in fault diagnoses during boundary scan testing of a circuit board. This instruction functions similarly to EXTEST, with the only difference being the presence of a weak pull-up device on all signals. The MC88110 is a CMOS design and therefore could suffer from a logically indeterminate input value if an input or bi-directional signal programmed as an input was inadvertently unconnected. The pull-up current will, given an appropriate charging delay, supply a deterministic logic 1 result on an open input. Note that heavily loaded nodes may require a charging delay greater than the two TCK periods needed to transition from the update-DR state to the capture-DR state. Two options are available: traverse into the run-test/idle state for extra TCK periods of charging delay or simply change the period of TCK leading up to the capture edge of the capture-DR state.

11.10.3 MC88110 Restrictions

The control afforded by the output enable signals using the boundary scan register and the EXTEST or CLAMP instructions requires a compatible circuit board test environment to avoid device-destructive configurations. The user must avoid situations in which the MC88110 output drivers are enabled into actively driven networks.

The MC88110 includes on-chip circuitry to detect the initial application of power to the device. The power-on reset (POR) signal is the output of this circuitry and is used to reset both the system and IEEE 1149.1 logic. POR is applied to the IEEE 1149.1 circuitry in order to avoid the possibility of bus contention during power-on. The time to complete device power-on is power supply dependent. The IEEE 1149.1 TAP controller, however, remains in the test-logic-reset state while POR is asserted. The TAP controller will not respond to user commands until POR is negated.

11.10.4 Non-IEEE 1149.1 Operation

In non-IEEE 1149.1 operation, there are two constraints that must be met. First, the test clock input does not include an internal pull-up resistor; therefore, it should not be left unconnected (this is in order to prevent mid-level inputs). The second constraint is that the IEEE 1149.1 test logic must be kept transparent to the system logic by forcing the TAP controller into the test-logic-reset controller state. During power-on, the POR signal forces the TAP controller into this state. However, to insure that the controller remains in the test-logic-reset state, several options are described below.

1. If TMS either remains unconnected or is connected to V_{CC}, then the TAP controller cannot leave the test-logic-reset state regardless of the state of the TCK pin.

2. $\overline{\text{TRST}}$ can be asserted either by connecting it to ground or by means of a logic network. Connecting $\overline{\text{TRST}}$ to the functional reset ($\overline{\text{RST}}$) signal and tying TCK either high or low also meets this requirement.

3. If $\overline{\text{TRST}}$ is asserted by a pulse signal, the controller will remain in the test-logic-reset state in the absence of a rising edge on the TCK pin when TMS is low.

APPENDIX A
BIT SCAN BIT DEFINITION

Table A-1. Bit Scan Bit Definition

Bit Number	Cell Type	Signal Name	Signal Type	Cntl Cell
0	IO.CELL	PSTAT2	Inout	PSTAT2_ctl
1	IO.CTL1	PSTAT2_ctl	—	—
2	IO.CELL	PSTAT1	Inout	PSTAT1_ctl
3	IO.CTL1	PSTAT1_ctl	—	—
4	IO.CELL	PSTAT0	Inout	PSTAT0_ctl
5	IO.CTL1	PSTAT0_ctl	—	—
6	IO.CELL	BP7	Inout	BP7_ctl
7	IO.CTL1	BP7_ctl	—	—
8	IO.CELL	BP6	Inout	BP6_ctl
9	IO.CTL1	BP6_ctl	—	—
10	IO.CELL	BP5	Inout	BP5_ctl
11	IO.CTL1	BP5_ctl	—	—
12	IO.CELL	BP4	Inout	BP4_ctl
13	IO.CTL1	BP4_ctl	—	—
14	IO.CELL	BP3	Inout	BP3_ctl
15	IO.CTL1	BP3_ctl	—	—
16	IO.CELL	BP2	Inout	BP2_ctl
17	IO.CTL1	BP2_ctl	—	—
18	IO.CELL	BP1	Inout	BP1_ctl
19	IO.CTL1	BP1_ctl	—	—
20	IO.CELL	BP0	Inout	BP0_ctl
21	IO.CTL1	BP0_ctl	—	—
22	IO.CELL	D63	Inout	D63_ctl
23	IO.CTL1	D63_ctl	—	—
24	IO.CELL	D62	Inout	D62_ctl
25	IO.CTL1	D62_ctl	—	—
26	IO.CELL	D61	Inout	D61_ctl
27	IO.CTL1	D61_ctl	—	—
28	IO.CELL	D60	Inout	D60_ctl

A

Table A-1. Bit Scan Bit Definition

Bit Number	Cell Type	Signal Name	Signal Type	Cntl Cell
29	IO.CTL1	D60_ctl	—	—
30	IO.CELL	D59	Inout	D59_ctl
31	IO.CTL1	D59_ctl	—	—
32	IO.CELL	D58	Inout	D58_ctl
33	IO.CTL1	D58_ctl	—	—
34	IO.CELL	D57	Inout	D57_ctl
35	IO.CTL1	D57_ctl	—	—
36	IO.CELL	D56	Inout	D56_ctl
37	IO.CTL1	D56_ctl	—	—
38	IO.CELL	D55	Inout	D55_ctl
39	IO.CTL1	D55_ctl	—	—
40	IO.CELL	D54	Inout	D54_ctl
41	IO.CTL1	D54_ctl	—	—
42	IO.CELL	D53	Inout	D53_ctl
43	IO.CTL1	D53_ctl	—	—
44	IO.CELL	D52	Inout	D52_ctl
45	IO.CTL1	D52_ctl	—	—
46	IO.CELL	D51	Inout	D51_ctl
47	IO.CTL1	D51_ctl	—	—
48	IO.CELL	D50	Inout	D50_ctl
49	IO.CTL1	D50_ctl	—	—
50	IO.CELL	D49	Inout	D49_ctl
51	IO.CTL1	D49_ctl	—	—
52	IO.CELL	D48	Inout	D48_ctl
53	IO.CTL1	D48_ctl	—	—
54	IO.CELL	D47	Inout	D47_ctl
55	IO.CTL1	D47_ctl	—	—
56	IO.CELL	D46	Inout	D46_ctl
57	IO.CTL1	D46_ctl	—	—
58	IO.CELL	D45	Inout	D45_ctl
59	IO.CTL1	D45_ctl	—	—
60	IO.CELL	D44	Inout	D44_ctl
61	IO.CTL1	D44_ctl	—	—
62	IO.CELL	D43	Inout	D43_ctl
63	IO.CTL1	D43_ctl	—	—
64	IO.CELL	D42	Inout	D42_ctl
65	IO.CTL1	D42_ctl	—	—

A

Table A-1. Bit Scan Bit Definition

Bit Number	Cell Type	Signal Name	Signal Type	Cntl Cell
66	IO.CELL	D41	Inout	D41_ctl
67	IO.CTL1	D41_ctl	—	—
68	IO.CELL	D40	Inout	D40_ctl
69	IO.CTL1	D40_ctl	—	—
70	IO.CELL	D39	Inout	D39_ctl
71	IO.CTL1	D39_ctl	—	—
72	IO.CELL	D38	Inout	D38_ctl
73	IO.CTL1	D38_ctl	—	—
74	IO.CELL	D37	Inout	D37_ctl
75	IO.CTL1	D37_ctl	—	—
76	IO.CELL	D36	Inout	D36_ctl
77	IO.CTL1	D36_ctl	—	—
78	IO.CELL	D35	Inout	D35_ctl
79	IO.CTL1	D35_ctl	—	—
80	IO.CELL	D34	Inout	D34_ctl
81	IO.CTL1	D34_ctl	—	—
82	IO.CELL	D33	Inout	D33_ctl
83	IO.CTL1	D33_ctl	—	—
84	IO.CELL	D32	Inout	D32_ctl
85	IO.CTL1	D32_ctl	—	—
86	IO.CELL	D31	Inout	D31_ctl
87	IO.CTL1	D31_ctl	—	—
88	IO.CELL	D30	Inout	D30_ctl
89	IO.CTL1	D30_ctl	—	—
90	IO.CELL	D29	Inout	D29_ctl
91	IO.CTL1	D29_ctl	—	—
92	IO.CELL	D28	Inout	D28_ctl
93	IO.CTL1	D28_ctl	—	—
94	IO.CELL	D27	Inout	D27_ctl
95	IO.CTL1	D27_ctl	—	—
96	IO.CELL	D26	Inout	D26_ctl
97	IO.CTL1	D26_ctl	—	—
98	IO.CELL	D25	Inout	D25_ctl
99	IO.CTL1	D25_ctl	—	—
100	IO.CELL	D24	Inout	D24_ctl
101	IO.CTL1	D24_ctl	—	—
102	IO.CELL	D23	Inout	D23_ctl

A

Table A-1. Bit Scan Bit Definition

Bit Number	Cell Type	Signal Name	Signal Type	Cntl Cell
103	IO.CTL1	D23_ctl	—	—
104	IO.CELL	D22	Inout	D22_ctl
105	IO.CTL1	D22_ctl	—	—
106	IO.CELL	D21	Inout	D21_ctl
107	IO.CTL1	D21_ctl	—	—
108	IO.CELL	D20	Inout	D20_ctl
109	IO.CTL1	D20_ctl	—	—
110	IO.CELL	D19	Inout	D19_ctl
111	IO.CTL1	D19_ctl	—	—
112	IO.CELL	D18	Inout	D18_ctl
113	IO.CTL1	D18_ctl	—	—
114	IO.CELL	D17	Inout	D17_ctl
115	IO.CTL1	D17_ctl	—	—
116	IO.CELL	D16	Inout	D16_ctl
117	IO.CTL1	D16_ctl	—	—
118	IO.CELL	D15	Inout	D15_ctl
119	IO.CTL1	D15_ctl	—	—
120	IO.CELL	D14	Inout	D14_ctl
121	IO.CTL1	D14_ctl	—	—
122	IO.CELL	D13	Inout	D13_ctl
123	IO.CTL1	D13_ctl	—	—
124	IO.CELL	D12	Inout	D12_ctl
125	IO.CTL1	D12_ctl	—	—
126	IO.CELL	D11	Inout	D11_ctl
127	IO.CTL1	D11_ctl	—	—
128	IO.CELL	D10	Inout	D10_ctl
129	IO.CTL1	D10_ctl	—	—
130	IO.CELL	D9	Inout	D9_ctl
131	IO.CTL1	D9_ctl	—	—
132	IO.CELL	D8	Inout	D8_ctl
133	IO.CTL1	D8_ctl	—	—
134	IO.CELL	D7	Inout	D7_ctl
135	IO.CTL1	D7_ctl	—	—
136	IO.CELL	D6	Inout	D6_ctl
137	IO.CTL1	D6_ctl	—	—
138	IO.CELL	D5	Inout	D5_ctl
139	IO.CTL1	D5_ctl	—	—

A

Table A-1. Bit Scan Bit Definition

Bit Number	Cell Type	Signal Name	Signal Type	Cntl Cell
140	IO.CELL	D4	Inout	D4_ctl
141	IO.CTL1	D4_ctl	—	—
142	IO.CELL	D3	Inout	D3_ctl
143	IO.CTL1	D3_ctl	—	—
144	IO.CELL	D2	Inout	D2_ctl
145	IO.CTL1	D2_ctl	—	—
146	IO.CELL	D1	Inout	D1_ctl
147	IO.CTL1	D1_ctl	—	—
148	IO.CELL	D0	Inout	D0_ctl
149	IO.CTL1	D0_ctl	—	—
150	IO.CELL	BPE_B	Inout	BPE_B_ctl
151	IO.CTL1	BPE_B_ctl	—	—
152	IO.CELL	DBB_B	Inout	DBB_B_ctl
153	IO.CTL1	DBB_B_ctl	—	—
154	IO.CELL	BR_B	Inout	BR_B_ctl
155	IO.CTL1	BR_B_ctl	—	—
156	IO.CELL	G2OUT	Inout	G2OUT_ctl
157	IO.CTL1	G2OUT_ctl	—	—
158	I.CELL	CLK	Input	—
159	I.CELL	BG_B	Input	—
160	I.CELL	DBG_B	Input	—
161	I.CELL	AACK_B	Input	—
162	I.CELL	PTA_B	Input	—
163	I.CELL	TA_B	Input	—
164	I.CELL	TRTRY_B	Input	—
165	I.CELL	TEA_B	Input	—
166	I.CELL	ARTRY_B	Input	—
167	I.CELL	SHD_B	Input	—
168	I.CELL	SR_B	Input	—
169	IO.CELL	TS_B	Inout	TS_B_ctl
170	IO.CTL1	TS_B_ctl	—	—
171	IO.CELL	MC_B	Inout	MC_B_ctl
172	IO.CTL1	MC_B_ctl	—	—
173	IO.CELL	GBL_B	Inout	GBL_B_ctl
174	IO.CTL1	GBL_B_ctl	—	—
175	IO.CELL	INV_B	Inout	INV_B_ctl
176	IO.CTL1	INV_B_ctl	—	—

A

Table A-1. Bit Scan Bit Definition

Bit Number	Cell Type	Signal Name	Signal Type	Cntl Cell
177	IO.CELL	SSTAT_B0	Inout	SSTAT_B0_ctl
178	IO.CTL1	SSTAT_B0_ctl	—	—
179	IO.CELL	SSTAT_B1	Inout	SSTAT_B1_ctl
180	IO.CTL1	SSTAT_B1_ctl	—	—
181	IO.CELL	ABB_B	Inout	ABB_B_ctl
182	IO.CTL1	ABB_B_ctl	—	—
183	IO.CELL	A0	Inout	A0_ctl
184	IO.CTL1	A0_ctl	—	—
185	IO.CELL	A1	Inout	A1_ctl
186	IO.CTL1	A1_ctl	—	—
187	IO.CELL	A2	Inout	A2_ctl
188	IO.CTL1	A2_ctl	—	—
189	IO.CELL	A3	Inout	A3_ctl
190	IO.CTL1	A3_ctl	—	—
191	IO.CELL	A4	Inout	A4_ctl
192	IO.CTL1	A4_ctl	—	—
193	IO.CELL	A5	Inout	A5_ctl
194	IO.CTL1	A5_ctl	—	—
195	IO.CELL	A6	Inout	A6_ctl
196	IO.CTL1	A6_ctl	—	—
197	IO.CELL	A7	Inout	A7_ctl
198	IO.CTL1	A7_ctl	—	—
199	IO.CELL	A8	Inout	A8_ctl
200	IO.CTL1	A8_ctl	—	—
201	IO.CELL	A9	Inout	A9_ctl
202	IO.CTL1	A9_ctl	—	—
203	IO.CELL	A10	Inout	A10_ctl
204	IO.CTL1	A10_ctl	—	—
205	IO.CELL	A11	Inout	A11_ctl
206	IO.CTL1	A11_ctl	—	—
207	IO.CELL	A12	Inout	A12_ctl
208	IO.CTL1	A12_ctl	—	—
209	IO.CELL	A13	Inout	A13_ctl
210	IO.CTL1	A13_ctl	—	—
211	IO.CELL	A14	Inout	A14_ctl
212	IO.CTL1	A14_ctl	—	—
213	IO.CELL	A15	Inout	A15_ctl

A

Table A-1. Bit Scan Bit Definition

Bit Number	Cell Type	Signal Name	Signal Type	Cntl Cell
214	IO.CTL1	A15_ctl	—	—
215	IO.CELL	A16	Inout	A16_ctl
216	IO.CTL1	A16_ctl	—	—
217	IO.CELL	A17	Inout	A17_ctl
218	IO.CTL1	A17_ctl	—	—
219	IO.CELL	A18	Inout	A18_ctl
220	IO.CTL1	A18_ctl	—	—
221	IO.CELL	A19	Inout	A19_ctl
222	IO.CTL1	A19_ctl	—	—
223	IO.CELL	A20	Inout	A20_ctl
224	IO.CTL1	A20_ctl	—	—
225	IO.CELL	A21	Inout	A21_ctl
226	IO.CTL1	A21_ctl	—	—
227	IO.CELL	A22	Inout	A22_ctl
228	IO.CTL1	A22_ctl	—	—
229	IO.CELL	A23	Inout	A23_ctl
230	IO.CTL1	A23_ctl	—	—
231	IO.CELL	A24	Inout	A24_ctl
232	IO.CTL1	A24_ctl	—	—
233	IO.CELL	A25	Inout	A25_ctl
234	IO.CTL1	A25_ctl	—	—
235	IO.CELL	A26	Inout	A26_ctl
236	IO.CTL1	A26_ctl	—	—
237	IO.CELL	A27	Inout	A27_ctl
238	IO.CTL1	A27_ctl	—	—
239	IO.CELL	A28	Inout	A28_ctl
240	IO.CTL1	A28_ctl	—	—
241	IO.CELL	A29	Inout	A29_ctl
242	IO.CTL1	A29_ctl	—	—
243	IO.CELL	A30	Inout	A30_ctl
244	IO.CTL1	A30_ctl	—	—
245	IO.CELL	A31	Inout	A31_ctl
246	IO.CTL1	A31_ctl	—	—
247	IO.CELL	RW_B	Inout	RW_B_ctl
248	IO.CTL1	RW_B_ctl	—	—
249	IO.CELL	TBST_B	Inout	TBST_B_ctl
250	IO.CTL1	TBST_B_ctl	—	—

A

Table A-1. Bit Scan Bit Definition

Bit Number	Cell Type	Signal Name	Signal Type	Cntl Cell
251	IO.CELL	TSIZ0	Inout	TSIZ0_ctl
252	IO.CTL1	TSIZ0_ctl	—	—
253	IO.CELL	TSIZ1	Inout	TSIZ1_ctl
254	IO.CTL1	TSIZ1_ctl	—	—
255	IO.CELL	LK_B	Inout	LK_B_ctl
256	IO.CTL1	LK_B_ctl	—	—
257	IO.CELL	UPA_B0	Inout	UPA_B0_ctl
258	IO.CTL1	UPA_B0_ctl	—	—
259	IO.CELL	UPA_B1	Inout	UPA_B1_ctl
260	IO.CTL1	UPA_B1_ctl	—	—
261	IO.CELL	CI_B	Inout	CI_B_ctl
262	IO.CTL1	CI_B_ctl	—	—
263	IO.CELL	WT_B	Inout	WT_B_ctl
264	IO.CTL1	WT_B_ctl	—	—
265	IO.CELL	TC0	Inout	TC0_ctl
266	IO.CTL1	TC0_ctl	—	—
267	IO.CELL	TC1	Inout	TC1_ctl
268	IO.CTL1	TC1_ctl	—	—
269	IO.CELL	TC2	Inout	TC2_ctl
270	IO.CTL1	TC2_ctl	—	—
271	IO.CELL	TC3	Inout	TC3_ctl
272	IO.CTL1	TC3_ctl	—	—
273	IO.CELL	CLINE	Inout	CLINE_ctl
274	IO.CTL1	CLINE_ctl	—	—
275	I.CELL	DBUG_B	Input	—
276	I.CELL	RST_B	Input	—
277	I.CELL	NMI_B	Input	—
278	I.CELL	INT_B	Input	—

A

INDEX

I

I

I

I

I

I

I

MOTOROLA