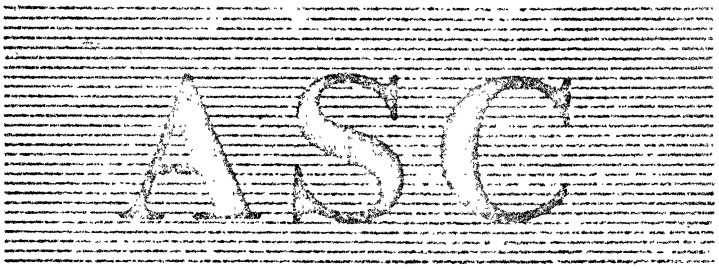
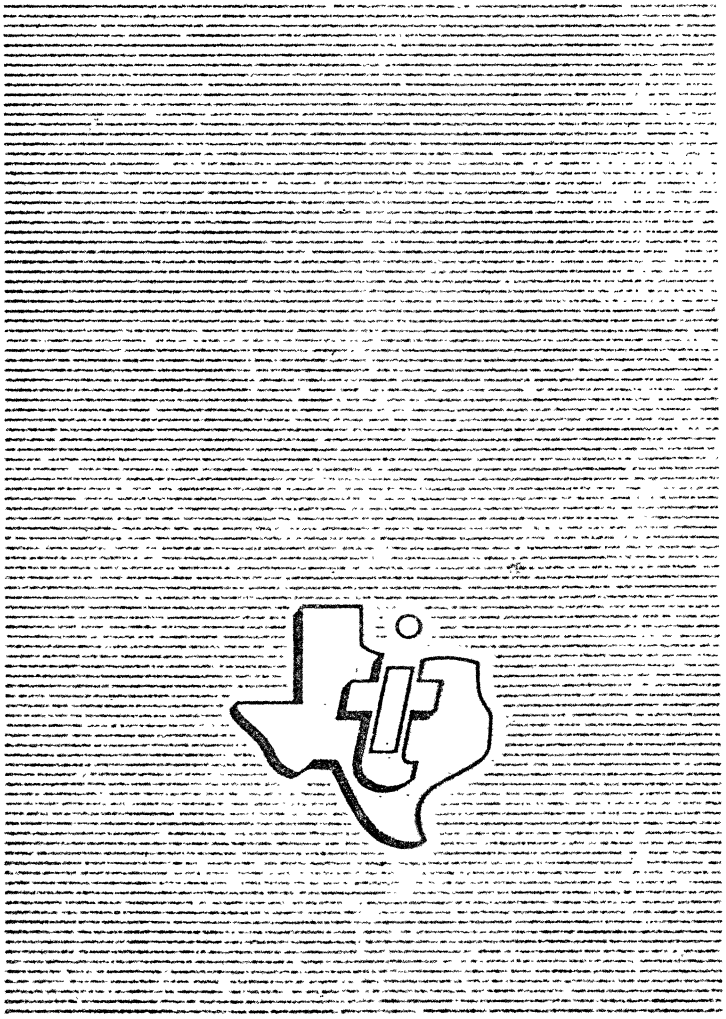
Central No. HW127

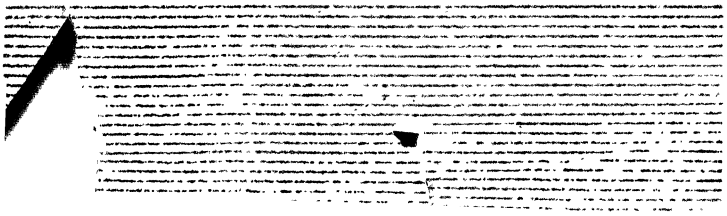


DESCRIPTION OF
THE ASC SYSTEM HARDWARE

October 1970



TEXAS INSTRUMENTS
INCORPORATED



PREFACE

This document describes the ASC System hardware as required by the system programming staff. It is also directed to all personnel associated with designing and configuring ASC Systems to be used as a primary reference source.

Section A describes the Memory System of the ASC including the Memory Control Unit, the Memory Modules, and the expansion features being incorporated into the system. Section B describes the Central Processor including the Memory Buffer Unit, the Instruction Processing Unit, and the Arithmetic Unit. This section also provides the CP instruction set, the timing analysis, and examples of vector instruction applications. Section C describes the Peripheral Processor including the Peripheral Processing Unit, the communication registers, the timing analysis, and the PP instruction set. Section D describes the Data Channel Units. Section E describes the ASC System operating panels, and the associated procedures. Section F describes the peripheral devices used with the system. Section G describes the maintenance provisions being designed into the system.

This document is designed to allow each part to be used alone. The overall list of effective pages following the Preface provides the effective date of the A page of each section. Changes, modifications, or additions will be issued as changed or added pages with the date of the change in the upper right hand corner of the changed page.

CONTENTS

INTRODUCTION

TABLE OF CONTENTS

GENERAL ORGANIZATION OF ASC

The ASC is an advanced computer designed especially for high-volume processing of well-ordered data in a multiprogramming environment. To achieve its extremely high processing speed, the computer utilizes a pipeline arithmetic section and a pipeline instruction section.

Organization of the ASC is illustrated in Figure 1. A Peripheral Processor links input/output equipment to a Central Processor and a Central Memory. The input/output equipment available for the system includes CRT keyboard/display units, magnetic tapes, magnetic discs, card equipment, and line printers.

The Peripheral Processor contains eight independent computers which control input/output devices and schedule work for the Central Processor.

The Central Processor provides the major execution facility of the system. The interface between Central Processor (CP) and Peripheral Processor (PP) consists of control communication links whereby the CP signals completion of jobs or its availability for other jobs and the PP initiates new jobs. All data and instruction for the CP are obtained through the Central Memory.

The Central Memory consists of high-speed semiconductor memory modules which have full cycle times of 160 nsec for 256-bit words. Since the ASC computer is a 32-bit/word computer, each memory cycle has access to eight computer words. The Central Memory interfaces with the CP and the Peripheral Processor.

One ASC configuration is shown in Figure 2. This diagram indicates four memory units, each containing 16K words of storage; thus, the

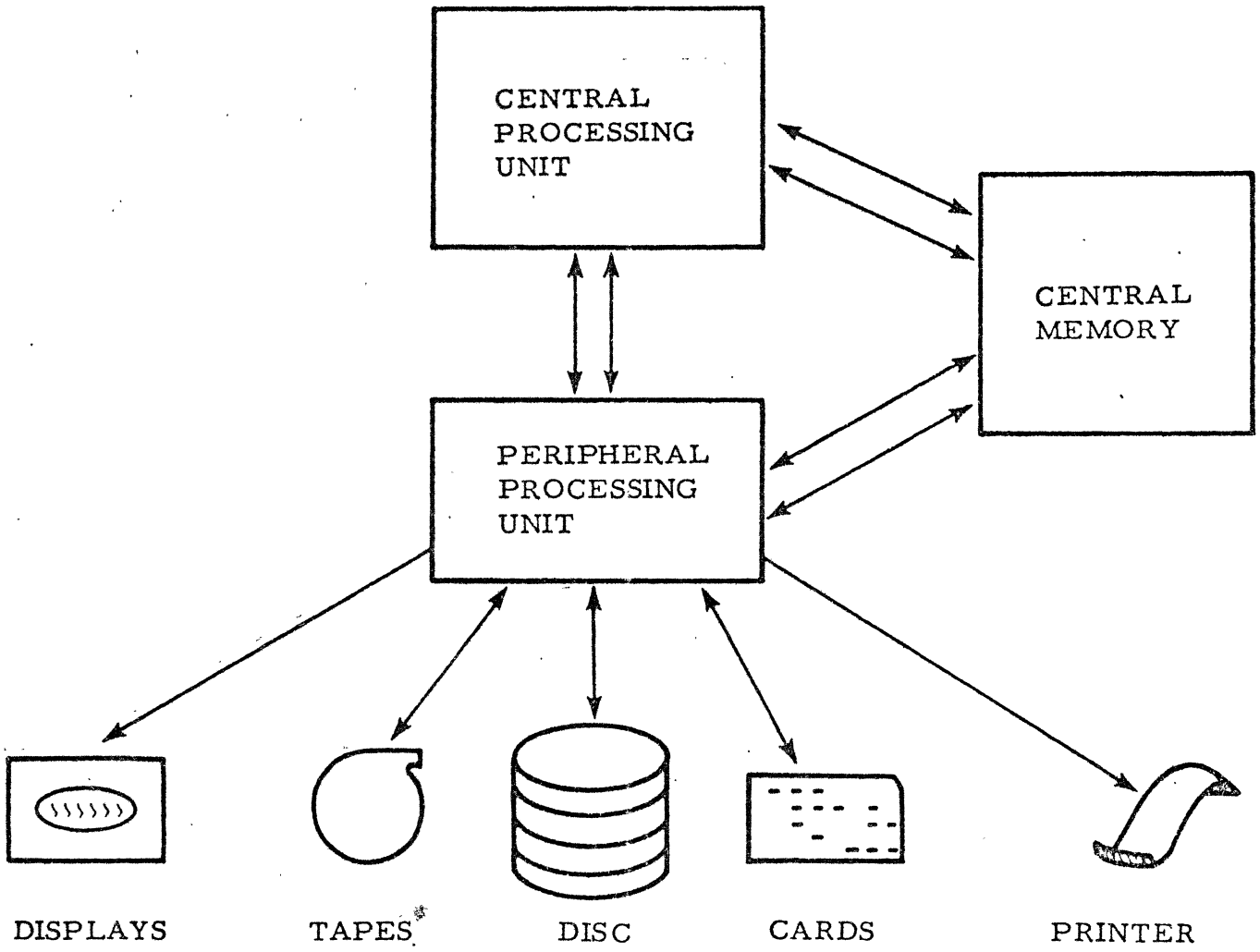


Figure 1. ASC Organization

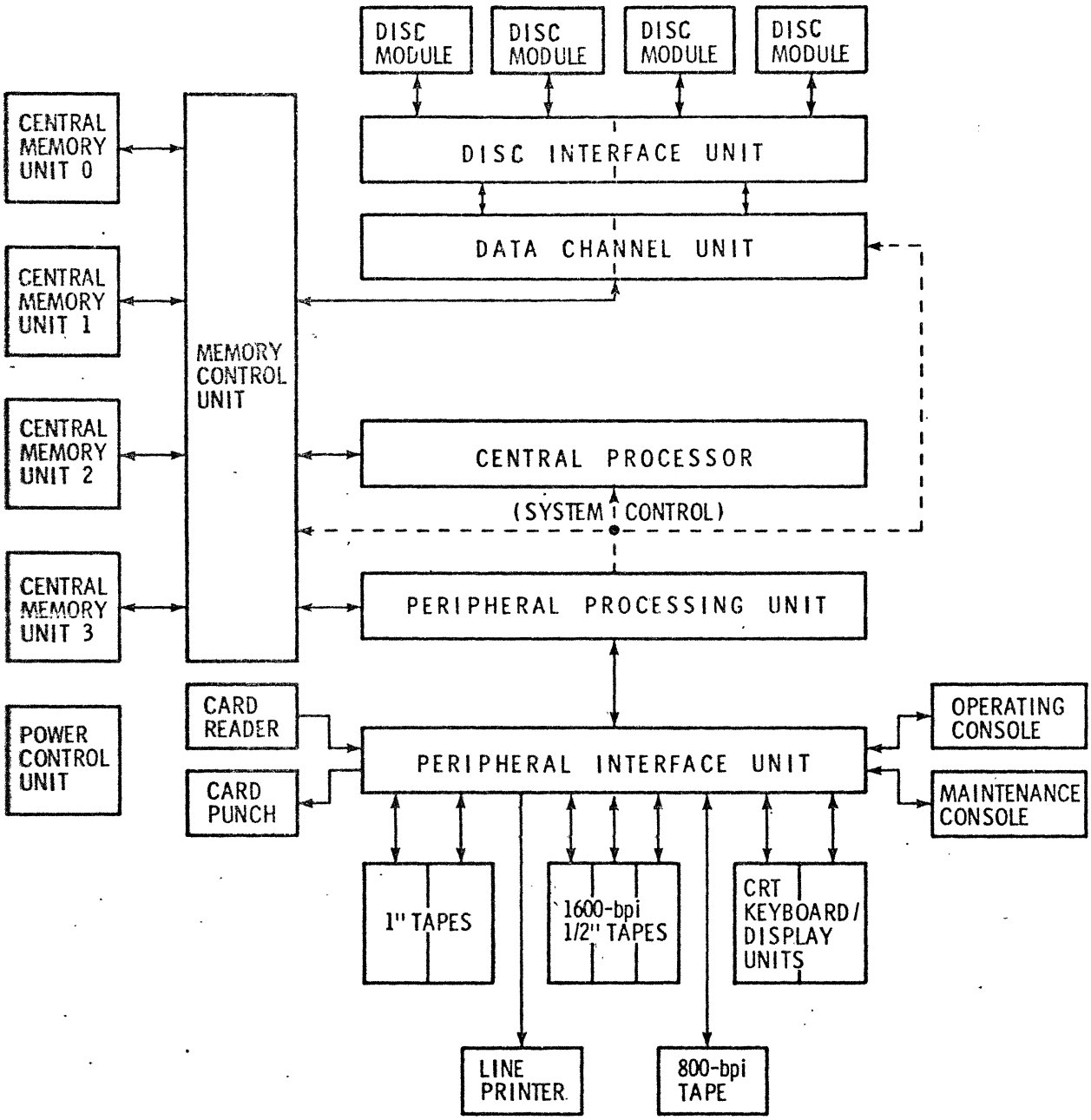


Figure 2. ASC System (Prototype)

basic memory size is 65K words. These four modules are controlled by a memory control unit which multiplexes the memory units to the Central Processor, Peripheral Processor, or data channel unit which is a special high-speed device capable of sustaining the high data transfer rates required by the disc system. A significant feature of the ASC system is its utilization of a large disc file with word transfer rates of 10^6/sec between it and the Central Memory. The disc file consists of four modules containing 25,000,000 words each.

USE OF PIPELINE CONCEPT

The pipeline concept being exploited in the Central Processor is illustrated by the example in Figure 3. This example shows a "pipe" which performs an operation consisting of three separate and distinct steps. This

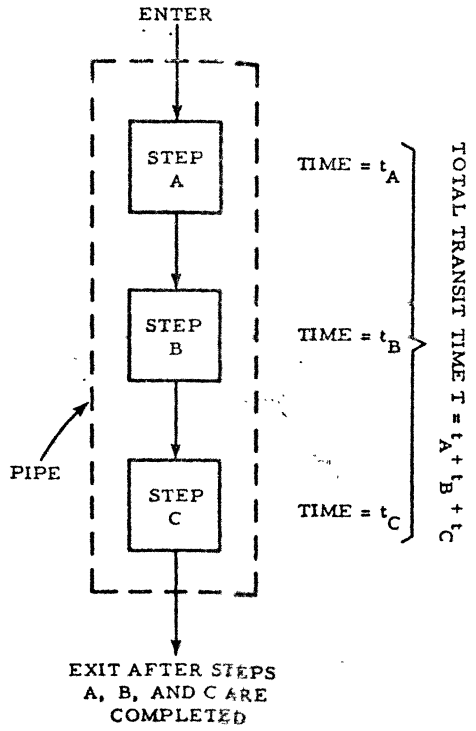


Figure 3. Pipeline Concept

operation can be performed on an operand by entering it in the pipe and collecting the result at the exit after transit time T has elapsed. Thus, the time required to perform an operation is the sum of the individual step times t_A, t_B, and t_C. If the steps are separate and distinct as stated, then the average operation time can be decreased by entering operands into the pipe so that different operands are at steps A, B, and C simultaneously. If a long series of operands are routed through the pipe so that the "fill-up" and "empty" times are negligible, the average time required for an operation will be

$$\frac{t_A + t_B + t_C}{3}$$

The ASC arithmetic unit is constructed from a number of "sections" (Figure 4), each of which can perform a separate arithmetic or logical operation in the same manner as the steps in the pipe of the previous example. These sections are connected in "pipe" fashion to generate a pipe for performing each instruction in the CP. Each section can be connected to any of the other sections, as required, to construct a pipe for executing any particular instruction. Figure 4 shows sections 1, 2, and 8 connected in a pipe by the solid line which may be the configuration required to perform an instruction. The dotted line connecting sections 1, 4, 6, and 8 illustrates a configuration which may be required to execute another instruction. In this fashion, the sections of the arithmetic unit are configured as required to execute the ASC instructions. The proper configuration of the arithmetic unit is established when the instruction and its operand are at the entry to the pipeline.

This pipeline concept is used in the design of the ASC because of its inherent ability to achieve high-speed operations on large volumes of well-ordered data. If the data are arranged so that a large number of identical operations are required in sequence, the pipeline can be filled, achieving an average operation speed equal to the time required for only one section of the pipe. This well-ordered type of data is represented by vector or array processing. For example, consider the vectors

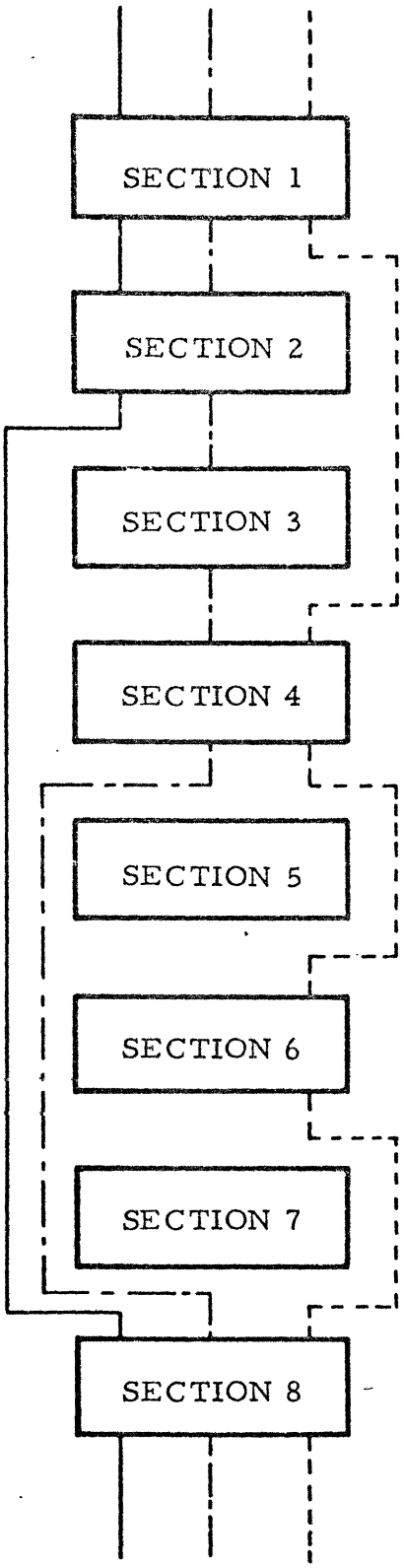


Figure 4. Sections of Arithmetic Unit Connected in "Pipe" Fashion

$$A = a_1 + a_2 + a_3 + \ldots . a_i$$

$$B = b_1 + b_2 + b_3 + \ldots . b_i$$

A vector addition of A + B would result in the vector C where

$$c_1 = a_1 + b_1$$

$$c_2 = a_2 + b_2$$

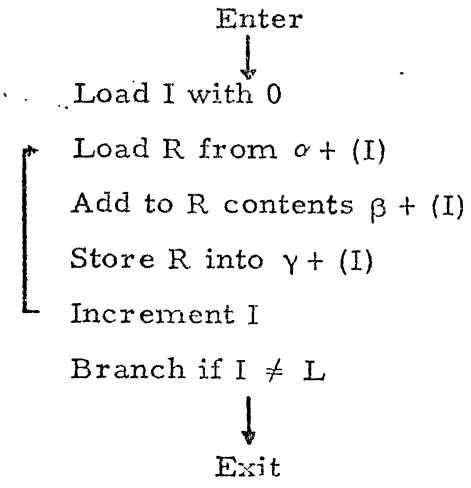
$$c_3 = a_3 + b_3$$

$$\vdots$$

$$c_i = a_i + b_i$$

The ASC instruction set contains vector and matrix instructions to perform such operations so that only one instruction is required to accomplish this operation on any length of vector. A conventional computer would require a series of instructions to achieve this same operation. This is illustrated in the following example:

- Conditions
  - — array A is stored in consecutive locations beginning at α
  - — array B is stored in consecutive locations beginning at β
  - — array C is to be stored in consecutive locations beginning at γ
  - — each array consists of L elements
- The subroutine required for a conventional machine is



The time required to accomplish the function of each section in the ASC pipeline is 60 nsec; thus, for processing vector or array instructions, the average time per element is only 60 nsec. The CP interface with the Central Memory is designed to sustain this rate of data processing so that the pipeline can be utilized to its fullest extent.

The CP interface with the Central Memory is shown schematically in Figure 5. There is one 256-bit data transfer bus between the CP and CM, which is shared by four CP storage buffers. Instructions are buffered in two storage files (IB and I), each containing eight computer words. One operand vector is buffered in two storage files (XB and X), each containing eight computer words, and the other operand vector is buffered in another set of storage files (YB and B). The resultant vector from the arithmetic unit is buffered in storage files ZB and Z in the same manner.

In addition to the pipeline construction of the arithmetic unit, the ASC employs a pipeline at the instruction processing level. Up to 12 instructions are in this pipeline at any time, so streams of instructions ready for execution are supplied at the exit of the instruction processing pipe in somewhat the same manner as streams of vector elements enter and exit from the arithmetic-unit pipe.

The Peripheral Processor provides communication with I/O devices, functions as system monitor, and fulfills job requests which do not require high arithmetic capability. Elements of the PP, shown in Figure 6, include one arithmetic unit which is shared by eight virtual processors, one of which is designated as the system monitor. Functions of the system monitor include assignment of system-control parameters, assignment of programs to each of the seven slave virtual processors, assignment of CP programs, and monitoring of the progress of all programs including the CP program.

The virtual processors communicate with I/O devices, Central Processor, data channel, and other system components via 64 Communication Registers (CR) which are 32 bits in length and can be set or read by the virtual processor or an external device.

Associated with the virtual processor is a Read-Only Memory (ROM) containing fixed programs which are executed by the virtual processors. These programs are stored in the ROM because they are frequently used and require fast access.

Each virtual processor has a single-word buffer which acts as memory address register and memory data register for that processor. Central Memory access requests from these single-word buffers are granted on a priority basis.

The virtual processors are operative and share the arithmetic unit as programmed by the system monitor. This sharing is accomplished by dividing the time into 16 time slots represented by the segments shown on the wheel in Figure 7. Time slots are assigned to the virtual processors according to their needs.

SYSTEM CONFIGURATION AND HARDWARE FEATURES

The recommended physical configuration of the ASC requires Approximately 4000 sq. ft. Four airconditioning units, furnished as part of the ASC, supply cooling air to system components requiring special cooling.

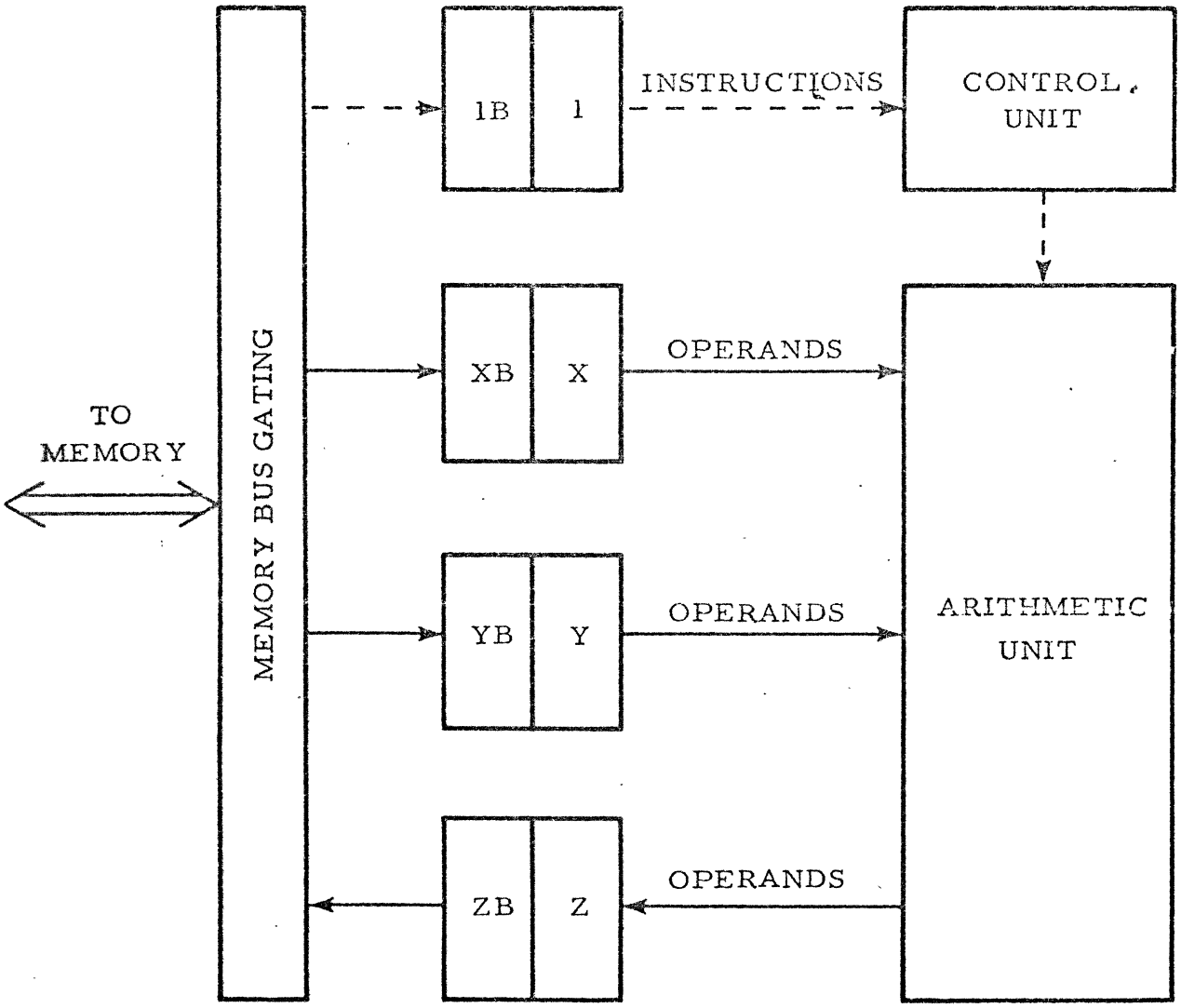


Figure 5. Interfacing of Central Processor and Central Memory

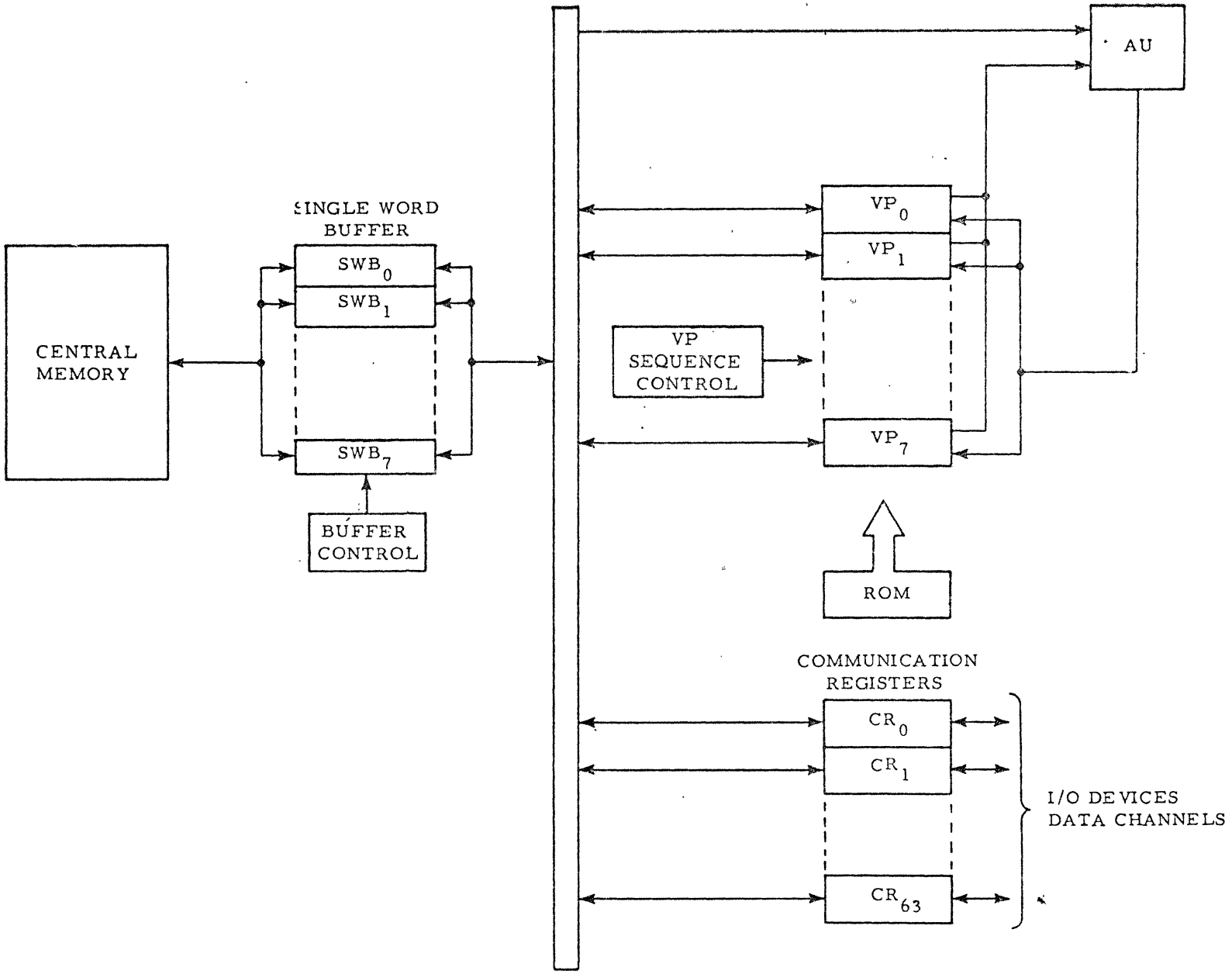
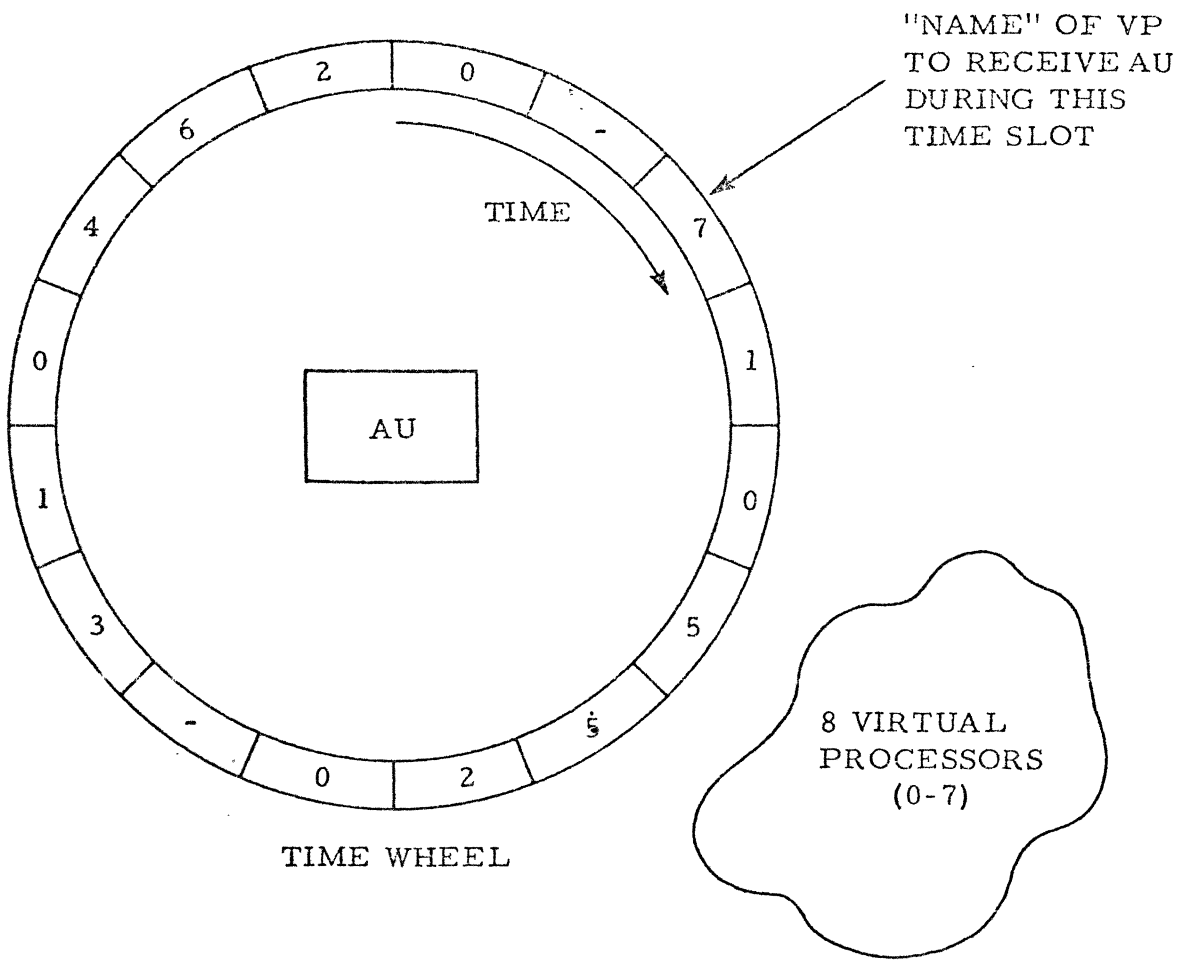


Figure 6. ASC Peripheral Processor



1 ARITHMETIC UNIT

8 "VIRTUAL" PROCESSORS WHICH SHARE AU

Figure 7. Time Slots Permitting Virtual Processors To Share Arithmetic Unit

Normal airconditioning is also required to maintain a comfortable working area.

The extremely high speed of ASC operation is possible because of the advanced logic components which implement the system. These logic circuits are emitter-coupled integrated circuits having gate speeds of approximately 2 nsec.

ASC hardware features: a high-speed semiconductor memory; a direct-access fixed-head disc auxiliary storage system; Peripheral Processor which provides system control and external-internal communications; a rapid Central Processor for data manipulation, with the feature of hardware logic for vector-matrix operations; automatic, rapid, context switching for efficient multiprogramming; high-speed peripheral input/output devices; and remote on-line graphic terminals.

MEMORY SYSTEM

SECTION A

MEMORY SYSTEM
SECTION A
CONTENTS

The Central Memory (CM) of the ASC system is configured from three basic units: the Memory Control Unit (MCU), the memory port Expander/Multiplexer (EX) and the selected Memory Units. Figure 1 illustrates a typical CM configuration.

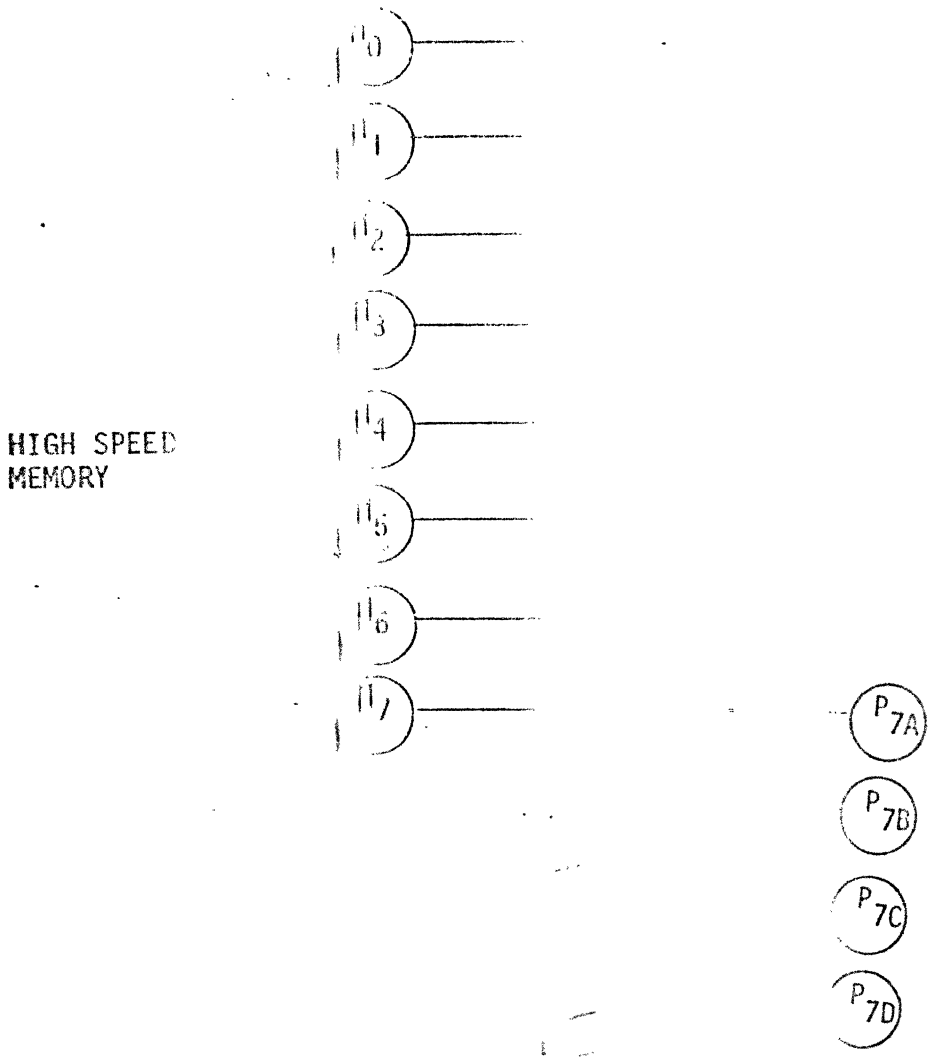
THE MEMORY CONTROL UNIT

The MCU is organized as a two-way 256-bit/channel parallel access traffic net between eight independent processor ports and nine memory buses, with each processor having full accessibility to all memories.

The nine memory buses are organized to provide eight-way interleaving for the first eight buses with the ninth buses reserved for bulk storage. However, a patchboard is provided within the MCU to facilitate addressing patterns from no interleaving to eight-way interleaving.

The MCU provides the facilities for controlling access from the eight processor ports to a CM having a 24-bit address space (16 million words). In addition, each port contains the necessary hardware for performing the MAP and PROTECT address processing functions (described subsequently). Conflicts at the memory buses are resolved on either a fixed priority bases (i.e., each processor port is assigned a relative priority) or a distributed priority basis (i.e., all processor ports are assigned equal priority).

The unit is asynchronously designed to operate independently of cable delays, processor clock rates, and memory unit access and cycle times; however, these times can affect the memory bandwidth. For comparative purposes, the total bandwidth of CM is computed as BW_{CM} = No. of words/cycle X No. of Independent memories/Memory Cycle Time, and the bandwidth provided each processor port is BW_p = No. of Words/Cycle /2 X Processor Clock Period.


HIGH SPEED
MEMORY
H_0
H_1
H_2
H_3
H_4
H_5
H_6
H_7
P_{7A}
P_{7B}
P_{7C}
P_{7D}
: EXAMPLE

The Memory MAP provides for dynamic address relocation of Central Memory of the block (i.e., page) level. Contiguous virtual page addresses from the processors are transformed into discontiguous actual page addresses for more efficient use of CM resources. The MAP is physically a set of up to 64 eight-bit (page address) registers accessed via the virtual page address of an individual request. The contents of the register addressed replaces the most significant bits of the virtual address to form the actual page address. Figure 2 illustrates the Mapping procedure. The size of each "page" is a function of the size of CM being mapped. The minimum page size is 4K words with the maximum being 256K for full 24 bit addressing.

The PROTECT facilities consist of three 24-bit bounds register-pairs for defining the upper and lower bound of a protected CM segment. The MCU compares the address of each processor request to the contents of one of the bounds register-pairs selected via a two-bit code developed by the processor. For example, the three central processor categories are READ, WRITE, and EXECUTE. A request pointed toward a protected CM area is denied access to that location and the processor is notified of the attempted violation.

The bounds register-pairs can be used to define a variety of CM protection functions. Figure 3 illustrates a typical CM arrangement for the central processor port. Note that segments as small as 16 word groups may be defined.

The MAP provides an additional protection feature. If a processor utilizing the MAP feature accesses a MAP register containing actual page 0, the request is denied access to CM and the processor is notified. Thus, the "0" code signifies that the requested page is not resident in the physical central memory.

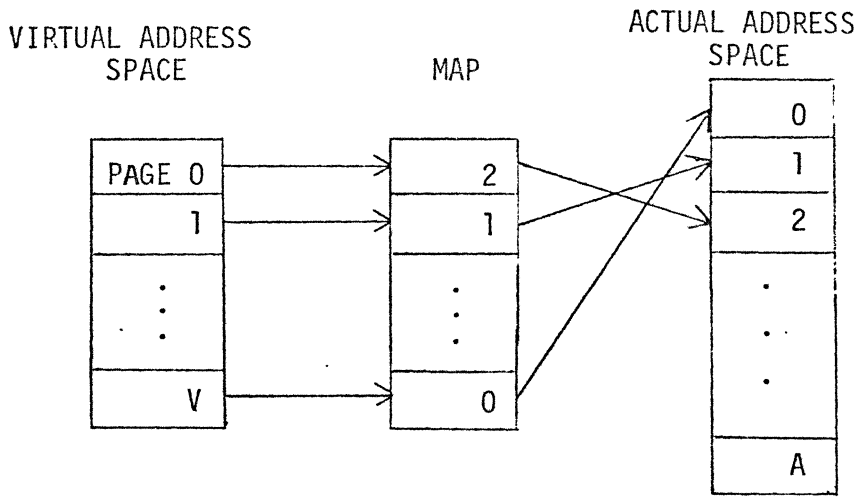


FIGURE 2: MEMORY MAPPING

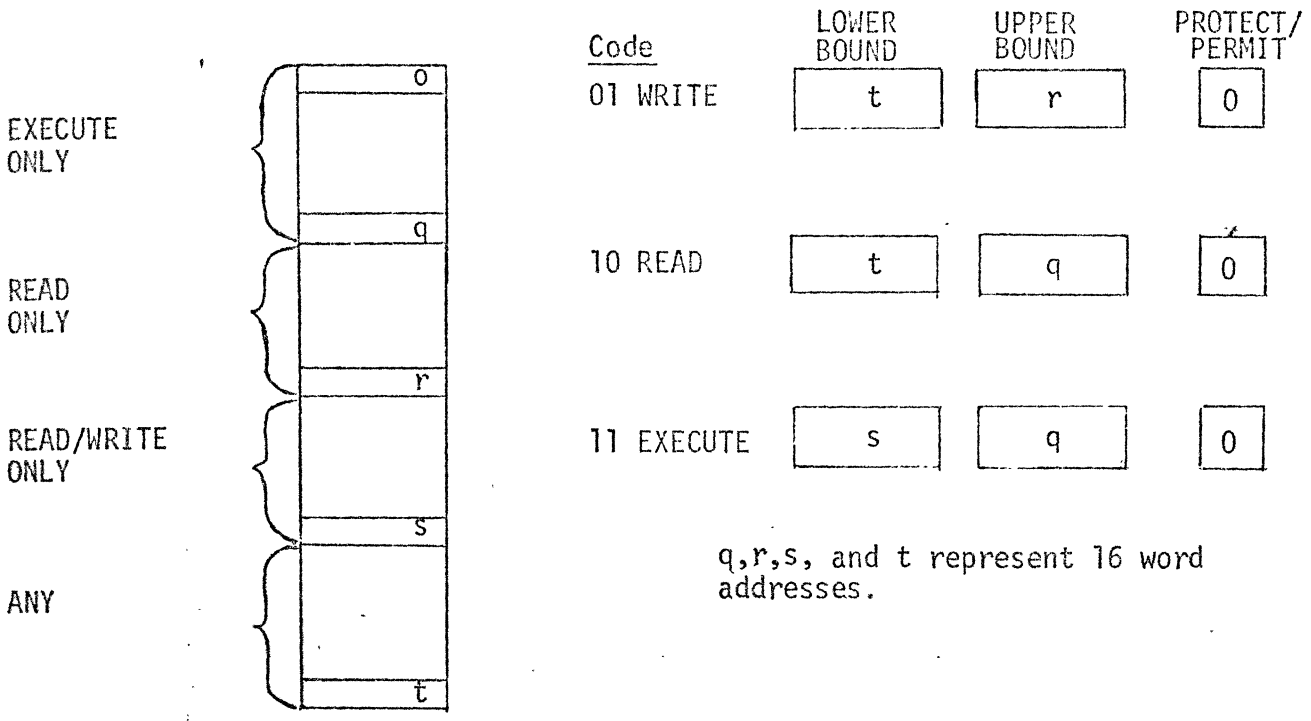
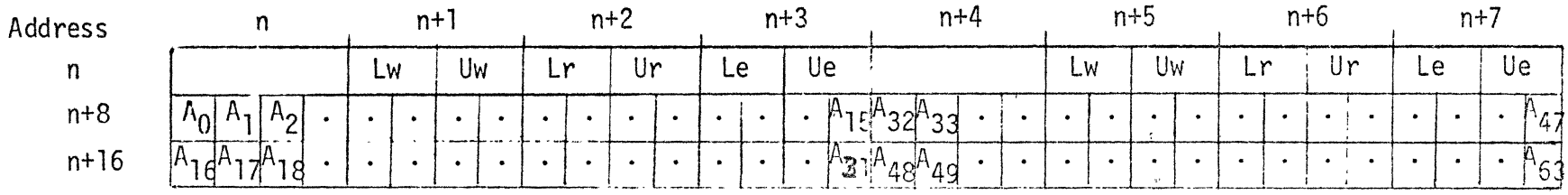


FIGURE 3: IMPLEMENTATION OF MEMORY PROTECTION

The MAP and PROTECT information for a particular processor is constructed in CM under control of the Operating System. The information is transferred to the MCU in response to a command from the PPU via the Common Command Register, or automatically via one of two "Context Switch" commands provided to the MCU from the automatic context switch logic in the central processor.

Command Command Register (CCR) operations cause the register addressed to be loaded with the contents of the location specified by the pointer in absolute CM location 38_{16}, whereas the Context Switch command causes a set of predefined MAP and PROTECT registers to be loaded from the location specified by the pointer at CM location 28_{16}. Figure 4 reflects the CM image of the MAP and PROTECT registers for use with the Context Switch command. A list of MCU CCR commands is presented in the maintenance section. CCR commands are also available to provide access to MCU control and address registers for maintenance and diagnostic purposes.



Lw - Lower Write Protect bound
Uw - Upper Write Protect bound
Lr - Lower Read Protect bound
Ur - Upper Read Protect bound
Le - Lower Execute Protect bound
Ue - Upper Execute Protect bound
An - Value of Actual Page Number corresponding to Virtual Page n of MAP

FIGURE 4: CM IMAGE OF "CONTEXT SWITCH" MAP AND PROTECT REGISTERS

THE EXPANDER/MULTIPLEXER (EX)

The EX adds the memory bus and the processor port expansion capabilities for configuring very large ASC Systems. The unit can be operated in any one of three distinct modes:

1) Up to four "Processors" can be multiplexed onto one MCU processor port. In this sense, a "processor" can be a data channel or a processor bus. Of course, the basic bandwidth limitations must be observed.
2) One MCU memory bus can be expanded to accommodate up to four Memory Units.
3) A single processor's memory bus can be fanned out to allow the processor to access up to four different memory systems.

The EX's can be interfaced with each other (i.e., by "treeing") to provide expansions to 16 or up to 64.

Conflicts at the single port interface are also resolved on either a fixed or a distributed priority basis, in a similar manner to the MCU. These modes are selected by patch card wiring in the expander hardware.

THE MEMORY UNITS

The Memory Control Unit is designed to operate with Memory Units configured as eight-word (octet) storage devices. The TRANSLATOR PC board (patchboard) is used to define the size of the unit on each memory port as well as the interleaving mode.

For the most effective use of ASC resources, the high speed storage devices should have access times in the range of 100-250 nanoseconds. However, the system will operate with slower memories and, due to the interleaving capability, the degradation of performance is not linear with respect to memory speed. Figure 5 illustrates the pipeline nature of memory requests with an assumed module access and cycle time of 140 nanoseconds.

The MCU also has the capability of reporting and testing the Parity logic normally provided with Memory Units.

The active element fast memory modules which are the standard ASC memory units have raw access and cycle times of 140 and 160 nanoseconds.

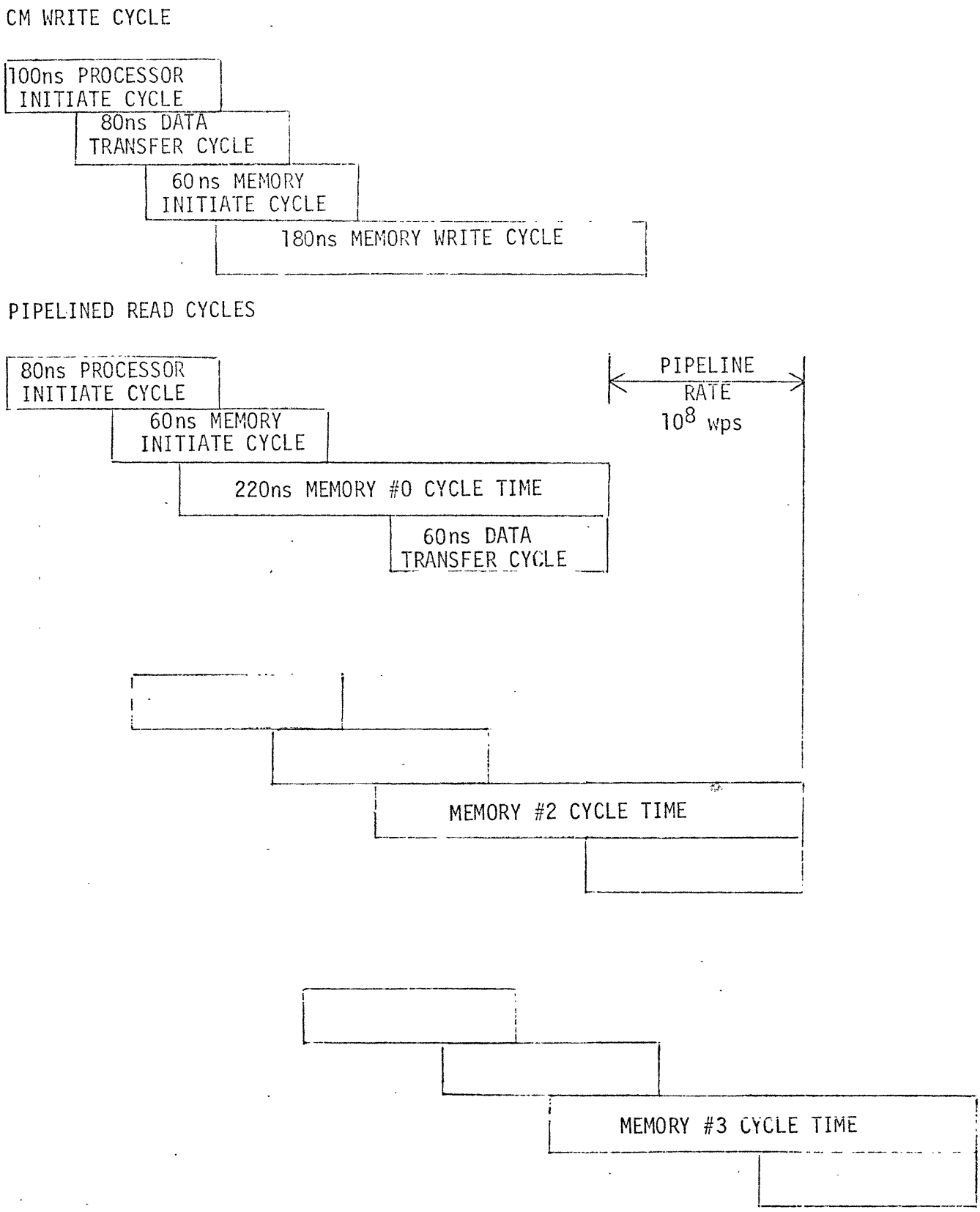


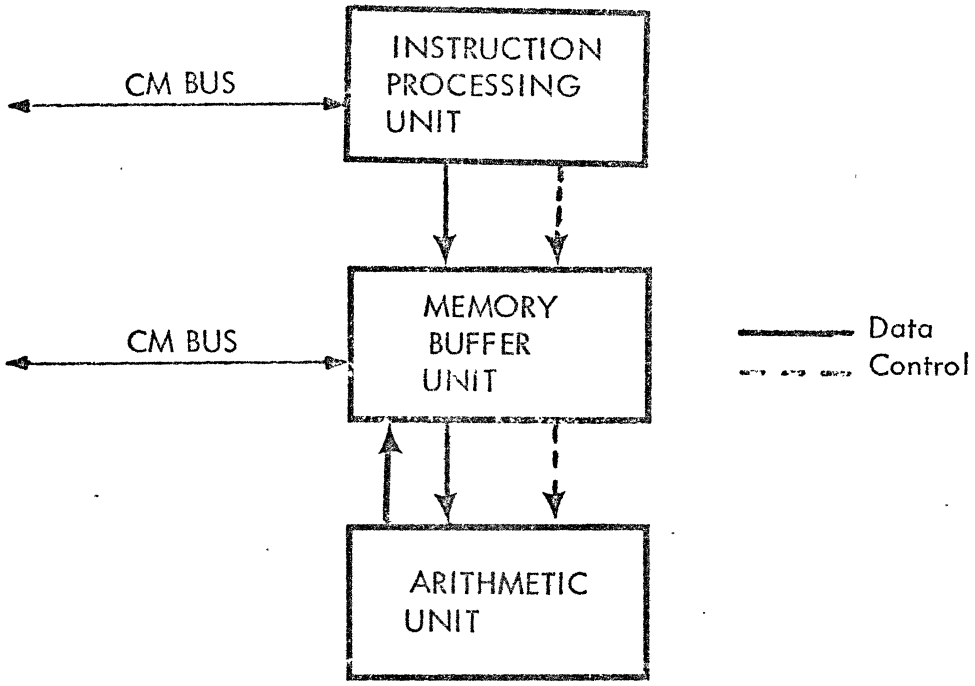
FIGURE 5: MEMORY CYCLE TIMING
140nsec MEMORY MODULE

CENTRAL PROCESSOR DESCRIPTION

TABLE OF CONTENTS

GENERAL

The Central Processor (CP) is comprised of the Instruction Processing Unit (IPU) to process the CP commands, the Memory Buffer Unit (MBU) to provide interfacing with the CM, and the Arithmetic Unit (AU) to perform the specified arithmetic operations. The interaction of these units is shown below:



INSTRUCTION PROCESSING UNIT

The primary function of the IPU is to supply a continuous stream of instructions to the MBU. The IPU block diagram is shown in Figure 1; it performs the following functions:

1. Instruction fetch
2. Instruction decode
3. Register operand selection
4. Effective address development through indexing and/or indirect addressing
5. Immediate operand development
6. Branch address development

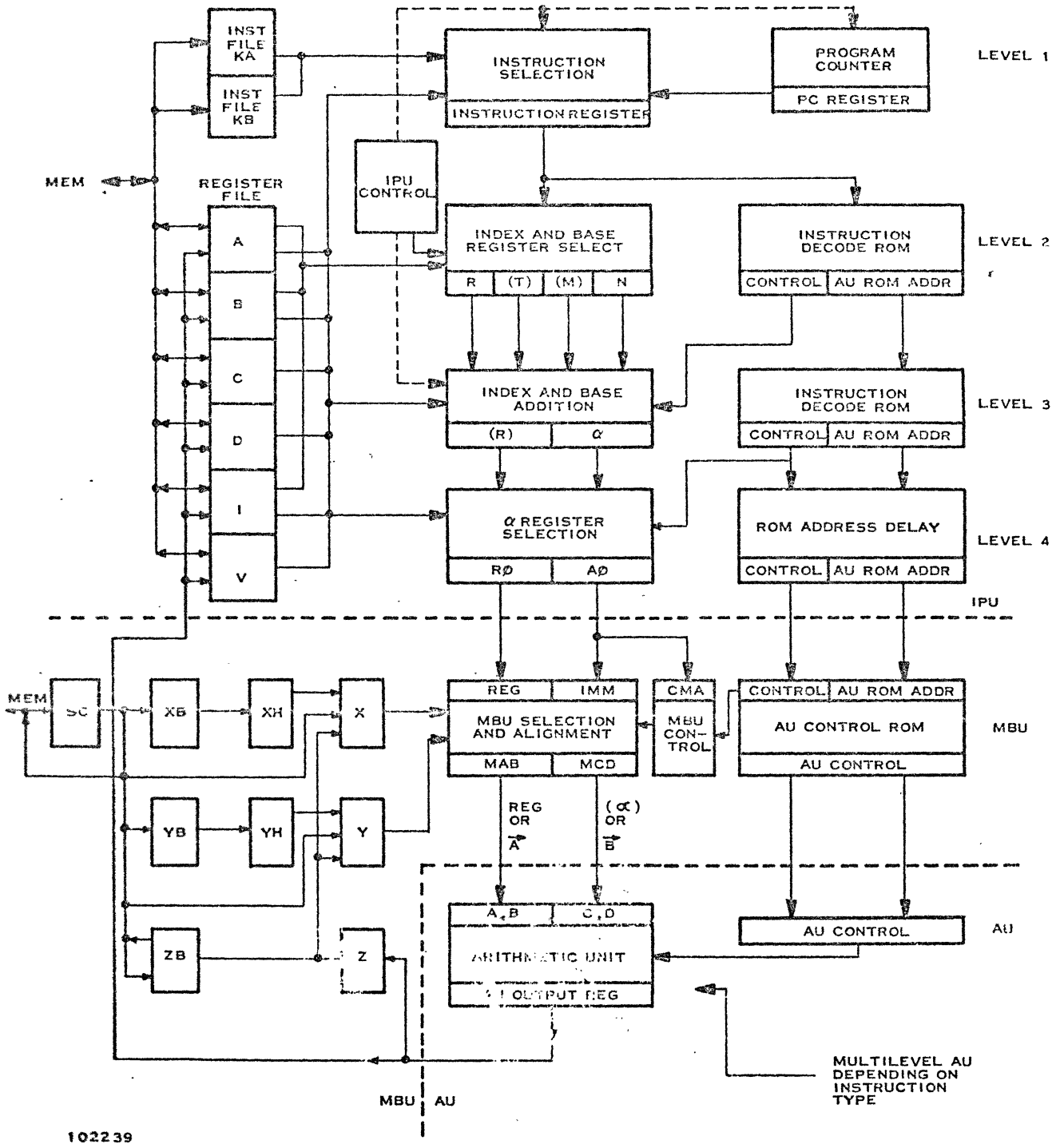


Figure 1. CP Block Diagram

7. Determination of branch condition

8. Storage of AU results into the register file

9. Scalar hazard and register conflict resolution

10. Generation of vector starting addresses

11. Transmittal of vector parameters to the MBU during vector initialization

MEMORY BUFFER UNIT

The primary function of the MBU is to supply the AU with a continuous stream of operands for vector processing and provide for the storing of results of the vector operations. The MBU performs the following functions:

1. Accept the initial vector starting addresses and parameter information from the IPU.

2. Fetch the memory operands requested by scalar instructions.

3. Retention of 16 words in temporary X and Y buffer registers for possible "scratch pad" operations involving data contained in the two most recently referenced memory octets. This temporary storage capability increases by a factor of 1, 2 or 4 depending upon whether a times 1, times 2, or times 4 ASC configuration is installed.

4. Storage of register operands into central memory as a result of scalar store instructions.

5. Temporary retention of 8 words in the Z-buffer register for data destined for one central memory octet address. Data stored by this means is released to central memory when the octet address of write data at the Arithmetic Unit output is different than the octet address of the data contained in the Z-buffer registers. This temporary storage capability increases by a factor of 1, 2 or 4 depending upon whether a times 1, times 2, or times 4 ASC configuration is installed.

6. Update capability from the Z-buffer registers to the X or Y buffer registers for keeping the X and Y registers current when they are being used for "scratch pad" operations.

ARITHMETIC UNIT

The primary function of the Arithmetic Unit is to perform the arithmetic operations specified by the operation code of the instruction currently at the AU level. The Arithmetic Unit is basically a sixty-four bit parallel operating unit which is split into two halves of thirty-two bits each. Double length operations are carried out using both thirty-two bit halves in parallel. Single length operations use the left half AU, while half length

operations use only one half of the capability of the left half AU. The same AU is used for both fixed and floating point instructions. Fixed point numbers are represented as signed integers with negative numbers in 2's complement notation. Floating point numbers are in sign and magnitude with a base 16 exponent represented by an excess 64 binary number.

The Instruction Processing Unit functional areas with four pipeline levels are shown in Figure 2.

INSTRUCTION FETCH

The instruction fetch function of the IPU is concerned with instruction look-ahead into the next octet and instruction look-ahead along the branch path when the loop look-ahead control is active. The principal registers involved in the process of instruction fetching are the present address register (PA), the look-ahead register (LA), and two eight-word instruction register files. The present address register contains the address of the instruction presently being selected from one of two eight-word instruction files. The look-ahead address register ordinarily contains the next instruction octet address ahead of the octet referenced by the present address register. Two eight-word instruction files ordinarily hold the present instruction octet and the next instruction octet.

An initial address is entered into the LA register for transmission to Central Memory via the address bus from the IPU. LA is then transferred to PA as the present address. When the address bus is released by CM, the look-ahead incrementing hardware advances the LA register by eight, equivalent to one octet look-ahead. The LA register sends this second address to CM immediately after the instructions has returned form CM.

The first octet of instructions recieved from CM is synchronized with the Central Processor clock at KCM. The octet is then transferred to file KA on the next clock. The second request for the look-ahead octet is synchronized at KCM when it arrives form CM, and then it is transferred to KB on the following clock.

One clock after the first octet is entered into file KA , the Instruction Register (IR) is loaded with the instruction word selected by the 3 least significant bits of PA. The PA register is incremented by one as each instruction is entered into the instruction register. When the 3 LSB's of PA are 111, the last word in file KA is entered into IR and the instruction look-ahead octet KB is selected. As this transfer occurs the contents of LA is transferred to PA and the LA register is advanced by eight to the address of the next look-ahead octet. This new look-ahead octet is requested from CM while instructions in the KB file are being executed. This process of overlapping instruction requests with instruction execution continues until a branch without look-ahead or an out-of-line indirect address request occurs.

In the case of a branch without look-ahead, the computed branch address replaces the contents of both the PA and LA registers. LA is requested from CM, then advanced by eight for the next look-ahead request and then the process described above is repeated.

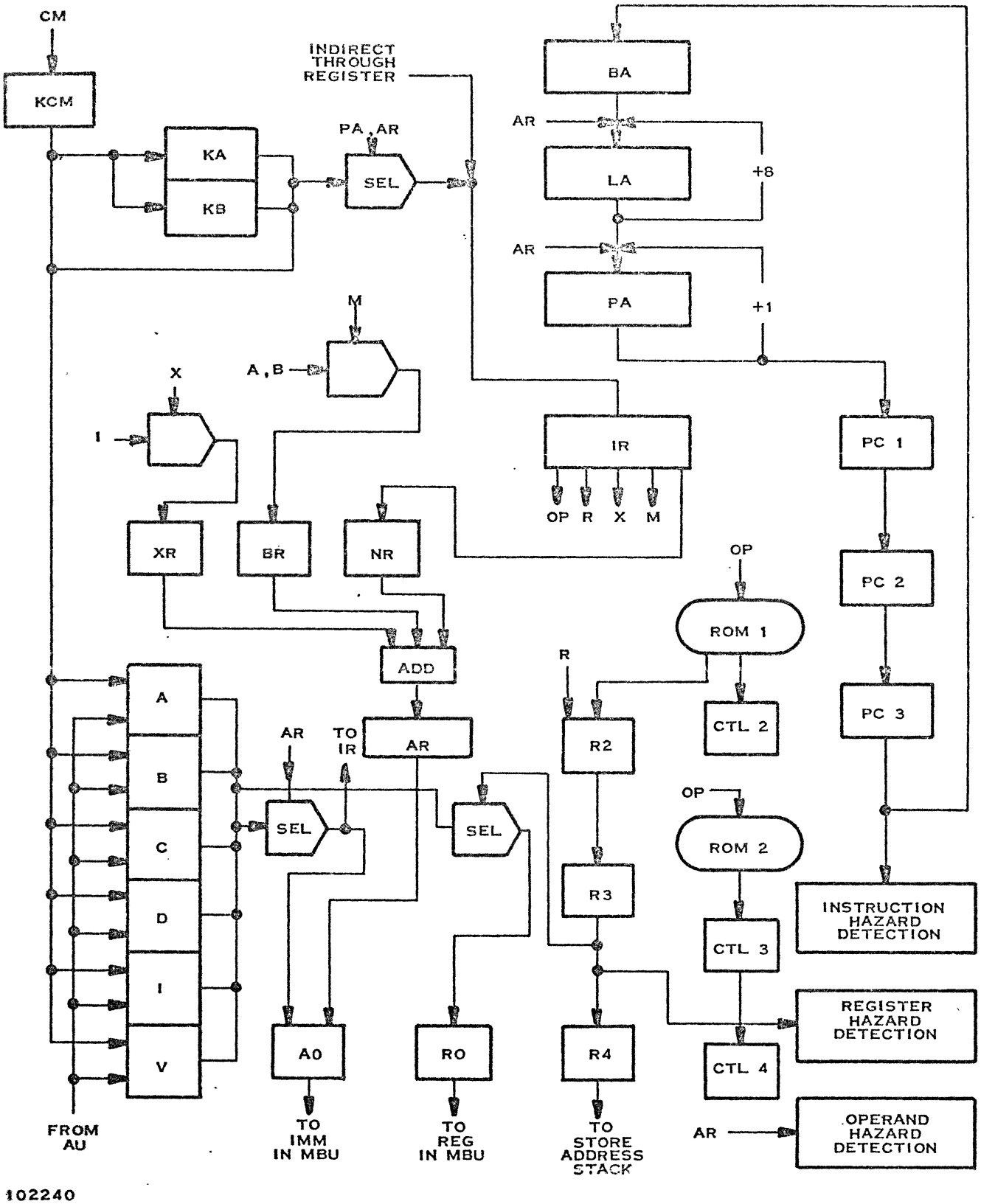


Figure 2. Instruction Processing Unit

A branch with look-ahead is set up by placing the branch instruction at the target location of a Load Look-Ahead (LLA) instruction. This branch does not cause a delay in instruction fetching if both the branch instruction and LLA instruction are properly located with respect to octet boundaries. The LLA should be at the top of an octet and the target branch instruction should be at the bottom of an octet for optimum timing. A penalty of one clock time is paid for execution of the LLA instruction for each pass through the program loop if the LLA is located in this manner.

A Load Look-Ahead instruction enters a count into a look-ahead counter register (LC) in the IPU and enters the program address value of the LLA instruction presently being executed into a branch address register (BA). The count from the N-field of the LLA instruction corresponds to the difference of the instruction locations of the LLA and its target branch instruction. The counter is decremented for each instruction executed following the LLA. When the look-ahead counter is lowered to a value which would indicate that the target branch instruction has been requested by the instruction look-ahead and the look-ahead is now ready to be incremented by eight to the octet beyond the instruction octet which contains the branch instruction, then the look-ahead control will override the normal next octet increment of eight and place the contents of the branch address register (BA) into the look-ahead address register (LA). This causes fetching of the octet which contains the LLA instruction and the loop control is re-initialized when the LLA instruction is executed again after the branch instruction returns the program to the LLA instruction. Loop control by use of an LLA instruction only applies to singular instruction loops up to 256 instructions including the LLA and the BRANCH.

A non-targeted branch instruction located between the LLA and the target branch instruction will inactivate the branch look-ahead control if the non-targeted branch instruction takes the branch path. If a non-targeted branch instruction does not branch, then an active branch look ahead remains active.

INSTRUCTION DECODE

Instruction decode in the IPU is accomplished by the first of two Ready-only Memories (ROM). The ROM size is 256 words by 32-bits.. The first ROM output is at level 2 of the pipeline and is used for IPU control. The second ROM is also used for control but in addition generates a field for use in driving the address inputs to a third ROM contained in the MBU. The third ROM is used to control the Arithmetic Unit. Outputs from both IPU ROM's contain preliminary instruction decoding information needed for the IPU and MBU. Such things as operand word size needed for register operand selection and effective address development are supplied by the ROM's.

REGISTER OPERAND SELECTION

The register operand selection takes place in level 3. Register addresses within a group of registers are specified by the R-field of the instruction word. Register groups are specified by the instruction type as determined by

the operation code. The operation code is used as an address for the first ROM which in turn supplies the two additional bits needed for register group selection. These two bits augment the four R-field bits to form a six bit register address for single length instructions. This six bit register address is applied to the input of a register selection network which can select any one of 48 single word registers in the register file.

The program addressable registers in the Central Processor make up the register file. Each register is 32-bits in length. All registers in the file can be loaded or stored individually or in groups of eight registers at a time with a single instruction. There are six groups of eight registers:

Register Locations	Group Function	Group Designated
00-07	Base address registers	A
08-0F	Base address registers	B
10-17	General registers	C
18-1F	General registers	D
20-27	Index registers	I
28-2F	Vector parameter registers	V

If the instruction word specifies a half length register operand, then the first ROM supplies an additional bit indicating which halfword is to be selected from the 6-bit singleleword address. If a doubleword register is specified, then one bit is dropped from the 6-bit register address and 64-bits are selected into the RØ register at level 4. Doubleword register operands are always selected from an even-odd singleword address pair.

Occasionally the effective address (α) is in the range $\alpha \leq 2F$ and the M-field is equal to zero, in which case the α addressed operand is selected from the register file. Operands of this nature are selected by the IPU after the effective address is developed. The register is selected using the 7 least significant bits of the AR register. These 7-bits include 6-bits of singleword address information plus one-bit for halfword selection if specified by the instruction type.

The output of the α selection network enters up to 64-bits of data into the AØ register if doublewords are specified. AØ is transmitted directly to the MBU input register (IMM) for entry into the MBU output register (MCD) and then to the Arithmetic Unit. The AØ register is at level 4 of the IPU. The register operand for each instruction is carried along and held in the RØ register in parallel with and in the same time relationship as the operand in AØ.

EFFECTIVE ADDRESS DEVELOPMENT

Effective address development through indexing and/or indirect addressing incorporates a major portion of the hardware in the IPU. The T, M, and N-fields of the instruction format specify the index register, base register, and address displacement, respectively. The MSB of the T-field is used to specify indirect addressing. An index register selected from an address by the 3 LSB's of the T-field is entered into the 25-bit XR register at level 2 in the IPU.

A base register selected from an address given by the M-field is entered into the 24-bit BR register at the same level. The N-field displacement is copied into the NR register. XR, BR, and NR form the three inputs to the index adder in level 3.

The XR register is shifted one bit position to the right, left, or not at all depending upon whether halfword, doubleword, or singleword addressing, respectively, is specified by the instruction code. The shifting takes place prior to addition in level 3.

The output of the index adder is entered into a 25-bit register, AR, which holds the effective address of the instruction presently at level 3. The LSB of AR is the halfword address selection bit. The LSB is forced to zero for singleword addresses and the two LSB's are forced to zero for doubleword addresses.

The 21 most significant bits of the 25-bit effective address register (AR) is sent to the Central Memory Address Requestor in the Memory Buffer Unit (MBU) if the addressed data is not presently residing in either the X or Y data registers of the MBU. If the addressed data is present in the X or Y registers, then the 4 LSB's of AR are sent to the MBU to perform the selection of the appropriate doubleword, singleword, or halfword from X or Y using 2, 3, or 4 of the four bits, respectively.

Indirect address requests cause transfer of the effective address register to the look-ahead address register (LA). The indirect address is requested from central memory by LA . The octet containing the indirect address is read from memory and entered into instruction file KCM. The indirect address is selected from KCM by AR. At this point, the instruction register contains the indirect address which is interpreted according to the indirect address format.

Bit positions 5 through 7 of the indirect format specify the index register to be selected into pre-index register, XR. Bit positions 8 through 31 of this format specify the indirect address, designated ADR.

The four most significant bits of the indirect address format (bit positions 0 through 3) must be zero to indicate a "no-operation" for the Arithmetic Unit. The 24-bits of ADR and XR are added in the index addition section of the IPU. The result appears in register AR.

If the indirect bit (bit position 4) of the indirect format is a one, then the contents of AR is a singleword central memory address which points to the next level of indirection. The next level indirect address is requested from central memory via the LA register path. The process described in the preceding paragraph is repeated for each level of indirect addressing.

If the indirect bit of the indirect format is zero, then the terminal indirect address has been reached and the index addition hardware of the IPU develops the address of the operand using displacement indexing according to the word size of the instruction being executed. The terminal indirect address is sent to the central memory address requestor in the MBU if the addressed data is not presently residing in either the X or Y data registers of the MBU. The 4 LSB's of the terminal indirect address are sent to the MBU word selection logic if the addressed data is present in either the X or Y data registers.

Subsequent scalar instructions from one of the instruction files follow the terminal indirect address into the IPU and normal instruction processing continues until another indirect, execute, or branch instruction is detected.

An exception to the above description on indirect addressing occurs for the case of a first level indirect address with $\alpha \leq 2F$ and an M - field of zero. The first level indirect address is the address of the instruction word (the word with four non-zero most significant op code bits and indirect bit equal to one).

In this case, the value of the next level indirect address is selected from the register file and placed in the instruction register. After this, all subsequent levels of indirection are through central memory. Indirect references through the register file can occur only once for a given instruction.

IMMEDIATE OPERAND DEVELOPMENT

Immediate operand instructions use the index adder for modifying immediate values. The M and N-fields of the instruction word combine to form a 16-bit value which is added to a nonshifted index register selected from the T-field. Sign extension into the left halfword occurs prior to addition using the MSB of the M register for singleword arithmetic immediate operand instructions. Zeros replace sign extension for singleword logical immediate operand instructions. Halfword immediate instructions use only the right halfword of the result from the index adder.

BRANCH ADDRESS DEVELOPMENT

Branch address development takes place in the index adder of level 3 using inputs from the index selection register, XR, and base selection register, BR, of level 2. When the M-field of the branch intruction is zero, the program counter value replaces the base register value in BR. The branch address is taken relative to the program counter plus index (if specified by T).

Indirect branch addresses are developed similar to indirect operand addresses with the exception that indirect branch instructions with $\beta \leq 2F$ and M = 0 reference central memory and not the register file.

DETERMINATION OF BRANCH CONDITION

Instructions of the type "Branch on Register Greater than" use a special adder unit in the IPU to determine the outcome of the branch test without having to wait for the Arithmetic Unit to perform the operation of adding to a register and testing the result with respect to the contents of another register. This special adder is incorporated in the IPU hardware and receives its inputs from the AØ and RØ registers at level 4.

For the branch instruction under discussion, the operation involves taking the singleword contents of the arithmetic register specified by the R-field and adding to it the contents of the arithmetic register specified by T. The result is compared with the contents of the arithmetic register specified by T plus one. In the IPU, this operation is accomplished in the branch test level by taking the register operand specified by R from the left half of 64-bit register RØ. The left half of 64-bit register AØ supplies the register addressed by T. The right half of register AØ supplies the register addressed by T plus one. These three singleword register values are added in the 3-input "branch test adder". The input from register T plus one is complemented (one's complement) before addition so that the addition which takes place is the evaluation of $(G)_R + (G)_T - (G)_{T+1}$ using one's complement addition. The result of this addition will appear to be one less than the 2's complement addition of the same number. If one desires the sum to be greater than zero in 2's complement addition, then the sum must be greater than minus one in one's complement addition. Therefore, the outcome of the branch test for this instruction is true if the output of the one's complement addition is zero or positive. This can easily be determined from the sign position alone.

Many other branch and test instructions fall into the class that can take advantage of the branch test level. More specifically, all of the increment or decrement test and branch instructions can use this means for determining the outcome of the branch test without waiting for the increment or decrement operation to take place in the arithmetic unit.

Conditional Branch instructions which compare the R-field value with the Condition Code or Result Code are of a different nature than the branch instructions just mentioned. The outcome of Conditional Branch instructions are known only if all previous instructions which set the Condition Code or Result Code have passed through the Arithmetic Unit. If the proper code has not been set, then the Branch on Comparison or Branch on Results instruction must wait in level 3 of the IPU until the code has been set by the AU. The branch address is held in the AR register of the index adder until the branch decision is made. Then, if the branch is taken, the branch address is transferred to the PA and LA registers and then central memory instructions along the branch path are requested.

STORAGE OF AU RESULTS INTO THE REGISTER FILE

The IPU has the function of retaining the destination register address of all scalar instructions (vector instructions cannot store into the register file). These register addresses are held in a chain of 7-bit registers. Seven bits address all 48 singleword registers down to the halfword level. The chain of register addresses is as long as there are sections to the Central Processor pipeline. Additional bits are carried along for control.

The IPU provides the proper alignment from the AU to the register file for single, half, and double length register operand results. It also performs the selection enabling to the gate inputs of the register file.

SCALAR HAZARD AND REGISTER CONFLICT RESOLUTION

These functions are described in Section B2. Scalar hazards occur when the effective operand address developed by an instruction is the same as the address of a store instruction which preceeds the read instruction and which has not yet passed completely through the Central Processor pipeline structure. The hazard condition will clear when the store instruction performs its write operation into central memory.

Register Conflicts exist when an instruction requires a register operand which is presently in the process of modification or is going to be modified by the Arithmetic Unit and which has not yet emerged from the AU output. The register conflict will resolve itself when the needed register is loaded with the result from the Arithmetic Unit and no other modification will occur to the contents of that register as a result of other instructions between the AU output and the instruction which requires that register.

GENERATION OF VECTOR STARTING ADDRESSES

The index and base selection level and the index addition level develop the effective starting addresses for vector instructions from the vector starting addresses plus index values as specified by the vector parameter file. The generation of continuous addresses for sustaining vector operations is carried on by the Memory Buffer Unit. Section B2 and B3, vector timing and the vector parameter file descriptions in the instructions describe the operations to be performed for generating vector starting addresses.

TRANSMITTAL OF VECTOR PARAMETERS TO THE MBU DURING VECTOR INITIALIZATION

The vector starting addresses contained in 29, 2A, & 2B and other vector parameters contained in register file addresses 28, 2C, 2D, 2E, 2F, and the 4 MSB's of 2A and 2B must be transmitted to the Memory Buffer Unit (MBU) prior to starting the first arguments of the vector operation through the Arithmetic Unit. Vector starting addresses are sent to central memory immediately after being received at the MBU and while the remaining vector parameters are still being transmitted from the IPU.

The parameters are selected via the A register operand selection network and gated into the AØ register of the IPU one word at a time. The output of AØ goes to the IMM register at the MBU input and from there the parameters are distributed to the operational registers (working registers) of the MBU. These operational registers control the vector address generating hardware in the MBU which sustains the vector operation.

When vector initialization is completed the IPU brings the next three scalar instructions, if such exist, down through levels 1, 2, and 3 of the IPU hardware. These instructions reside in the top of the IPU pipeline until the last element result of the vector operation has been sent to central memory.

Some vectors, namely Vector Order, Dot Product, Search, Compare, and Peak Pick, require special consideration. These vectors must be restarted at their beginning addresses when reinstated following a context switch operation if the switching occurred during their vector processing interval. For these vectors the IPU retains the vector instruction in level 3 and reserves level 2 for recomputing that vectors starting addresses. The following instruction after the vector resides in IPU level 1.

The memory buffer unit (MBU) provides an interface between central memory and the arithmetic unit (AU). The communication with central memory is via a private port of the memory control unit (MCU). During scalar operations, data specified by effective addresses developed in the instruction processing unit (IPU) are fetched or stored as required. For most vector operations, two operand data strings are fetched while a result data string is stored. Addresses for sustaining the vector operation are computed in the MBU using parameters initially specified by the vector parameter file in the IPU. Details for the one times ASC are described below. An overall block diagram of the Central Processor is depicted in Figure 1.

DATA PATHS IN MBU

Octets from central memory are received and synchronized in the register designated SC. A direct path is provided for transfer to either the X or Y registers from SC. The XH and XY registers provide a second level of buffering so that vector processing can be sustained at a high rate with a minimum of memory limiting. The SC register is always transmitted to its destination on the next clock after it is received from CM. The XB and YB registers provide a third level of buffering and are used to equalize the processing rate between the two operand paths. Both X and Y can also be stored in central memory for maintenance purposes.

Results from the AU which are to be stored in central memory are aligned and placed in the Z register. The Z register can be transferred to either X or Y so that memory references are not necessary for scalar memory opeands which reside in Z.

If the result of the output of the AU is in a different octet than the octet currently represented in Z, then Z must be transferred to the ZB register which in turn must be transferred to central memory. The transfer of Z to ZB must be held up until the previous write request no longer requires ZB. If ZB contains half words, it is possible to have incompletely specified single words. A path from SC to ZB is provided to permit half word fill-in from central memory.

The register pairs designated MAB and MCD present two operands to the AU receiver registers. Each pair can contain half words, single words or doublewords. Their positioning is shown in Figure 3.

each is shown below:

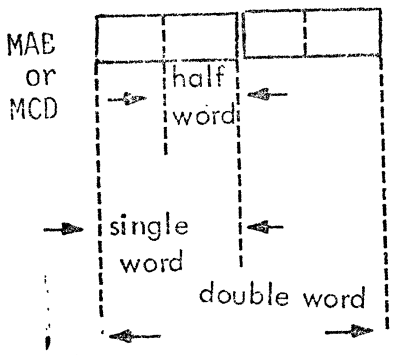


Figure 3. MBU Output Registers

Selection networks are provided for both X and Y and are capable of selecting half words, single words, or double words. The outputs are therefore 64 bits wide. Single words are placed in the most significant bit positions. Half words are aligned and signs extended so that they appear as single words. The X register file selection can be transferred to either MAB or MCD. The transfer to MCD is for scalar requests of the X register file. The Y register file is transferred only to MCD.

Register and immediate operands from the IPU are received in the REG and IMM registers respectively. They can then be transferred to the MAB and MCD registers as required.

The MBU receives vector initialization data from the vector parameter file in the IPU via the IMM register. A path from IMM to MAB is required for vector A immediates.

SCALAR OPERATION

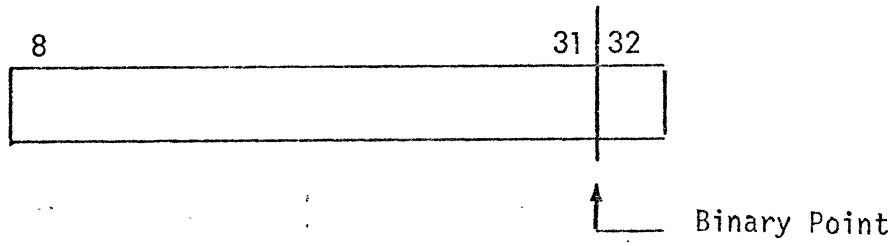
During scalar operation the MAB register presents the IPU's register data to the AU while the MCD register presents the "memory" data. The "memory" data can be selected either from X or from Y or can be an immediate operand from the IPU.

The IPU sends two types of addresses, octet and element. The octet addresses are required when an octet is to be read or written and is not currently represented in the MBU. The four bit element addresses specify which elements are to be read or written. Operand addresses are accompanied by destination (X or Y) tag. Both operand and result addresses are accompanied by word size information. Scalar operations utilize part of the structure required for vector operations. Details are discussed with vector operations.

VECTOR PROCESSING

During vector operation, the X and Y registers present elements of vectors $\vec{A}$ and $\vec{B}$ to the AU. The addresses for these operand data strings are computed in the $\vec{A}$ ADDRESS GENERATOR and the $\vec{B}$ ADDRESS GENERATOR. A detailed diagram is shown in Figure 4 for the $\vec{A}$ ADDRESS GENERATOR.

Successive addresses are the outputs of a 25 bit adder with the format shown below. This output is the sum of the last address and one of the increments corresponding to the self, inner, or outer loops. The increments associated with a self loop are one element increments and will be ±1/2, ±1 or ±2 depending on word size. The inner loop increment is designated DAI. The outer loop increment is designated DAØ. Both DAI and DAØ are initially 16 bit signed 2's complement numbers in the vector parameter file. They are adjusted according to word size, sign extended, and then placed in 25 bit registers in the MBU.



Address Format

The initial address (IA) is transferred to the NAA register from the IMM register. The generation process is capable of providing addresses for continuous processing of single word vector elements since a new address can be computed at the rate of one address per clock.

In order to minimize and frequently eliminate memory delay, a "three level" look ahead scheme is provided. The octet address being processed references data which is to replace the contents of the XB register. The address of an element must in some cases be formed 16 clocks prior to its use in the selection network. The XBA register receives a 21 bit octet address while a bit is set which initiates a memory request when appropriate. Octet comparisons are made between NAA and XBA. Memory requests are made only when a new octet is required..

The four bit element selection is stored in a file designated the circular address file (CAF). Since an additional 15 elements could be generated during the interval between the generation and use of a particular address, the CAF has space for 16 four bit entries. An additional three bit "TAG" must be stored for each entry. The first of these three bits indicates the presence of an address. The second bit indicates the end of a self loop. The third bit indicates the first address of a new octet.

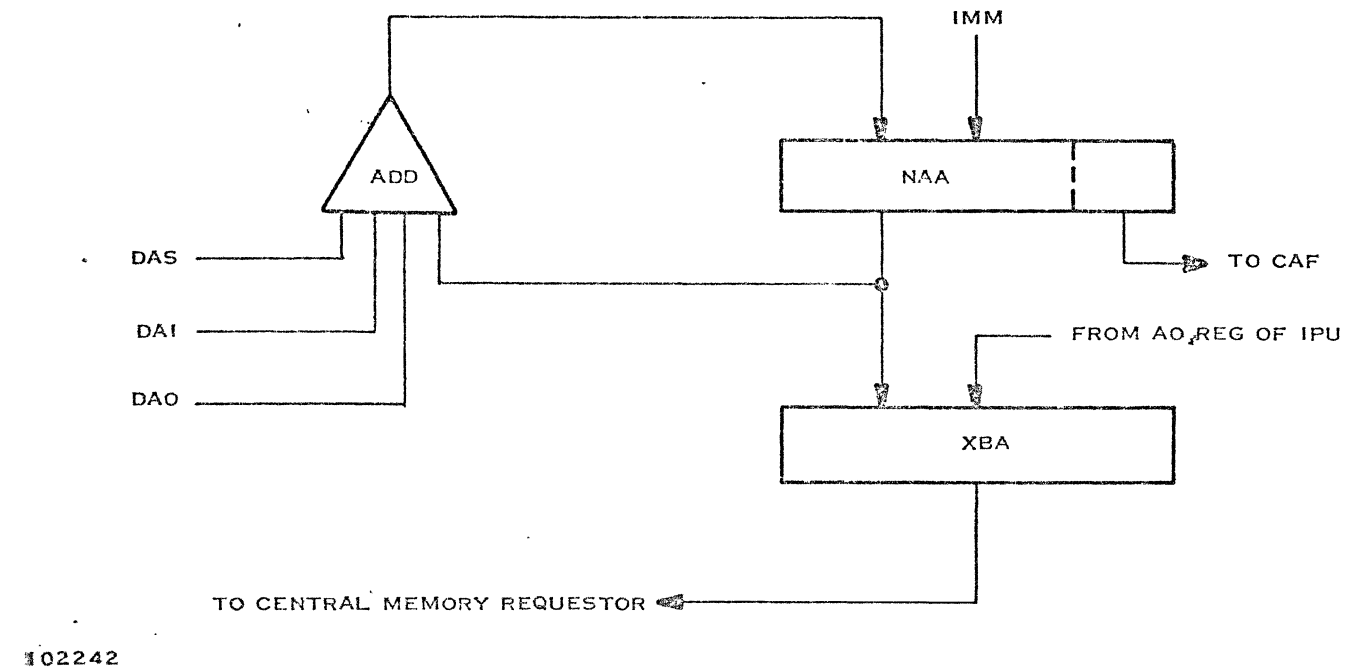


Figure 4. $\overrightarrow{A}$ Address Generator

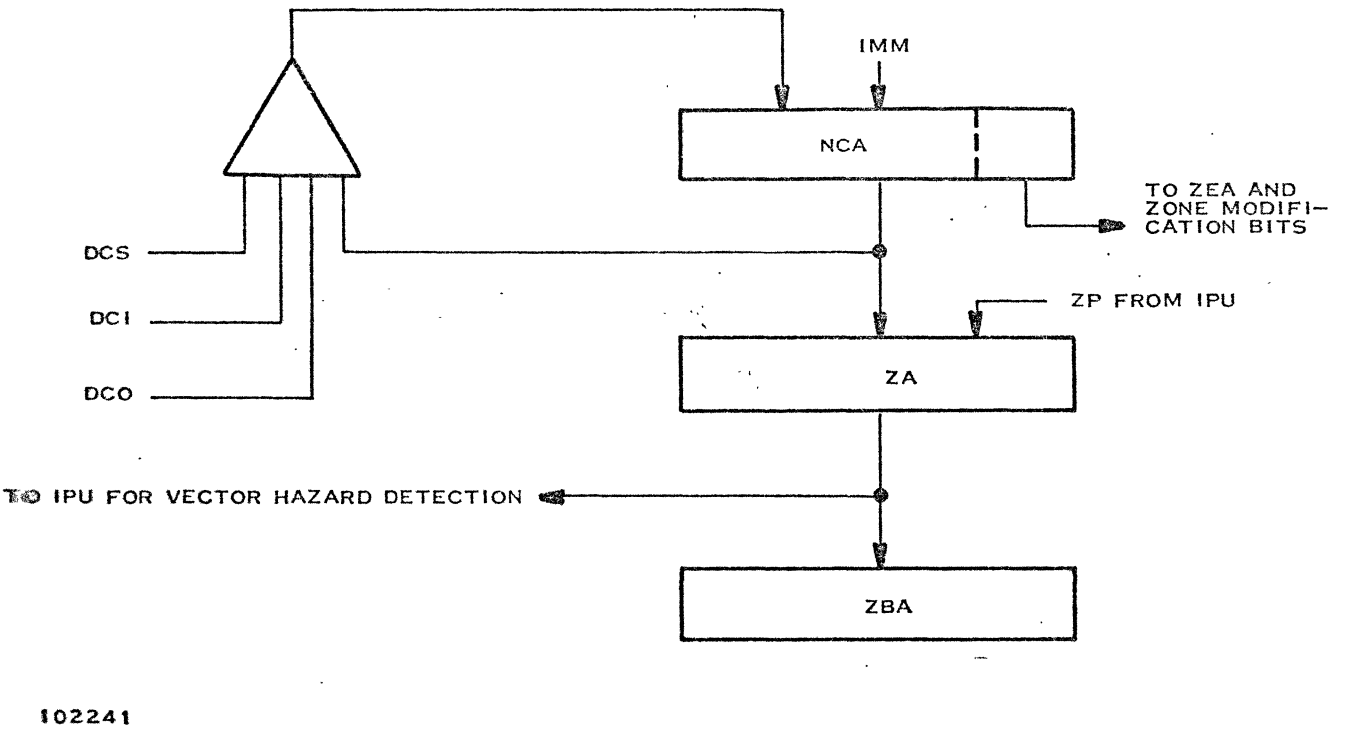


Figure 5. $\overrightarrow{C}$ Address Generator

The operand address register XBA is also used for scalar operation. Appropriate portions of the 25 bit addresses from the AR register in the IPU are transferred to the XBA register and the XA register. The IPU indicates whether or not the octet is currently in the X register. The XBA register is sent to the ØA register only if a new octet is required. The XA register is used to enable the X selection network.

During vector operation, the Z register accepts elements of vector $\vec{C}$ from the AU. The addresses for the output data string are computed in a section designated the $\vec{C}$ ADDRESS GENERATOR. Addresses are generated one clock prior to their use (at the alignment network). Successive addresses are the outputs of a 25 bit adder in an arrangement similar to that for operand address generation. The initial (IC) is transferred to the NCA register where successive addresses are also placed. New octets are detected between NCA and ZA. If ZBA is available ZA can be transferred to ZBA and the write request can be processed. Sixteen bits are required to record modification (down to half words) for both Z and ZB. Half word fill-in can then be accomplished when required. The ZB modification bits are combined in pairs to specify zone control bits for all ZB write operations. The ZA register is presented to the IPU during vector operation so that hazards can be detected in the IPU.

During scalar operation, addresses from ZP in the IPU are sent to ZA and ZEA. Since the IPU indicates octet changes, the MBU continually transmits the availability status of ZBA for write operations. When ZBA is not "busy", ZA can be transferred to ZBA and the write operation processed.

Operation of the MBU is under control of the section designated the SEQUENCE CONTROL. Inputs to this section are status of the memory requestor, status of each address generator, and status of the loop counters. Three sets of loop counters are provided, two from operands and one for results. The inputs combined with the present state of the SEQUENCE CONTROL determine gate enables as well as the next state.

The AU control ROM is located in the MBU. Instruction codes are received from the IPU and used as addresses for the ROM which generates an array of signals which control the AU.

CENTRAL MEMORY REQUESTS

All central memory requests are made through a controller designated the CENTRAL MEMORY REQUESTER. The CENTRAL MEMORY REQUESTER establishes priority for the three address generators and makes the appropriate requests to the MCU. It also provides for the distribution of read data upon arrival in the MBU. A maximum of four requests may be in some state of development during vector operation. Requests can also be processed for the hard core controller during maintenance operations.

MEMORY BUFFER UNIT SUMMARY

Effective addresses developed by the index unit in the IPU are routed to the memory buffer unit. For most scalar instructions the memory buffer unit obtains one operand from the central memory location specified by the effective address and one operand from a register as presented from register selection. The memory buffer presents these operands to the arithmetic unit for processing. The arithmetic unit results replaces the register operands to the register file of the IPU. When results are to be stored into central memory, the memory buffer unit receives the effective address into which the data is to be stored and after AU processing provides for the storing operation.

For vector operations, the memory buffer unit supplies the consecutive operands to be processed and stores the results in central memory.

The memory buffer unit is comprised of two triple buffered eight-word register groups for reading and one double buffered eight-word register group for writing in the one times ASC system. Triple buffering is provided so that vector processing can be sustained at a high rate with a minimum of memory limiting.

For scalar operations, buffers X and Y are alternated for memory read operations. Buffer Z is used for memory write operations. In either case, the strategy is to invoke a memory cycle only when one is needed. For example, a read request for data within an octet currently residing in a buffer is terminated at the buffer. A write operation into a previously defined write octet is likewise terminated at the buffer. An actual read cycle occurs only when the required data is not within a current octet. An actual write operation occurs only when a new write octet is defined.

For vector operations, buffers X and Y supply strings of numbers to be processed and buffer Z accepts the resultant string of numbers.

GENERAL

The ASC Arithmetic Unit is basically a sixty-four bit oriented unit. The unit is used for both fixed and floating point instructions. Floating point numbers are in sign and magnitude along with an exponent represented by an excess 64 number.

A distinguishing feature of the ASC AU is the pipeline structure which allows efficient processing of vector instructions. There are seven exclusive partitions of pipeline involved, each of which is designed to provide an output every sixty nanoseconds. The seven sections are referred to as (1) Exponent Subtract, (2) Align, (3) Add, (4) Normalize, (5) Multiply, (6) Accumulate, and (7) Output.

The first four sections mentioned above are the basic structure of a floating point add instruction. Each of the sections perform parts of other instructions; however, they are primarily partitioned in this way to increase the floating point add time. Each of these sections is capable of operating on double length operands so that vector double length instructions can proceed at the clock rate. The align section is used to perform right shifts in addition to the floating point alignment for add. The normalize section is used for all normalization requirements and will also perform left shifts for fixed point operands. The add section employs second level look-ahead techniques to perform both fixed and floating point additions. This section is also used to add the pseudo sum and carry which is an output of the multiply section.

The multiply unit is able to perform a 32 by 32 bit multiplication in one clock time. The multiplier is also the basic operator for the divide instruction and double length operations for both of these instructions require several iterations through the multiply unit to obtain the result. Fixed point multiplications and single length floating point multiplications are available after only one pass through the multiplier. The output of the multiply unit is two words of 64 bits each, i.e., the pseudo-sum and pseudo-carry which must be added to the add section to obtain the proper solution. A double length multiplication will be performed by pipelining the three following sections: multiply, add, and accumulate. The accumulate section is similar to the add unit and is used for special cases such as VDP or any instruction which needs to form a running total. Double length multiplication is such a case, as three separate 32 x 32 bit multiplications will be performed and then added together in the accumulator in the proper bit positions. A double length multiplication would therefore require six clock times to yield an output while single length would require only four. A double length multiplication implies that two sixty-four bit floating point numbers (56 bits of fraction) are multiplied to yield a sixty-four bit result with the low order bits truncated after postnormalization. This multiplication ignores a possible round-off which is obtained by making a fourth pass with the two least significant halves of the operands. A fixed point multiplication will perform a 32 x 32 bit multiplication and yield a sixty-four bit result.

As would be expected, division is the most complex operation to be performed by the AU in the ASC. The method used takes advantage of the fast multiplication capabilities and employs an iteration technique which upon a specified number of multiplications will form the quotient to the desired accuracy. This method does not form a remainder. However, a remainder can be obtained under program control. Assuming X/Y = Q was the solution, the remainder can be formed by multiplying Y * Q and subracting from X; R = X - Y * Q. The remainder will be accuracte to as many bits as the dividend X. For floating point operations, each of the operands, along with the result, are equal in length. For fixed point single length division, the divisor and result are 32 bits while the dividend is 64 bits in length.

The output section is used to gather outputs from all other sections and also to do simple transfers, Booleans, etc. which will require only one clock time for execution in the AU.

FLOATING POINT OPERANDS

A guard digit consisiting of four least significant bits is provided to avoid loss of one hexadecimal digit of accuracy which would result from truncation prior double length additional and subtraction. The addition of these bits is sufficient since the only times normalization will be required with a possibility of loss of accuracy, the normalization will require a shift of only one hexadecimal digit. Normalized operands are required for the guard digit to be of maximum use. For example, in multiplication, given two operands which are normalized, the fractions will be $2^{-4} \leq f < 1$. The result will be $2^{-8} \leq f < 1$. Thus, the result will always require at the most one four-bit shift to normalize. The addition case is more involved but can be explained by discussing three possibilities. If the exponents are equal, no alignment is required therefore the guard digit is not necessary. If the exponents differ by one, the guard digit will retain significant information. Finally, if the exponents differ by more than one, it can be shown that the result to be normalized will require at most a shift of one hexadecimal digit. Thus, the guard digit contains information that can be retrieved.

The results of floating point operations treat overflow and underflow as suggested by the Share XXVII conference. Any overflow or underflow results in the correct mantissa and the exponent correct modulo 128. An output from the AU indicates overflow or underflow. The output can then be employed by taking proper action.

STRUCTURE

Exponent Subtract - The exponent subtract section is primarily responsible for determining the proper inputs to the add section for use in floating and instructions. It is used for both scalar and vector operations and is also responsible for supplying proper input to the accumulator section for floating vector-dot product instructions.

This section determines the difference in the exponents of floating point operands or in the case of equal exponents, which mantissa is larger. Upon determining the larger number, the true or complement of the operands are gated into registers according to the operation to be performed such as Add, Subtract, Add Magnitude, etc. Also, at this time a seven bit subtracter determines the exponent difference which is used in the align section to properly align the floating point operands.

Since logic is required in this section to determine relative magnitudes of the mantissa, the test instructions for greater than, less than, or equal to are also performed in this section to avoid repetition of hardware.

Align - The align section is in operation for all floating add instructions or for any right shift instruction. Floating point instructions are performed after one pass through the align section while fixed point shifts require two cycles.

The shift logic has provisions to allow any shift length which is a multiple of four to be performed in one cycle. Since floating point numbers are represented in hexadecimal digits, this will facilitate the very fast floating point additions. The length of right shift can be obtained from the instruction word for a shift instruction or from the exponent difference information as supplied by the exponent subtract section.

Fixed point right shifts are performed by first shifting in one cycle the largest multiple of four-bits contained within the shift value. Then, the residue of 0, 1, 2, or 3 bits is shifted on the next cycle. This results in a minimum of shift paths into each latch since the four bit paths already exist.

Add - The add section is shared for many instructions depending only upon which paths are selected into the section. Floating add instructions are entered by way of the align section. Fixed point add operands enter the arithmetic unit at the add section. The adder is also used to add the pseudo sum and carry from the multiply section to obtain any multiplication result.

The adder is 64 bits in length and contains second level look-ahead logic. The floating point numbers are in the proper format when entering the add section, however, the fixed point operands are modified to reflect either add, subtract, or add magnitude type instructions.

Normalize - The normalize section is employed for both floating add instructions and fixed point left shift instructions. Divisors are routed through this section to guarantee bit normalized inputs for divide instructions.

This section closely resembles the align section in that floating point operations require only one cycle while fixed point shifts require two. The major difference in the two sections is that the align section is given the information concerning length of shift for hexadecimal digits in floating point. The normalize section has to compute the length of shift required to normalize a floating point number by examining to determine which four bit group contains the most significant logical one. An adder is also required to update the exponent when a normalization takes place.

Fixed point left shift instructions are supplied with the length of left shift from the instruction word.

Multiply - The multiply section is required to operate on both floating and fixed point operands. The floating point numbers are represented in sign and magnitude where the fixed point numbers are in two's complement from. The method of multiplying is keyed to two's complement operands with the floating point multiplication performed by arbitrarily assigning positive signs during the multiplication and then applying the proper sign when multiplication is complete.

The multiplier is capable of multiplying any two numbers up to 32 bits in length in only one pass through the multiply section. The result is in the form of a pseudo sum and carry which must be added together to obtain the result. The sum and carry are added together in the add section which has been discussed separately.

The multiply section is also used to perform division by a sequence of multiplication operations.

Accumulator - The accumulator is a special purpose section which is employed when an vector operation is being performed requiring an accumulated total. A prime example of this is the Vector-Dot-Product instruction.

Like the add section which was previously described, the accumulator performs a second-level look-ahead to facilitate a fast addition.

Output - All results to be sent to the CP must be gated through the output section. Information could have originated in any one of the other sections of the AU with the exception of the multiply section.

Simple instructions such as Booleans, transfers, masks, etc. are performed in this section and gated out in one pass through the section.

CENTRAL PROCESSOR TIMING ANALYSIS

TABLE OF CONTENTS

Scalar instruction processing time in the Central Processor (CP) can be predicted with a fair degree of accuracy by considering the following factors:

1. Arithmetic Unit clock time
2. Operand fetching time
3. Register conflict delay
4. Multiple store instruction delay
5. Read after write delay
6. Indirect address generation time
7. Execute instruction fetching time
8. Instruction fetching time after a branch instruction without look ahead
9. Instruction hazard refetch time

ARITHMETIC UNIT CLOCK TIMES

Arithmetic Unit clock times (Table 1) are defined as the number of clocks required to propagate input arguments through the AU, to the AU output registers. Certain instructions can be executed in sequence by the Arithmetic Unit without creating periods of inactivity or delays in pipeline flow. The instructions listed in Table 2 are divided into fourteen groups. Any single instruction in groups 2 through 11 may follow any instruction in groups 1a and 1b without creating a delay due to different AU types.

OPERAND FETCHING TIME

Operand fetching time refers to Central Memory access time for obtaining memory operands. This time is measured from the clock at which the effective address of the operand is at the address output register (AØ) of the Instruction Processing Unit (IPU) to the clock at which the requested operand resides in the output register of the Memory Buffer Unit (MBU). This time is normally ten clocks if there are no memory conflicts at the MCU or priority delays at the MBU memory controller as illustrated below:

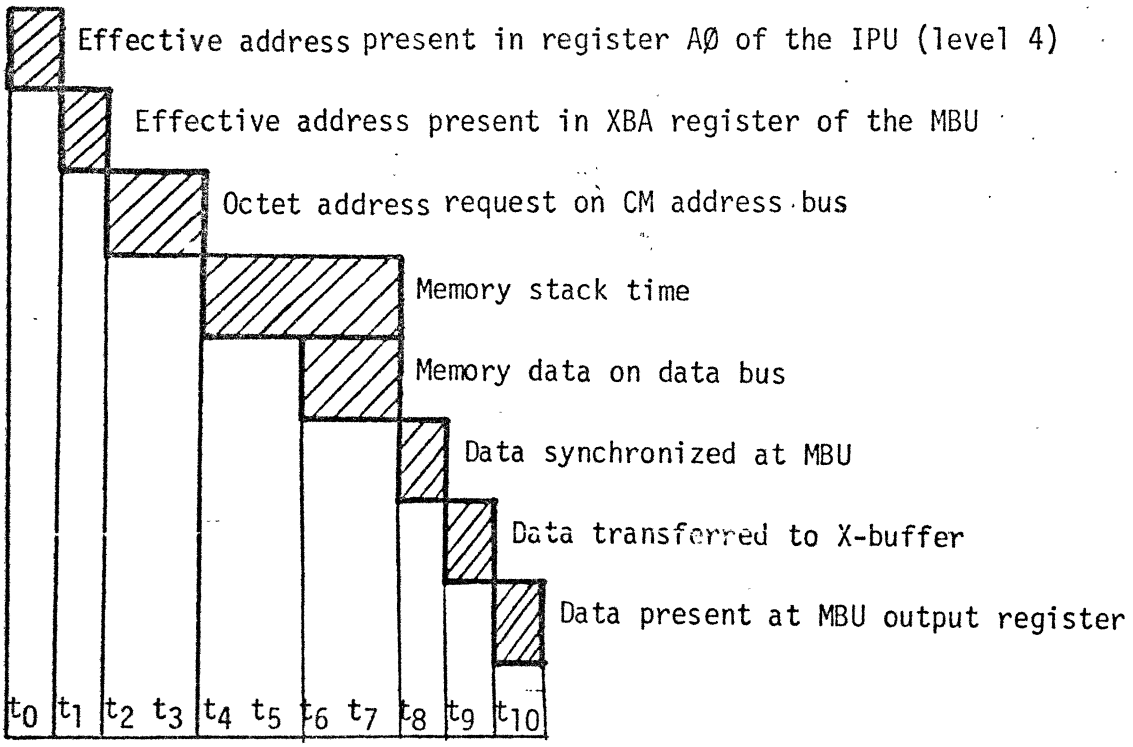


TABLE I. ARITHMETIC UNIT TIME REQUIREMENTS FOR SCALAR INSTRUCTIONS

LØAD		Clock Times
	L	1
	LI	1
	LH	1
	LIH	1
	LR	1
	LL	1
	LD	1
	LM	2
	LMH	2
	LMF	1
	LMD	1
	LN	2
	LNH	2
	LNF	1
	LND	1
	LNM	2
	LNMH	2
	LNMF	1
	LNMD	1
	LF	1
	LFM	*
	XCH	2
	LAM	1
	LAE	1
	LLA	1
	LO	1
	ST	1
	STH	1
	STR	1
	STL	1
	STD	1
	SPS	1
	STZ	1
	STZH	1
	STZD	1
	STN	2
	STNH	2
	STNF	1
	STND	1
	STO	1
	STOH	1
	STF	1
	STFM	*

ARITHMETIC		Clock Times	
	A	2	
	AI	2	
	AH	2	
	AIH	2	
	AF	5	
	AFD	5	
	AM	2	
	AMH	2	
	AMF	5	
	AMFD	5	
	S	2	
	SI	2	
	SH	2	
	SIH	2	
	SF	5	
	SFD	5	
	SM	2	
	SMH	2	
	SMF	5	
	SMFD	5	
	M	3	
	MI	3	
	MH	3	
	MIH	3	
	MF	4	
	MFD	6	
		R-even	R-odd
	D	29	18
	DI	29	18
	DH	18	18
	DIH	18	18
	DF	15	15
	DFD	25	25

*Determined by memory access time. An LF instruction requires six memory read cycles and an SF instruction requires six memory write cycles.

TABLE 1 (CONTINUED)

		Clock Times
LOGICAL	AND	1
	ANDI	1
	OR	1
	ORI	1
	XOR	1
	XORI	1
	EQC	1
	EQCI	1
	ANDD	1
	ORD	1
	XORD	1
	EQCD	1
SHIFT	SA	3
	SAH	3
	SAD	3
	SL	3
	SLH	3
	SLD	3
	SC	3
	SCH	3
	SCD	3
	RVS	6
ARITH. COMPARE	C	2
	CI	2
	CH	2
	CIH	2
	CF	2
	CFD	2
LOGICAL COMPARE	CAND	1
	CANDI	1
	COR	1
	CORI	1
	CANDD	1
	CORD	1

		Clock Times
TEST & BRANCH	BCC	-
	BRC	-
	BEC	-
	BAE	-
REG. MOD & TESTING	IBZ	2
	IBNZ	2
	DBZ	2
	DBNZ	2
	ISE	3
	ISNE	3
	DSE	3
	DSNE	3
	BCLE	2
	BCG	2
STACK	PSH	3*
	PUL	3*
	MOD	3*
SUB-ROUTINE	BLB	1
	BLX	1
ANALYZE	LEA	1
	INT	1
	XEC	-
CONVERSION	FLFX	5
	FLFH	5
	FDFX	5
	FXFL	4
	FXFD	4
	FHFL	4
	FHFD	4
NORMALIZE	NFX	3
	NFH	3
CALL	MCP	1
	MCW	1
VECTOR	VECT (See Vector Timing)	

* Stack Instructions take multiple passes through CP pipeline.

Table 2. CP Instructions grouped according to those which may follow one another without delay in the Arithmetic Unit.

GROUP 1a			GROUP 1b
L	AND	CAND	ST
LI	ANDI	CANDI	STH
LH	OR	COR	STR
LIH	ORI	CORI	STL
LR	XOR		STD
LL	XORI		STZ
LD	EQC		STZH
LMF	EQCI		STZD
LMD	ANDD	CANDD	STNF
LNF	ORD	CORD	STND
LND	XORD		STO
LNMF	EQCD		STOH
LNMD	BLB		MCP
LO	BLX		SPS
LAM	LEA		
LAC	INT		

NOTE: Multiple store instruction delay occurs when sequential store instructions write into different central memory octets for instructions in group 1b.

GROUP 2				
LM	A	S	C	IBZ
LMH	AI	SI	CI	IBNZ
LN	AH	SH	CH	DBZ
LNH	AIH	SIH	CIH	DBNZ
LNM	AM	SM	CF	STN
LNMH	AMH	SMH	CD	STNH

GROUP 3	GROUP 4	GROUP 5	GROUP 6
M	MH	MF	MFD
MI	MIH		

GROUP 7		GROUP 8	GROUP 9	GROUP 10	GROUP 11
AF	SF	D	DH	DF	DFD
AFD	SFD	DI	DIH		
AMF	SMF				
AMFD	SMFD				

Table 2 (Continued)

Instruction which cannot follow one another immediately on the next clock in the AU pipeline are listed below in groups 12 and 13. Instructions which do not use the Arithmetic Unit are listed in group 14.

GROUP 12

FXFL	FLFX	NFX	SA	SL	SC	RVS
FXFD	FLFH	NFH	SAH	SLH	SCH	BCLE
FHFL	FDFX					
FHFD			SAD	SLD	SCD	BCG

GROUP 13

PSH	XCH	ISE
PUL	MCW	ISNE
MOD		DSE
		DSNE

GROUP 14

LF	STF	LLA	BCC
LFM	STFM	XEC	BRC
			BAE
			BEC

NOTE: Instructions in groups 3 through 11 require long micro-op sequences in the Arithmetic Unit. The read-only-memory (ROM) in the Memory Buffer Unit generates these micro-sequences using a feedback arrangement from the ROM output register to the ROM address register. Due to the requirement of keeping the data and control of a given instruction in the same level of the CP pipeline at the same time, it is necessary to hold the next instruction (the one following the one with a long micro-sequence) in the input level of the MBU while the first instruction completes its micro-sequence in the ROM.

A second instruction's entry into the Arithmetic Unit will follow a first instruction into the AU on the clock in which the first instruction's result is placed in the AU output register.

The X and Y buffer registers, used for streaming vector operands into the Arithmetic Unit, can be used during scalar operations to retain the most recently used operand octets from CM. If a request for a word in CM is not contained in either the X or Y buffers, the octet of words containing the requested word replaces the contents of the X-buffer if Y was last used or replaces the contents of the Y-buffer if X was last used. An effective address request for a word in an octet which is presently contained in either the X or Y buffers is terminated at the buffer (the address is not sent to Central Memory) and the intended operand is read from the buffer in which it is resident. There is no pipe delay when the required operand is resident in either the X or Y buffer registers.

The algorithm for operating the X and Y buffers during scalar instructions is as follows:

If α = X, then set LUF = 0

If α = Y, then set LUF = 1

If $\alpha \neq$ X and $\alpha \neq$ Y and LUF = 1, then load X with (α) and set LUF = 0

If $\alpha \neq$ X and $\alpha \neq$ Y and LUF = 0, then load Y with (α) and set LUF = 1.

Example

This example is for a series of instructions which require operands from octets in the order a, a, b, b, a, b, c, d, b, d, d, c, d. $(LUF)^n$ represents the state of the last used file indicator at time n and set $(LUF)^{n+1}$ represents the next setting for the last used file indicator at time n+1.

Request for CM octet	$(LUF)^n$	X-buffer	Y-buffer	Set $(LUF)^{n+1}$
a	1	-	-	0
a	0	a	-	0
b	0	a	-	1
b	1	a	b	1
a	1	a	b	0
b	0	a	b	1
c	1	a	b	0
d	0	c	b	1
b	1	c	d	0
d	0	b	d	1
d	1	b	d	1
c	1	b	d	0
d	0	c	d	1
-	1	c	d	-

If two successive instructions request CM operands from different octets and neither one is resident in the X or Y buffers, then it is possible for both requests to be issued to CM before the IPU needs to be stopped to wait for CM access. This provides overlap of CM requests, rather than having to wait the entire memory cycle time for each memory octet fetch. The second octet request can be placed on the CM address bus two clocks after the first octet request. The second octet data will be available in the second buffer two clocks after the first arrives providing that no memory conflicts occurred and that the second read was from an alternate memory module than the first. If the two read requests were to the same stack, then an additional two clocks will elapse before the second read data is available at the buffer register due to memory stack conflict.

A third read request in a string of sequential instructions, for which the first two octet addresses were different, will be held in level 3 of the IPY while the second read instruction waits in level 4 for the data to return from the first instruction's read request. The first instruction was advanced to the MBU input level while the CM request was being processed. It cannot proceed past this level because the selection of the particular operand word from the X or Y buffer is accomplished at the MBU input level and the selection cannot be performed until the data is available.

An instruction can proceed to the AU without memory delay if the required operand is presently residing in either the X or Y buffer registers.

REGISTER CONFLICT DELAY

A register conflict delay occurs whenever an instruction requires the contents of a register (base, index, general arithmetic, or vector parameter) and that register is presently in the process of being modified by a previous instruction which has not yet passed through the AU output level. Register conflicts occur because of the pipeline nature of the Central Processor. A register conflict delay (RCD) can occur at any of the first three levels of the IPU; the Instruction Register (IR) level 1, the Preindex (XR) level 2, or the Index (AR) level 3.

An RCD at the Instruction Register level is created when an instruction attempts to select a base or index register to develop an effective address and finds that the base register specified by the M-field or the index register specified by the X-field is presently in the process of being modified by a previous instruction somewhere downstream in the CP pipe. The IR level conflict is relieved only after all instructions which specify either of these two registers as a target register address have entered their results into the register file. The time required to relieve this conflict depends upon a consideration of scalar timing factors 1, 2, 3, 4, 5, and 9 for all instructions below and including the conflict register modifying instruction which exists downstream in the pipeline, if an analysis is to be made to determine the time required to relieve the register conflicts.

An RCD at Index level 3 is created when an instruction specifies the use of a register operand which is presently in the process of being modified by a previous instruction. Such a conflict is relieved when the instruction specifying the conflicting register, as a target register address, has entered its result into the register file. Scalar timing factors 1, 2, 3, 4, 5, and 9 must again be considered for all instructions below and including the conflict register modifying instruction if an analysis is to be made to determine the register conflict delay.

An RCD at the Index level 3 is also created when an instruction is encountered for which the effective address specifies a register operand and another instruction downstream is in the process of modifying that register location. The effective address specifies a register operand whenever an instruction is encountered for which the M-field ecuals zero, the indirect bit equals zero, and the effective address α is less than or equal to 2F hex. The conflict is relieved when the instruction specifying the conflicting register, as a target register address, has entered its result into the register file. Scalar timing factors 1, 2, 4, 5, and 9 must be considered in order to make an analysis of the time required to relieve the register conflict.

Example

This is an example of an instruction which requires a general arithmetic register operand which is presently in the process of being modified by a floating point add instruction which is currently at the MBU output register level. There are no delays in the pipeline below the MBU output level which would cause the pipe to halt momentarily and no Operand Fetching delay for the register modifying instruction since it has already fetched its operand from the X or Y buffer and has entered the operand into the MBU output register.

For the timing analysis with all the other possible delays assumed non-existent: Count the number of pipe sections from the register modifying instruction to the register file. The floating point add instruction must pass through the following registers:

1. AU Receiver
2. Exponent Subtract section
3. Align section
4. Add section
5. Normalize section
6. AU output section

On the seventh clock the conflicting general arithmetic register is loaded with the result of the floating point add instruction. Pipeline flow can now continue on the 8th clock when the required register operand is entered into the RØ register of level 4 of the IPU. If the conflict had not existed, then RØ would have been loaded on clock 1. Therefore, this register conflict delay caused a loss of seven clock times.

MULTIPLE STORE INSTRUCTION DELAY

A multiple Store instruction delay is caused when two or more Store instructions, all with different octet addresses, occur consecutively or with only one instruction separation in an instruction stream. The MBU and AU pipe sections are provided with one address register to retain the octet address of one store instruction. A second store instruction occurring in a rapid sequence will be delayed at level 3 of the IPU until the first store instruction of the sequence has passed to the AU output level.

A delay due to Central Memory write conflicts may also occur for any two or more Store Instructions which are too closely spaced, but that is a different type of delay than the Multiple store instruction delay being considered here. The multiple store delay has a tendency to ease the memory write conflict problem, since the pipeline operates at a reduced speed when the multiple stores are detected.

There is no such multiple store delay for a series of two or more consecutive store instructions which all address the same octet or which write consecutively into monotone increasing or decreasing address locations.

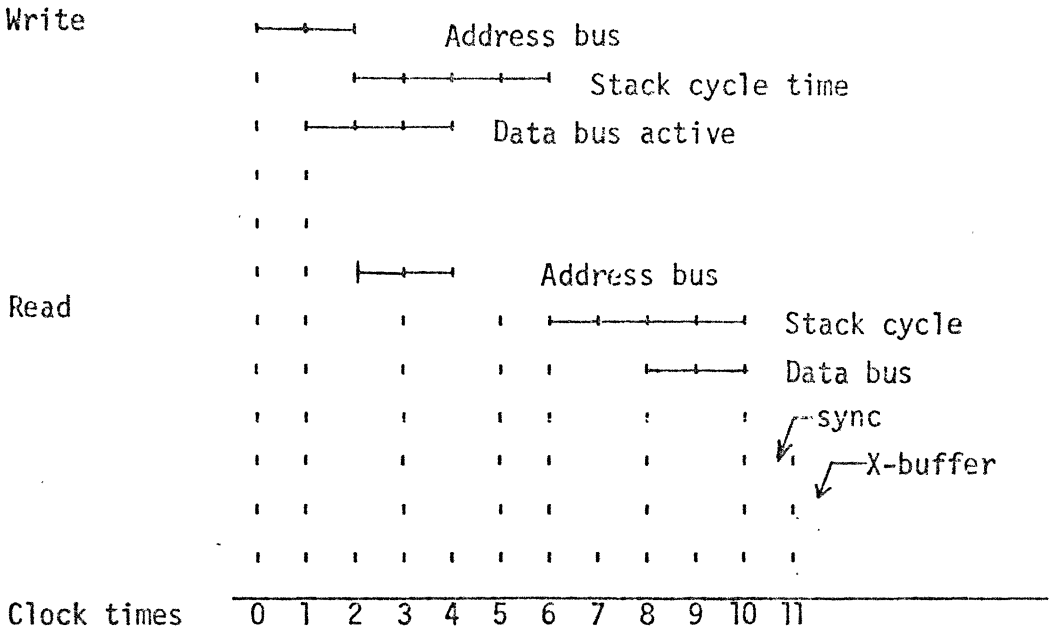
READ AFTER WRITE DELAY (Same Octet)

This delay is caused by attempting to read from a Central Memory location while a previous write instruction is still "in the process" of writing into the same location or into the same memory octet. A write instruction is "in the process" of writing into memory if it is anywhere below the IPU, but not yet into central memory.

It is possible to acquire a modified operand over the Z to X update path providing that there are no other stores into different memory octets which exist between the store instruction whose octet address agrees with the operand read octet address of the instruction presently at level 3 of the IPU. The update from Z to X occurs after all stores into the agreeing octet (which are in either the MBU or AU) have been entered into the Z-buffer. The update may occur simultaneously with the arrival of the read data from CM. In which case, the read data must be merged with the update information from Z.

A wait for CM writing occurs if there is agreement as above, but another store into a different octet exists between the agreeing store and the operand read instruction at AR of the IPU. A read from the octet address of AR is made even though the address in ZBA agrees with AR, since the write from ZBA will arrive in memory before the read request is received.

The memory timing for a write then read in the same memory module is



INDIRECT ADDRESS GENERATION TIME

Each level of indirect addressing requires 10 clock times assuming no memory conflicts.

EXECUTE INSTRUCTON FETCHING

Each Execute instruction fetch requires 10 clock times assuming no memory conflicts. The only difference between Executes and indirects is that an Execute instruction is fetching an instruction to be executed whereas an indirect reference is fetching the address of an operand or the address of the address of an operand, etc.

INSTRUCTION FETCHING AFTER BRANCHING

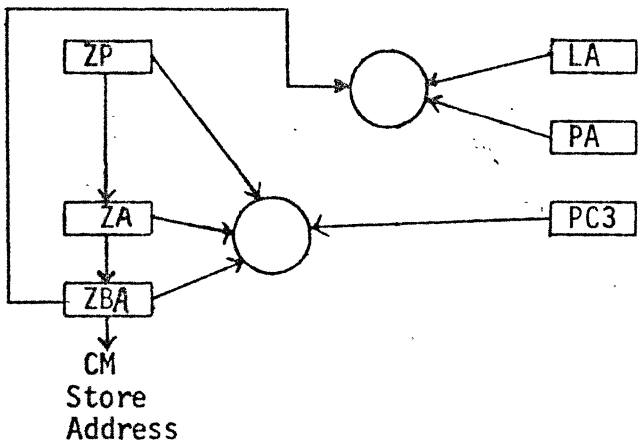
A delay equivalent to that of indirect or Execute occurs when fetching instructions following a branch without look-ahead.

INSTRUCTION HAZARD CONDITION

An instruction hazard condition exists when a Store instruction is in the process of modifying a word in a central memory octet from which instructions previously read are currently residing in the Instruction Processing Unit (IPU). These instructions are then the "old" commands and hence the Central Processor must prevent their being executed.

Prevention of execution is accomplished simply by cancelling the instructions above the point in the IPU where instruction addresses agree with store addresses. In order to restart the program, the CP must wait until the culprit Store instruction has completed writing into CM, then the IPU must recall the instructions which were cancelled.

Detection of an instruction hazard condition is accomplished by comparing the three store address registers (ZP, ZA, and ZBA) with the program counter value at level 3 in the IPU and comparing the ZB address with the instruction look-ahead and present address registers. These five comparisons detect all instruction hazards. When an octet address agreement occurs in any of the five comparators, an instruction hazard condition exists. The comparisons are diagrammed below:



Indirect address chains are sent through PC3 for checking to be sure that no stores are presently writing into a location which is currently being used for indirect addressing.

An indirect addressing instruction must wait for an immediately preceding Vector operation to terminate before the indirect addressing operation can proceed. This prevents an indirect operation from taking an "old" indirect linkage when the preceding Vector operation is modifying the indirect address file in Central Memory.

SHORT CIRCUIT PATH

A short path exists from the AU output register to the AU receiver register. This path is used whenever an instruction occurs in an instruction stream for which the immediately preceeding instruction has made reference to the same register of the CP register file. In this instance the preceeding instruction must be one which will store into the same register that the succeeding instruction requires as its register operand. When this condition arises, the succeeding instruction will not wait for the normal register conflict delay, but instead will proceed down the Central Processor pipeline without its register operand and will acquire the operand via the AU short circuit path upon the succeeding instructions arrival at the Arithmetic Unit.

Short circuit (SC) condititions can only occur for adjacent pairs of instructions of the same word size (double with double, single with single, etc.) SC conditions can occur for an unlimited number of instructions in a chain as long as they all use the same register and have the same word size. Any single instruction which does not use the same register will break the chain. Also, two successive branch instructions which use the branch test level to determine the outcome of the branch condition cannot use the SC path to achieve a faster branch decision for the second of the two branches because the branch test level does not have a path equivalent to the AU short circuit path. The branch test level does not solve for the value of the argument in an Increment Test and Branch instruction, but rather determines only whether the branch test passes or fails.

The timing for the SC path can be determined by following the first of the pair of instructions down the pipeline to the AU output level. The time at which the result of the first instruction arrives at the AU output can be determined. The second instruction will advance to the AU receiver level and wait there if it has arrived before the first instruction's result is available from the AU output. One clock is used to route the AU result back to the AU receiver level. The second instruction advances through the AU when all of its required arguments are supplied at the AU input.

Vector priming is divided into four distinct processes, which are:

1. Vector parameter file fetch
2. MBU initialization
3. Memory operand fetch
4. AU fill

When a vector instruction terminates, a process of AU emtying occurs. This process involves only the depletion of operand arguments from the Arithmetic Unit.

VECTOR PARAMETER FILE FETCH

Vector parameter file fetching occurs as a result of specifying a vector instruction for which a new VPF is requested from central memory. The new VPF is requested when the effective address of the vector instruction has been developed by the normal IPU indexing hardware. This hardware does the pre-indexing and index addition required for generating the effective address.

VPF fetching (1) begins after the vector instruction has reached the index addition level (level 3) and the VPF address has been developed. VPF fetching requires 8 clock periods. However, this CM request is overlapped in time with the previous scalar instructions presently being processed downstream by the Arithmetic Unit. Loading of a new VPF appears no differend to the Instruction Processing Unit (IPU) than if the IPU had encountered a scalar Load Register File instruction (LF). The fetching of data for these files is carried out entirely by the IPU and the MBU has not been involved up until now with the vector instruction. Also, the memory fetching time for the VPF will be seen to be less than the time required for memory operand fetching (2) because the register files have a simpler interface with memory than the interface which exists at the Memory Buffer Unit.

Overlapping memory cycles between the IPU and MBU during VPF fetch will exist if the scalar instruction immediately prior to the vector instruction requests a central memory operand from a new octet and all previous scalar memory requests have been granted (data received from memory). For if this condition exists, the scalar instruction requesting the new octet is allowed to advance beyond level 3 (index addition level), providing that "path ahead" is clear, and level 3 is filled with the developed address for the VPR request of the vector instruction. Thus, the time required to fetch the VPF is completely overlapped by the time required for fetching an operand of a prior scalar instruction. The VPF fetching is essentially free in this case.

VPF FETCH NOT REQUESTED

The vector instruction could be one which specifies the use of the vector parameter data currently residing in the VPF registers. In this case, there is no vector parameter file fetch cycle required and the vector priming operation proceeds immediately to the MBU initialization process (2).

MBU initialization begins upon detection of a vector instruction at level 4 of the IPU. This MBU initialization is then overlapped with previous scalar instruction processing going on downstream in the Arithmetic Unit.

MBU INITIALIZATION

Initialization of the Memory Buffer Unit (MBU) involves the transferring of vector parameter data from the VPF registers in the IPU to the vector working registers of the MBU. This process begins immediately after new data has been entered into the VPF registers, if a VPF fetch is specified; or immediately upon detection of a vector instruction in level 4 of the IPU if a request for the current VPF data is specified.

MBU initialization requires 10 clock periods. This time begins with the starting address development in the IPU for vectors A, B, and C, (in that order). Development utilizes the pre-index and index addition levels (levels 2 and 3) of the IPU. Then the remaining five words of the vector parameter data is transmitted a singleword at a time to the MBU for distribution to its working registers that control the vector operation.

There would be no advantage gained by transmitting the remaining inner and outer loop increment information to the MBU at a faster rate, since the memory operand fetch operation (3) is overlapped with the transmission of the remaining data. The transmission of VPF data is completed seven clocks before the first operand arguments are available as inputs to the AU receiver register, even though the VPF data is sent only one word at a time.

MEMORY OPERAND FETCH

This cycle begins five clocks after MBU initialization began. It is considered to start at the time when the first address reaches the central memory address requestor in the MBU. Although by this time the A address generator has produced the first three element addresses of vector A.

Memory operand fetching of the first octet of data for vector A and B is completed when the first two operand arguments are placed in the MBU output registers and are available as inputs to the Arithmetic Unit. The process of first operand octet fetch requires 12 clock periods. Subsequent octet fetches are requested at 8 clock intervals during the self loop and the pipeline flow of operands to the AU is maintained throughout the vector operation. The initial operand fetch is an overhead penalty which occurs only once during the vector priming procedure.

AU FILL

The Arithmetic Unit can proceed to fill its internal pipe sections as soon as the operand stream is presented to its input registers from the MBU. An AU receiver register accepts the operands from the MBU, which have been transmitted between physical cabinets containing the MBU and the AU. The receiver register therefore adds one clock time to the AU operation times given for scalar instructions in the timing section for scalars. This figure (scalar AU time plus one) gives the number of clock intervals before the first result appears at the Arithmetic Unit output. Scalar AU times vary from instruction to instruction, but once the AU has been filled in a vector mode, AU results are produced at one clock intervals for most single length vector instructions. The exceptions to this rule appear in Table II which lists the vector flow rates for all vector instructions.

AU EMPTY

At the termination of a vector instruction the AU will exhaust the final results into the Z-buffer registers and a Z-write operation is forced to purge the output buffers of the vector results, so that scalar hazard detection can begin fresh.

However, when a scalar instruction follows a vector, overlapping occurs again. Two general constraints need to be listed here, though: (a) If the subsequent scalar instruction uses indirect addressing, it will wait in level 1 until the vector operation is completely terminated. This prevents an erroneous indirect addressing linkage through an area of central memory which is being modified by a vector instruction. (b) If the vector instruction is of the class requiring the storage of an item count at the completion of each self loop, then a subsequent scalar instruction must wait at level 1. Vectors which store item counts use the 2nd and 3rd levels of the IPU to restart the vector in case a context switch operation prematurely terminated the vector.

Overlapping of the subsequent scalar begins after the MBU has determined that all self, inner, and outer loops are completed and that the ZB register has initiated its last write cycle for the vector instruction. At this time the subsequent scalar may use the facilities of the MBU to request a memory operand required for scalar instruction execution. Thus, the requesting of a next scalar octet is overlapped with the termination (AU empty) of the present vector instruction in the AU.

In conclusion, vector timing can be computed approximately from the formula T = P + R · L · NI · NØ. If either NI or NØ equal zero, replace appropriate term with value 1.

where P = Vector prime

R = Vector Rate (clocks/element) from Table 3

L = Vector dimension

NI = Inner loop count

NØ = Outer loop count

T = Time in clock periods

P is broken up into the following approximate times (in clocks):

VPF fetch = 8

MBU Initialization = 5 (overlap adjusted)

Operand fetch = 12

AU fill = 1 + Scalar AU time

If the vector instruction uses the current vector parameter file, then do not add in the time for the VPF fetch.

TABLE 3. VECTOR INSTRUCTION TIMING

Rate is defined as the number of clock times required to obtain each element of the result, if a vector result exists. For scalar results, the rate is defined as the average number of clock times required to perform each instruction.

Timing for some of the vector instructions is based on input rate rather than output rate from AU. These vector instructions are the ones which produce infrequent results at the AU output. They include:

Vector Dot Products
Vector Peak Picking
Vector Searches
Vector Comparisons

The rates given in the table for these instructions correspond to the rate at which the inputs change.

Vector Instruction		Rate
ADD	VA	1
	VAH	1
	VAF	1
	VAFD	1.75*
ADD MAGNITUDE	VAM	1
	VAMH	1
	VAMF	1
	VAMFD	1.75*
SUBTRACT	VS	1
	VSH	1
	VSF	1
	VSFD	1.75*
SUBTRACT MAGNITUDE	VSM	1
	VSMH	1
	VSMF	1
	VSMFD	1.75*
MULTIPLY	VM	1
	VMH	1
	VMF	1
	VMFD	3

Vector Instruction		Rate
DOT PRODUCT (Input Rate)	VDP	1
	VDPH	1
	VDPF	1
	VDPD	4

		SV=0	SV=1
DIVIDE	VD	18	8
	VDH	9	9
	VDF	8	8
	VDFD	18	18

LOGICAL	VAND	1
	VOR	1
	VXOR	1
	VEQC	1
	VANDD	1.75*
	VORD	1.75*
	VSORD	1.75*
	VEQCD	1.75*
SHIFT	VSA	2
	VSAH	2
	VSAD	2

* See doubleword timing in Table 4.

Vector Instruction		Rate
SHIFT CON'D	VSL	2
	VSLH	2
	VSLD	2
	VSC	2
	VSCH	2
	VSCD	2
MERGE (INPUT RATE)	VMGH	2
	VMG	2
	VMGD	2
ORDER (OUTPUT RATE)	VO	2
	VOD	2
	VOF	2
	VOFD	2
ARITHMETIC COMPARISON (Input Rate)	VC	1
	VCH	1
	VCF	1
	VCFD	1.75*
LOGICAL COMPARISON (Input Rate)	VCAND	1
	VCANDD	1.75*
	VCOR	1
	VCORD	1.75*
SEARCH (Input Rate)	VL	1
	VLH	1
	VLF	1
	VLFD	1.75*

Vector Instruction		Rate
SEARCH CON'D	VLM	1
	VLMH	1
	VLMF	1
	VLMFD	1.75*
	VS	1
	VSH	1
	VSF	1
	VSFD	1.75*
	VSM	1
	VSMH	1
	VSMF	1
	VSMFD	1.75*
PEAK PICKING (Input Rate)	VPP	1
	VPPH	1
	VPPF	1
	VPPFD	1.75*
CONVERSION	VFLFX	2
	VFLFH	2
	VFDFX	2
	VFXFL	2
	VFXFD	2
	VFHFL	2
	VFHFD	2
NORMALIZE	VNFX	2
	VNFH	2

* See doubleword timing in Table 4.

ASC VECTOR TIMING FOR SINGLE-VALUED VECTORS

This table shows memory limitations for the different cases using single-valued vectors. The vector instructions that refer to this table are the ones which would require only one AU clock per element if they were not restricted by memory speed.

	TIME IN CLOCKS PER ELEMENT	
ABC	HALF-WORD OR SINGLE-WORD	DOUBLEWORD
VVV	1	1.75
VSV, SVV	1	1.25
VVS	1	1
VSS, SVS	1	1

Note: V represents directly addressed vectors.

S represents directly addressed single-valued vectors or immediate single-valued vectors.

TABLE 4.

CENTRAL PROCESSOR

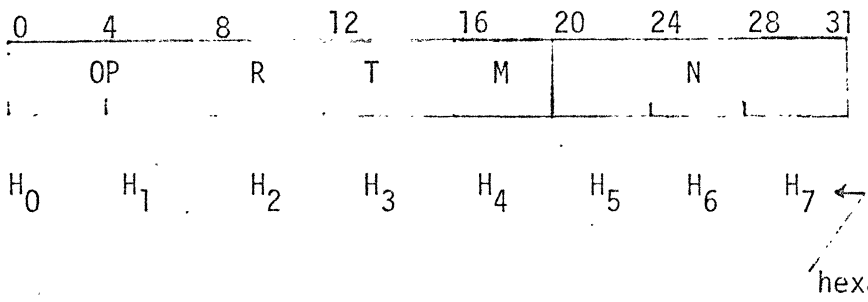
INSTRUCTION SET

TABLE OF CONTENTS

INSTRUCTION FORMAT

The instruction word of the Central Processor contains 32 bits and is divided into five fields:

Field Name	Bit Positions	Field Size	Function
OP	0-7	8	Operation Code
R	8-11	4	Register address
T	12-15	4	Address modifier tag
M	16-19	4	Base address designator
N	20-31	12	Displacement address



- OP-Field

  The OP-Field specifies the machine instruction to be executed.

- R-Field

  The R-field addresses one of 16 registers from the arithmetic, base, or index register group.

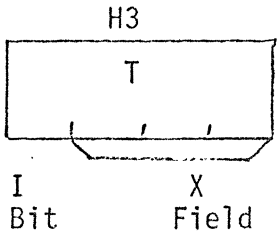
- T-Field

  The T-Field is an address modifier tag that has the following interpretation:

T	Addressing Type	Virtual Address, α, of Memory Operand
0	Direct address	N +(M)
1-7	Indexed address	N + (M) + (T)
8	Indirect	(N + (M))
9-F	Indexed indirect address	(N + (M) +(T-8))

A symbol or expression enclosed by parentheses () represents "the contents of".

The T-field may be decomposed into an I- bit and an X-field, where the most significant I-bit designates indirect addressing and the 3-bit X-field specifies one of seven index registers used in the indexing operation. The index registers are physically assigned to register file address locations 21 through 27 (hexadecimal). A special set of index instructions is used to load, store, modify, and test the index registers.



Displacement indexing is provided such that the indexing operation is compatible with word size; i.e., the index registers are automatically aligned according to word size. If an index register contains the value K, the K^{th} element of an array is accessed, whether it is a halfword, word, or doubleword.

- M-Field

  The M-field is a base register designator. It is used to extend the addressing range capability of the ASC to a potential 16.7 million words. The M-field selects one of fifteen 24-bit base registers to be added to the N-field displacement before indexing or indirect addressing. No base addressing is used when M equals 0.

- N-Field

  The N-field is the address displacement relative to the base address contained in M.

The M- and N-fields also may be interpreted as immediate operands when immediate instructions are specified by the operation code.

ALPHA ADDRESS DEVELOPMENT

There are two basic methods of developing an address in the Central Processor. The two methods are referred to as α addressing and β addressing. Central Memory operand requests use address development. Instruction branching uses β address development. Both of these methods may use base and index modification. The table of addressing types on page 1 has reference to addressing and represnets the addition of base and index registers for singleword addressing as follows:

- No base addressing — M = 0
  No indexing — X = 0

0	0	0	H_5	H_6	H_7	N-field (12 bits)
V_2	V_3	V_4	V_5	V_6	V_7	virtual address (24 bits)

- No base addressing — M = 0
  Indexing — X = k for k = (1,2,3,...7)

0	0	0	H_5	H_6	H_7	N-field (12-bits)
$+I_2$	I_3	I_4	I_5	I_6	I_7	index register k (24-bits)
V_2	V_3	V_4	V_5	V_6	V_7	virtual address (24-bits)

- Base addressing — M = b for b = (1,2,3,...,15)
  No indexing — X = 0

0	0	0	H_5	H_6	H_7	N-field (12-bits)
$+B_2$	B_3	B_4	B_5	B_6	B_7	base register b (24-bits)
V_2	V_3	V_4	V_5	V_6	V_7	virtual address (24-bits)

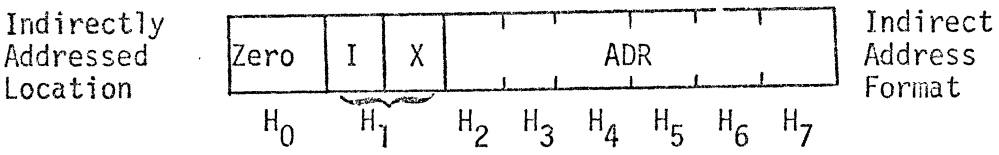
- Base addressing — M = b for b = (1,2,3,...,15)
  Indexing — X = k for k = (1,2,3,...,7)

0	0	0	H_5	H_6	H_7	N-field (12-bits)
$+B_2$	B_3	B_4	B_5	B_6	B_7	base register b (24-bits)
$+I_2$	I_3	I_4	I_5	I_6	I_7	index register k (24-bits)
V_2	V_3	V_4	V_5	V_6	V_7	virtual address (24-bits)

For the cases when M = 0, a virtual address in the range 00 through 2F(hex) is interpreted as an absolute register address. If M = b, where (b) = 0, the corresponding virtual address range refers to Central Memory locations 0 through 2F.

When the indirect bit (I-bit of hex. character H3) of an instruction (not an immediate instruction) is a "one", then the α address developed by the instruction references a location either in central memory or in the register file depending on the M-field and the range of α. A register from the register file is referenced if $\alpha \leq 2F$ and M = 0.

The location addressed by an instruction using indirect addressing is interpreted according to the format:



ZERO is the no-op code (4-bits)
I is the indirect flag (1-bit)
X is the index tag (3-bits)
ADR is a full 24-bit address

The base registers are not used after the original indirectly addressed instruction is interpreted. Multilevel indirect addressing is provided with independent indexed and/or indirect addressing at each level.

Indirect addresses, using the above format, always reference Central Memory single words. The terminal indirect address (I=0) is indexed by displacement indexing according to instruction word size when $X \neq 0$, but the operand acquired is always from Central Memory.

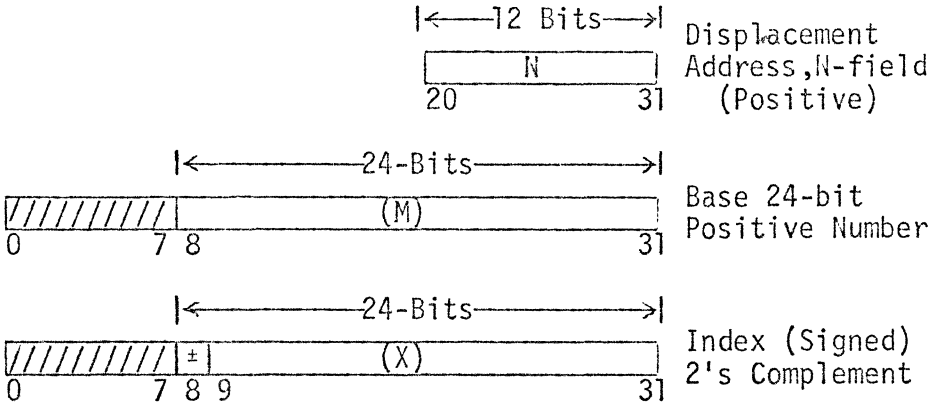
An indirect address memory request is tagged as an execute request when transmitted to central memory.

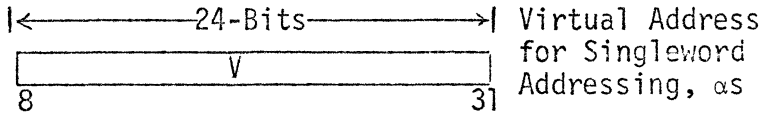
An illegal operation code program interruption occurs if the 4 most significant bits are not all zeros at the location specified by an instruction with an indirect flag equal to one or at the location specified by a multilevel indirect addressing chain where the address is specified to be another indirect address.

DISPLACEMENT INDEXING OF ALPHA ADDRESSES

SINGLEWORD ADDRESSING, α

For singleword addressing, the index value is a signed two's complement number where the sign is in bit position 8 of the index register. The fraction is in the remaining 23 least significant bits of the index register (bit positions 9 through 31). The N-field and base registers are interpreted as positive 12- and 24- bit numbers, respectively.





The addition of a positive index to a large base may result in "wrap around" to a low virtual address. Also, addition of a negative index to a small base plus displacement may result in a "negative wrap around" to a high virtual address

Wrap around as just described will occur only if the maximum size memory to contain the full 24-bit address range is connected to the system. Any central memory address outside the address limit of the physical memory will result in a memory protection violation. The wrap around and address limit are normally thought of in terms of singleword addresses, but apply equally well to halfword and doubleword addressing.

HALFWORD ADDRESSING, α_h

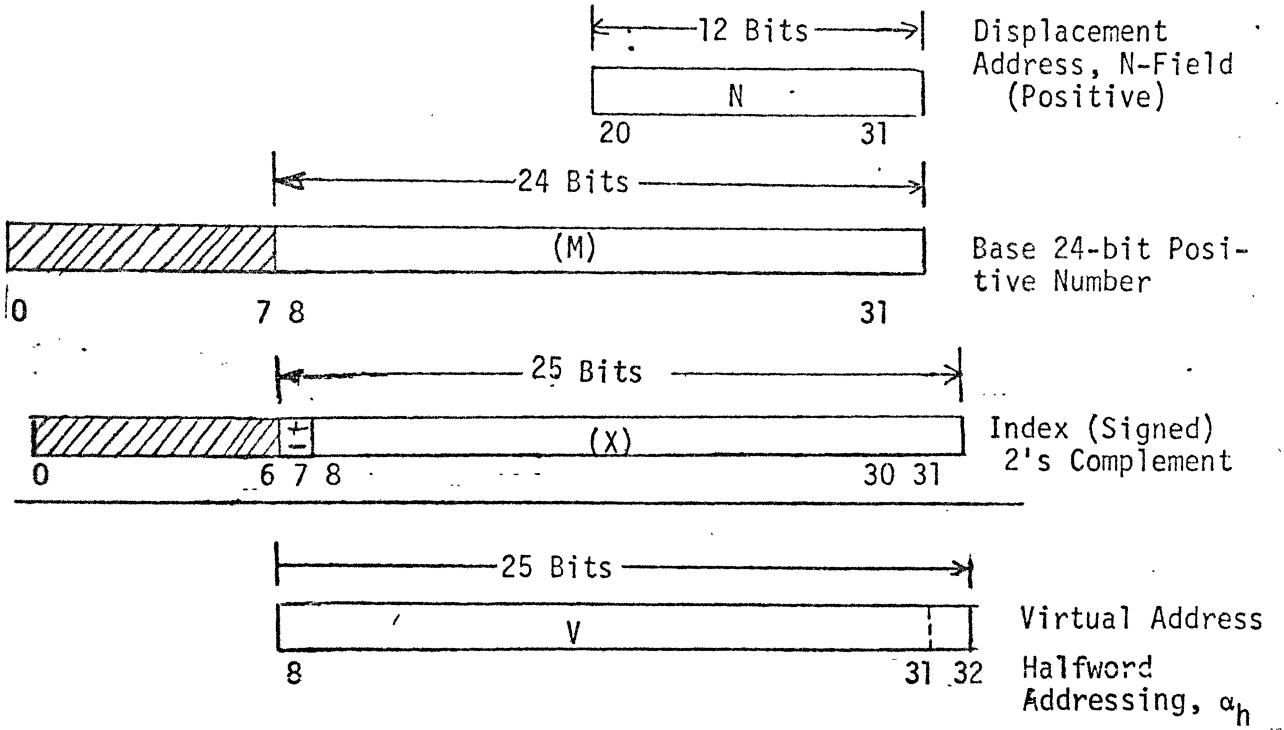
Displacement indexing is used for halfword addressing. When a halfword address is not indexed, the left halfword of a Central Memory singleword is selected. An odd index value addresses halfwords in the least significant half (right half) of a Central Memory word. An even index value addresses the left halfword of a Central Memory singleword. This is true for all halfword instructions, except for four special halfword load and store instructions (LR, LL, STR, and STL).

The LR, LL, STR, and STL instructions address the right half of a Central Memory singleword when not indexed. If indexed, an even index value selects words from the right half of a Central Memory singleword. An odd index value addresses the left halfword. When an array is addressed consecutively by indexing with one of the four special halfword load/store

instructions, an even index value addresses the right half of a Central Memory singleword, as just mentioned; but it should be noted that when this even index value is incremented by one (forming an odd index value), the memory operand acquired by this instruction is from the left half of the next consecutive Central Memory singleword.

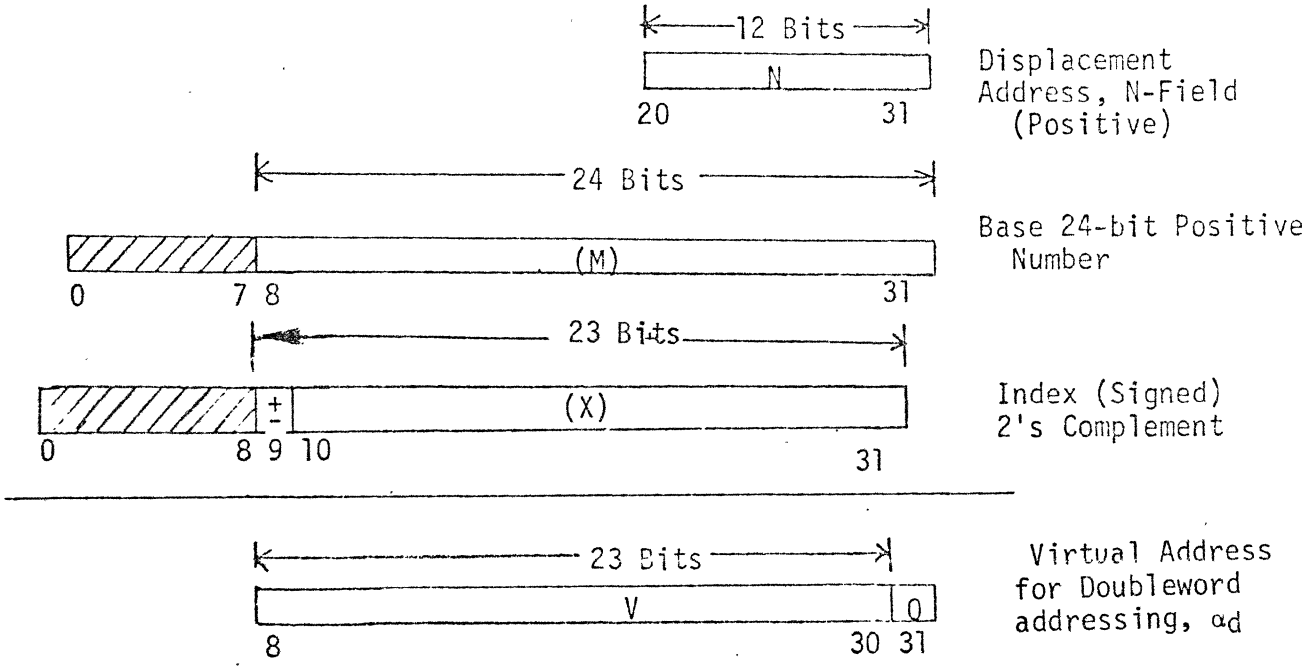
The index unit in the CP hardware accomplishes displacement indexing of halfwords by shifting the index register one bit position to the right before the base (M) and displacement N-field are added. The least significant bit of the index value effectively determines whether the left or right half word of a Central Memory singleword is addressed. A 25-bit virtual address is generated for halfword instructions. The index value is interpreted as a signed two's complement number. For halfword addressing, the sign of the index value is in bit position 7, and the fraction is in the remaining 24 least significant bits of the index register (bit positions 8 through 31). The N-field is interpreted as a positive 12-bit number. The base (M) is interpreted as a positive 24-bit number.

The addition of a positive index to a large base may result in "wrap around" to a low virtual address. Also, addition of a negative index to a small base plus displacement may result in a "negative wrap around" to a high virtual address.



DOUBLEWORD ADDRESSING, α_d

The index register is displaced one bit position to the left relative to the base (M) and displacement address N before being added in the index unit. The least significant bit of the sum is forced to 0, and the remaining 23 bits of the sum address a doubleword in Central Memory. If both the base and displacement address are odd, a carry will be generated in the least significant bit position of the sum and will produce a doubleword address one greater than the address obtained when either the base or displacement address (or both) has a value diminished by 1.



The index value is interpreted as a single two's complement number. For doubleword addressing, the sign bit is in bit position 9, and the fraction is in the remaining 22 least significant bits of the index register (bit positions 10 through 31). The N-field is interpreted as a positive 12-bit number. The base (M) is interpreted as a positive 24-bit number. Displacement indexing allows one to address the Kth doubleword in a data array by an index value equal to K.

Doublewords are always selected from and stored into even-odd singleword memory address pairs and registers address pairs.

BRANCH ADDRESSING

The branch address, β, for Branch instructions are a function of the T, M, and N-fields of the instruction word as follows:

T	M	Branch Address, β (Singleword Addressing)	
0	0	N*+(PC)	Relative to program counter
1-7	0	N*+(PC)+(T)	Relative to program counter plus index
0	1-F	N+(M)	Base plus displacement
1-7	1-F	N+(M)+(T)	Base plus displacement plus index
8	0	(N*+(PC))	Indirect relative to program counter
9-F	0	(N*+(PC)+(T-8))	Indirect relative to program counter plus index
8	1-F	(N+(M))	Indirect relative to base plus displacement
9-F	1-F	(N+(M)+(T-8))	Indirect relative to base plus displacement plus index

where N + (M) is Base address plus displacement (N is positive, 12-bit number) and N* ≡ Signed N-field, 11-bits plus sign bit, 2's complement.

This branch address definition is used for all test and branch instructions.

These include:

BE, BG, BGE, BL, BLE, BNE, B
BCZ, BCO, BCNM, BCM, BCNO, BCNZ
BZ, BPL, BZP, BMI, BZM, BNZ
BRZ, BRO, BRNM, BRM, BRNO, BRNZ
BU, BO, BUO, BX, BXU, BXO, BXUO, BD
BDU, BDO, BDUO, BDX, BDXU, BDXO, BDXUO
BXEC, BLB, BLX
IBZ, IBNZ, DBZ, DBNZ

When an indirect branch address is specified (T ≥ 8), the indirect address format is the same as that used by indirect α addressing, except that addresses less than 2F reference central memory regardless of M.

If a branch address is less than or equal to 2F (β ≤ 2F), then the program branches to central memory location β regardless of the M and T-field specifications. Branches cannot reference the register file.

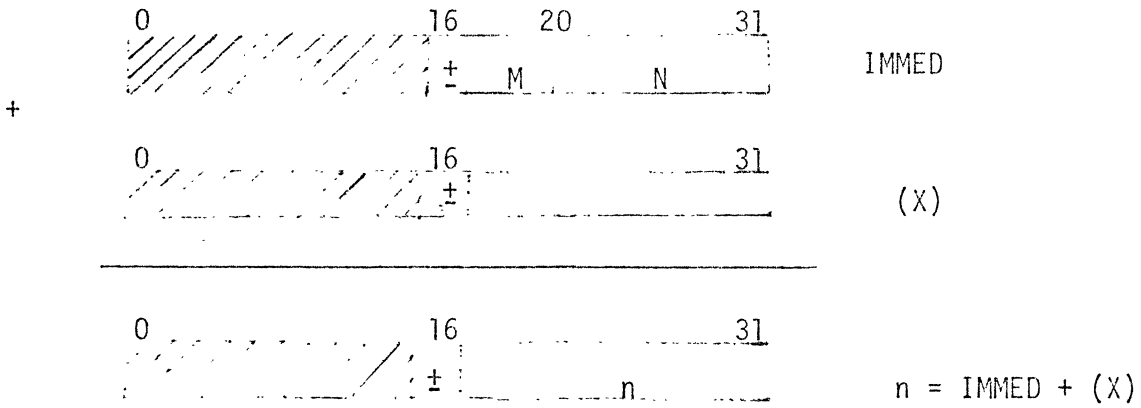
IMMEDIATE OPERANDS

Immediate operands have the following characteristics:

a) Halfword Immediate Operand Instructions

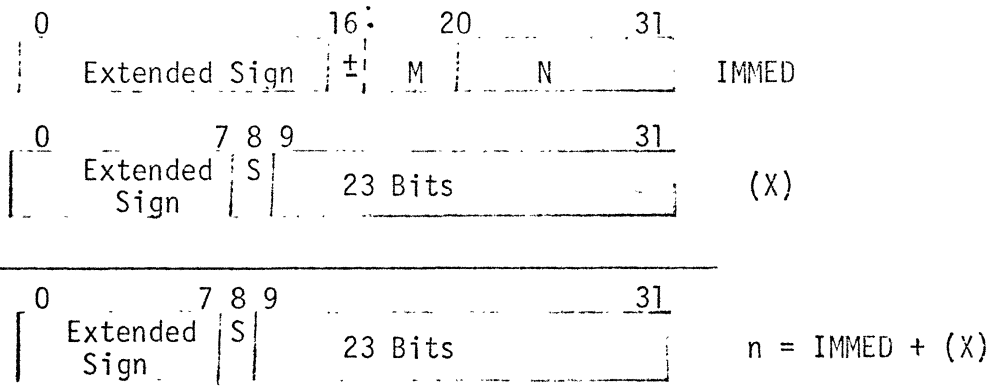
The combined M and N-fields form the immediate operand for halfword instructions. The MSB of the right half of the instruction word is the sign bit. Negative numbers are represented in two's complement form.

This immediate operand can be modified by the right half of index register X. If X ≠ 0, the index register specified by X is added to the M and N fields. For this case, the 16th bit position of index register X is a sign bit. If X = 0, no modification occurs.



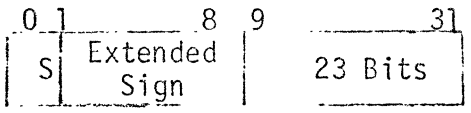
b) Singleword Arithmetic Immediate Operands

Single word length immediate operands for arithmetic instructions are formed from the combined M and N-fields of the instruction word with extended sign (two's complement representation for negative numbers). The left half of IMMED consists of the extended sign of the most significant bit of the right half of IMMED. This immediate operand can be modified by an index register when X ≠ 0. For this case, the contents of index register X are interpreted as a signed number (two's complement representation for negative numbers) within the range $-2^{23} \leq (X) \leq 2^{23} - 1$. If X = 0, no modification occurs.



In effect, the sign bit in the 8th bit position of the contents of index register X is extended into the most significant eight bits (bit positions 0 through 7) before being added to IMMED. The true 32-bit value contained in index register X remains unchanged; the sign extension occurs in the index unit hardware and not in the register file. The modified immediate operand n is restricted to the range $-2^{23} \leq 2^{23} - 1$, since the parallel adder in the index unit is only 24 bits wide. The sign bit in the 8th bit position of n is extended into the most significant eight bits of n before being used as a modified immediate operand by the Arithmetic Unit.

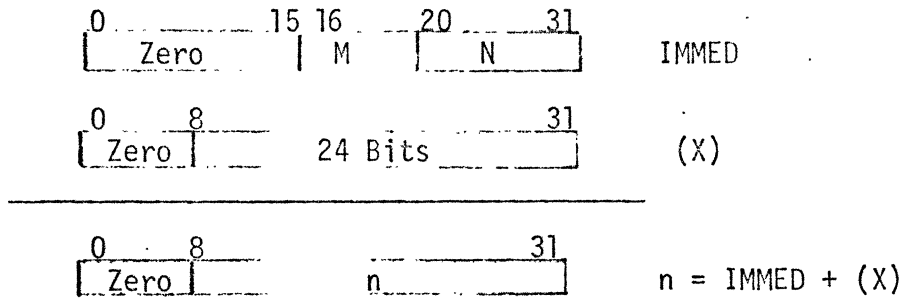
The Arithmetic Unit interprets this singleword immediate operand as though the sign bit were in the most significant bit position as shown below:



n = IMMED + (X)

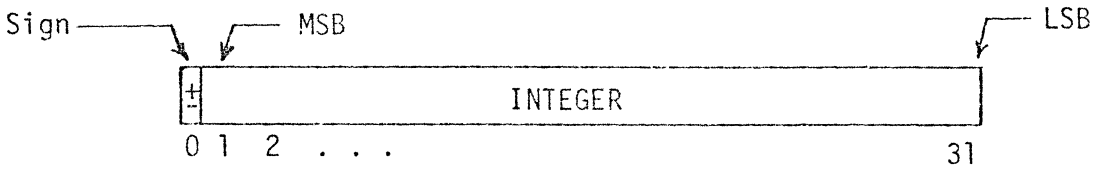
c) Singleword Logical Immediate Operand

A singleword immediate operand for logical instructions is formed from the combined M and N-fields of the instruction, except tnat the left half of IMMED consists of 0's instead of the extended sign. When $X \neq 0$, the 24 least significant bits of index register X are added to IMMED. If $X = 0$, no modification occurs.



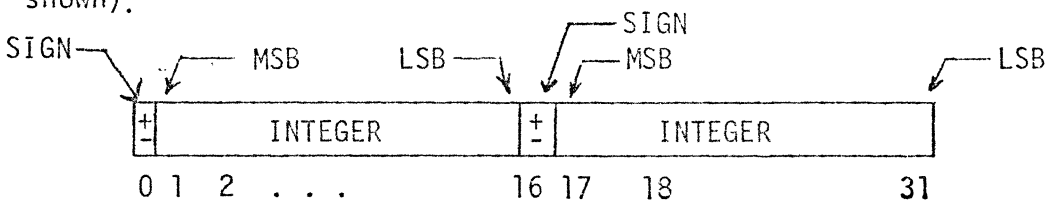
DATA FORMATS

1. Fixed point, single length, 32-bit word.



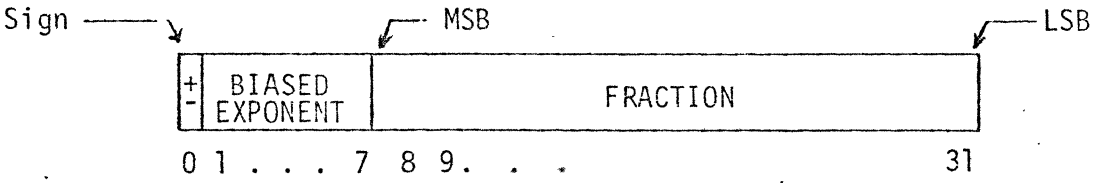
2's complement representation for negative numbers.

2. Fixed point, half length, 16-bit word (two half length words shown).



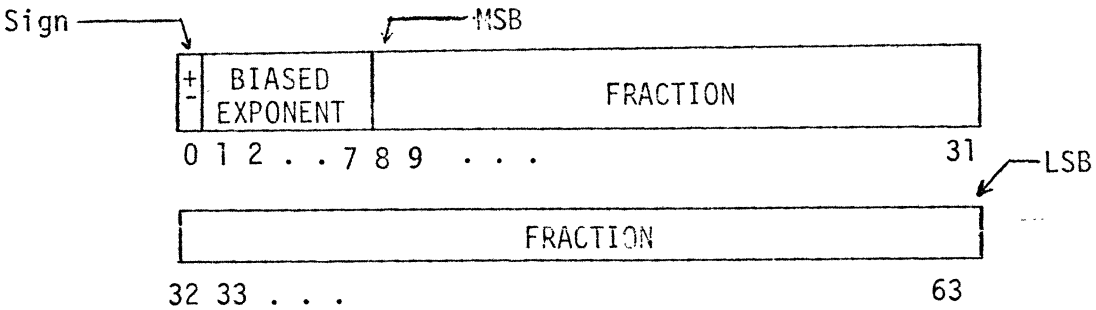
2's complement representation for negative numbers.

3. Floating point, single length, 32-bit word.



sign and magnitude representation for fractional portion.

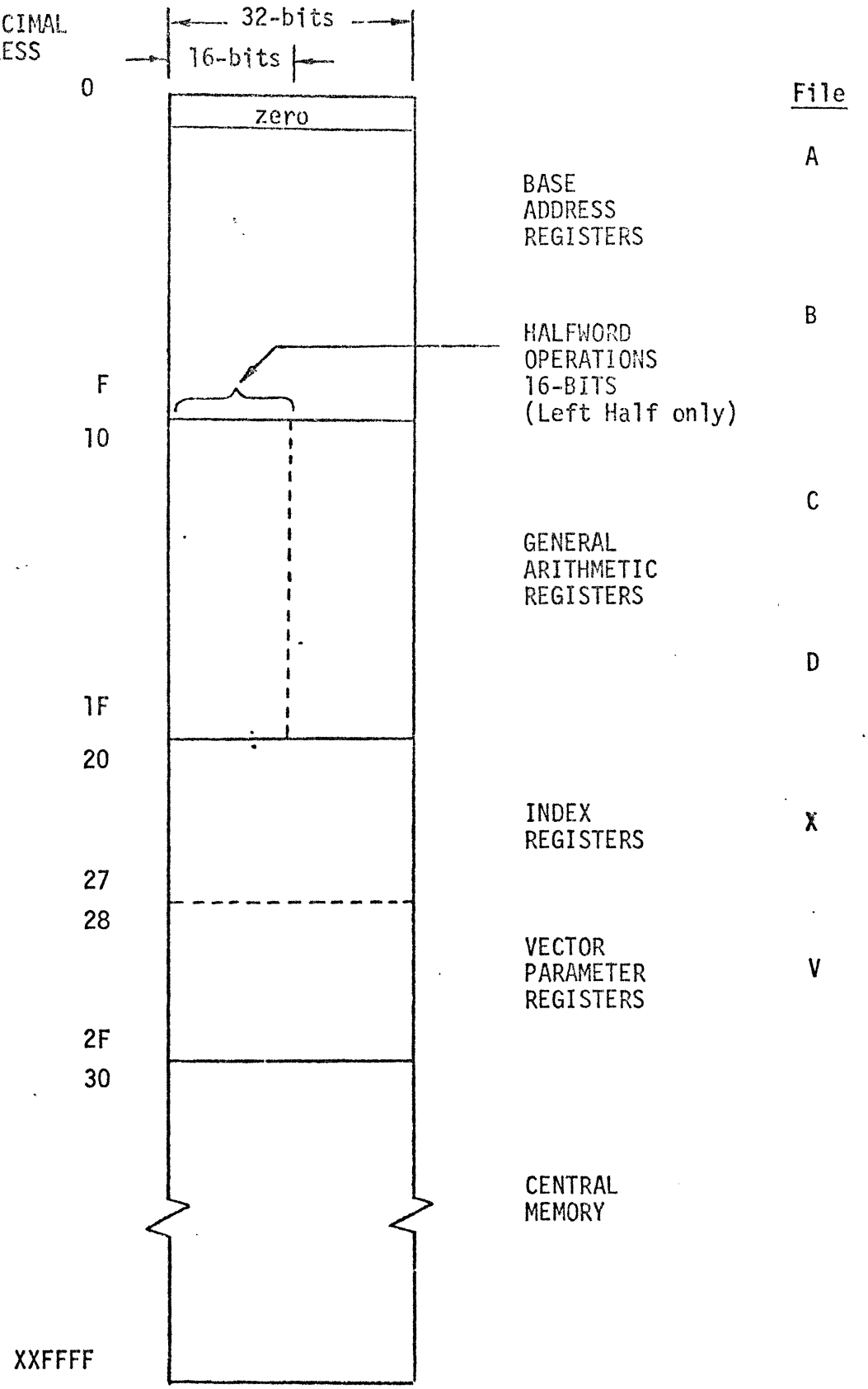
4. Floating point, double length, 64-bit word.



sign and magnitude representation for fractional portion.

Figure 1. Data Formats

REGISTER FILES



DATA FORMS

INFINITE FORMS AND INDEFINITE FORMS:

FLOATING ADD	A.U. OUTPUT	FLOATING POINT OVERFLOW
(+ ∞) + (+ ∞)	+ ∞	Yes
(+ ∞) + (- ∞)	IND	Yes
(- ∞) + (+ ∞)	IND	Yes
(- ∞) + (- ∞)	- ∞	Yes
(+ ∞) + (± N)	+ ∞	Yes
(- ∞) + (± N)	- ∞	Yes
(IND) + (± N)	IND	Yes
(IND) + (± ∞)	IND	Yes

FLOATING POINT SINGLE LENGTH FORMS ARE:

+ ∞	7FFF	FFFF	Positive infinite form.
- ∞	FFFF	FFFF	Negative infinite form.
IND	7F00	0000	Indefinite form.

FLOATING POINT DOUBLE LENGTH FORMS ARE:

+ ∞	7FFF	FFFF	FFFF	FFFF
- ∞	FFFF	FFFF	FFFF	FFFF
IND	7F00	0000	0000	0000

The indefinite form, 7F00 ···00, is generated by the Arithmetic Unit when an indefinite form or a "dirty zero" appears at either input to the Arithmetic Unit during a floating point arithmetic operation.

A "dirty zero" is a floating point form consisting of a zero mantissa and a non-zero exponent. It has the form XX00···00, where at least one X is not equal to zero.

FLOATING ADD MAGNITUDE	A. U. OUTPUT	FLOATING POINT OVERFLOW
$(+\infty) + \lvert(\pm\infty)\rvert$	$+\infty$	Yes
$(-\infty) + \lvert(\pm\infty)\rvert$	IND	Yes
$(+\infty) + \lvert(\pm N)\rvert$	$+\infty$	Yes
$(-\infty) + \lvert(\pm N)\rvert$	$-\infty$	Yes
$(\pm N) + \lvert(\pm\infty)\rvert$	$+\infty$	Yes
(IND) + $\lvert(\pm N)\rvert$	IND	Yes
(IND) + $\lvert(\pm\infty)\rvert$	IND	Yes
$(\pm N) + \lvert$(IND)$\rvert$	IND	Yes
$(\pm\infty) + \lvert$(IND)$\rvert$	IND	Yes

FLOATING SUBTRACT	A. U. OUTPUT	FLOATING POINT OVERFLOW
$(+\infty) - (+\infty)$	IND	Yes
$(+\infty) - (-\infty)$	$+\infty$	Yes
$(-\infty) - (+\infty)$	$-\infty$	Yes
$(-\infty) - (-\infty)$	IND	Yes
$(+\infty) - (\pm N)$	$+\infty$	Yes
$(-\infty) - (\pm N)$	$-\infty$	Yes
$(\pm N) - (+\infty)$	$-\infty$	Yes
$(\pm N) - (-\infty)$	$+\infty$	Yes
(IND) - $(\pm N)$	IND	Yes
(IND) - $(\pm\infty)$	IND	Yes
$(\pm N)$ - (IND)	IND	Yes
$(\pm\infty)$ - (IND)	IND	Yes

FLOATING SUBTRACT MAGNITUDE	A. U. OUTPUT	FLOATING POINT OVERFLOW
$(+\infty) - \lvert(\pm\infty)\rvert$	IND	Yes
$(-\infty) - \lvert(\pm\infty)\rvert$	$-\infty$	Yes
$(+\infty) - \lvert(\pm N)\rvert$	$+\infty$	Yes
$(-\infty) - \lvert(\pm N)\rvert$	$-\infty$	Yes
$(\pm N) - \lvert(\pm\infty)\rvert$	$-\infty$	Yes
$(IND) - \lvert(\pm N)\rvert$	IND	Yes
$(IND) - \lvert(\pm\infty)\rvert$	IND	Yes
$(\pm N) - \lvert(IND)\rvert$	IND	Yes
$(\pm\infty) - \lvert(IND)\rvert$	IND	Yes

FLOATING MULTIPLY OR FLOATING VECTOR DOT PRODUCT	A. U. OUTPUT	FLOATING POINT OVERFLOW
$(+\infty) \cdot (+\infty)$	$+\infty$	Yes
$(+\infty) \cdot (-\infty)$	$-\infty$	Yes
$(-\infty) \cdot (+\infty)$	$-\infty$	Yes
$(-\infty) \cdot (-\infty)$	$+\infty$	Yes
$(+\infty) \cdot (\pm N)$	$\pm\infty$	Yes
$(-\infty) \cdot (\pm N)$	$\mp\infty$	Yes
$(\pm\infty) \cdot (0)$	IND	Yes
$(\pm N) \cdot (0)$	0	No
$(0) \cdot (0)$	0	No
$(IND) \cdot (\pm\infty)$	IND	Yes
$(IND) \cdot (\pm N)$	IND	Yes
$(IND) \cdot (0)$	IND	Yes

FLOATING DIVIDE	ALU OUTPUT	FLOATING POINT OVERFLOW	DIVIDE CHECK
(± ∞) ÷ (± ∞)	IND	Yes	No
(+ ∞) ÷ (N)	+ ∞	Yes	No
(+ ∞) ÷ (-N)	- ∞	Yes	No
(- ∞) ÷ (N)	- ∞	Yes	No
(- ∞) ÷ (-N)	+ ∞	Yes	No
(± ∞) ÷ (0)	± ∞	Yes	Yes
(± N) ÷ (± ∞)	0	No	No
(0) ÷ (± ∞)	0	No	No
(0) ÷ (± N)	0	No	No
(0) ÷ (0)	IND	Yes	Yes
(N) ÷ (0)	+ ∞	Yes	Yes
(-N) ÷ (0)	- ∞	Yes	Yes
(IND) ÷ (± ∞)	IND	Yes	No
(IND) ÷ (±N)	IND	Yes	No
(IND) ÷ (0)	IND	Yes	Yes
(± ∞) ÷ (IND)	IND	Yes	No
(± N) ÷ (IND)	IND	Yes	No
(0) ÷ (IND)	IND	Yes	No

PROGRAM STATUS DOUBLEWORD

Control conditions within the CP which are critical to a CP program are:

1. Program counter or instruction address.
2. Arithmetic exception mask.
3. Arithmetic exception code.
4. Condition code.
5. Memory protection controls and memory map controls.

PROGRAM STATUS LOCATIONS

The CP status information exists in the CP as a collection of separate registers and flip-flops. The designations assigned to the control registers and flip-flops along with their function are as follows:

PC - Program Counter. A 24-bit counter which contains the current instruction address at the index addition level of the Instruction Processing Unit.

AE - Arithmetic Exception Mask. Consists of flip-flops designated MD, MF, MØ, and MU. When the AE MASK flip-flops are "one" (masked ON) a signal from the CP to the PP is activated upon detection of a maskable condition within the CP for which a PP interrupt is desired. The maskable conditions and their respective mask flip-flops (ff) are:

AE Mask ff	Maskable Conditions
MD	Divide check - Divisor is equal to zero (Fixed point).
MF	Fixed point overflow.
MØ	Floating point exponent overflow.
MU	Floating point exponent underflow.

When one of the AE MASK flip-flops (MD, MF, MØ, and MU) is "zero" (masked OFF), that condition corresponding to the zero flip-flop will not activate the interrupt signal from the CP to the PP. The interrupt signal is either inhibited or allowed to occur depending on the setting of the AE MASK bits (1 ~ masked ON, 0 ~ masked OFF). For example, if MD = 1, MF = 0, MØ = 0, and MU = 0, the only maskable arithmetic exception condition which can cause an interrupt signal from the CP to the PP is a divide check.

The AE MASK bits can be changed by a Load Arithmetic Mask (LAM) instruction executed by the CP.

The PP interrupt conditions within the CP which are not maskable include an undefined operation code, CM protection bounds, CM parity error on read, and Breakpoint, and Specification error.

AE, COND - Arithmetic Exception Condition. An arithmetic exception indicator bit (D, F, Ø, or U) is set to "one" whenever one of the following maskable arithmetic exception conditions is detected. Table 1 presents the arithmetic exception condition for each scalar instruction.

D - Divide Check - A divide check interrupt condition is recognized when the divisor is zero in fixed or floating point operations. The interrupt is enabled if a one has been loaded into the MD bit of the AE mask.

F - Fixed Point Overflow - When a high-order carry occurs or high-order significant bits are lost in fixed-point add, subtract or left arithmetic shift operations, etc., a fixed point overflow condition is recognized. The operations is completed by ignoring the information placed outside the register but a one will be placed in the X bit of the AE condition register. The interruption is enabled if a one has been placed in the MX bit of the AE mask register.

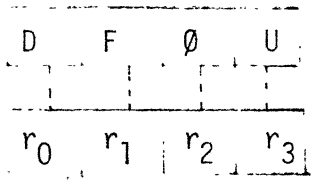
Table 1. Arithmetic Exception Condit Table for Scalar Instructions

Divide Check (D)	Fixed Point Overflow (F)	Floating Point Exponent Overflow (Ø)	Floating Point Exponent Underflow (U)	Unassigned Operation Code	Arithmetic Exception Condition Not Possible	
D	LM,LMH	AF,AFD,	AF,AFD,	10,11	L,LI,LH,LHI	SL,SLH,SLD,
DI	LN,LNH,SN,SNH	AMF,AMFD,	AMF,AMFD,	26	LR,LL,LD,	SC,SCH,SCD,
DH	A,AI,AH,AIH,	SF,SFD,	SF,SFD,	53,57,5A,5B	LM,LMH,LMF,LMD,	RVS,
DIH	AM,AMH	SMF,SMFD,	SMF,SMFD,	5D,5E,5F,	LN,LNH,LNF,LND,	C,CI,CH,CIH,
DF	S,SI,SH,SIH,	MF,MFD,	MF,MFD	61,63,69,6B,	LNM,LNMH,LNMF,LNMD,	CF,CFD
DFD	SM,SMH,	DF,DFD,	DF,DFD,	71,73,76,77,79,	LF,LFM,XCH,	CAND,CANDI,CANDD
	M(arith,R-odd)	FXFL,FXFD,		7B,7E,7F,	LAM,LLA,LØ,	COR,CORI,CORD,
	M(base,index)	FHFL,FHFD,		9A,9B,9E,	ST,STH,STR,STL,	IBZ,IBNZ,DBZ,DBNZ,
	MI(arith,R-odd)			A3,A4,A5,A6,	SPS,STD,	ISE,ISNE,DSE,DSNE,
	MI(base,index)			A7,AE,AF,	STZ,STZH,STZD,	BCLE,BCG,
	D(R-even)			B1,B2,B3,B4,	STF,STFM,	BC,BL,
	DI(R-even)			B5,B6,B7,B8,	M(arith,R-even),	BR,BRL,
	DH,DIH,			B9,BA,BB,BC,	MI(arith,R-even),	BAE,BXEC,
	SA,SAH,SAD,			BD,BE,BF,	MH,MIH,	PSH,PUL,MOD,
	FLFX,FLFH,FDFX			CZ,DO,D1,D2,	D(R-odd)	BLB,BLX,
				D3,D4,D5,D6,	DI(R-odd)	LEA,INT,XEC,
				D7,DA,DB,DC,	AND,ANDI,ANDD,	NFX,NFH,
				DD,DF,	OR,ORI,ORD,	MCP,MCW,
				EA,EB,EE,EF,	XOR,XORI,XORD,	
				F1,F3,F5,F7,	EQC,EQCI,EQCD,	
				F9,FA,FB,		
				FD,FE,FF,		

Ø - Floating Point Exponent Overflow - When the result characteristic exceeds 127 in floating-point addition, subtraction, multiplication, or division, an exponent overflow is recognized. The AE condition register is set with a one in bit Ø. The interrupt will occur if a one has been placed in the MØ bit of the arithmetic exception mask register. The result will be set to +∞ for positive overflow and -∞ for negative overflow.

U - Floating Point Exponent Underflow - When the characteristic is less than zero in floating-point addition, subtraction, multiplication, or division, an exponent underflow is recognized. The operation is completed by making the result a true zero. A one is set in bit U of the AE condition register. Interrupt will occur if a one has been placed in the MU bit of the arithmetic exception mask register. When the result of a floating-point addition or subtraction has an all zero fraction, the operation is completed by making the result a true zero.

The bits (D, F, Ø, U) so set will remain set until interrogated by a Test Arithmetic Exception Code and Branch (AE) instruction. The R-field of an AE instruction is "ANDed" with the arithmetic exception register and if any of the resulting four bits are "one", then the branch will be taken. Only the AE register bits corresponding to "ones" in the R-field are reset to zero during execution of an AE--instruction.

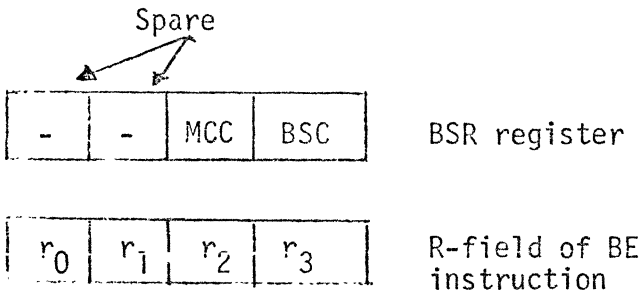


The Test Arithmetic Exception Code and Branch instruction allows a program to sense a divide check or overflow condition (without CP interrupt) and perform corrective action if necessary. A CP to PP Interrupt signal will not occur for the maskable arithmetic conditions if the corresponding MASK bits (MD, MX, MØ, and MU) are zero.

Unassigned Operation Code - If an unassigned operation code is encountered the operation is not executed and the interrupt signal is sent from the CP to the PPU. Also, if an indirect addressing chain detects an indirect address for which the four most significant bits are not all zeros, then the indirect addressing instruction is not executed and the unassigned operation code interrupt signal is sent from the CP to the PPU.

BSC - Execute instruction branch or skip condition. This is a four bit register (BSR) of which only the two least significant bits are used as indicators.

The BSC-bit of the BSR register is set to "one" whenever an Execute Instruction (XEC) executes a branch or skip type instruction and the condition for branching or skipping is satisfied. The BSC-bit is reset to "zero" whenever the condition for branching or skipping is not satisfied. No branch will occur when the BSC-bit in the BSR indicator is set. Instead, the instruction following the XEC instruction is executed.



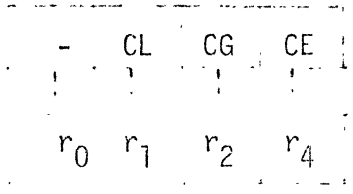
A Branch on Execute Condition (BEC) instruction can determine whether the condition for branching or skipping was satisfied for the case of an Execute instruction executing any conditional branch or skip type instruction. If a BEC instruction (one for which R = 0001) branches, then the condition for branching was satisfied.

The MCC-bit of the BSR register is set to "one" whenever an Execute instruction executes a Monitor Call and Proceed (MCP) or a Monitor Call and Wait (MCW). The monitor call does not write into central memory nor is the PPU signaled of a monitor call when an MCP or MCW is executed by an Execute instruction.

If a Branch on the Execute Condition instruction (one for which R=0010) branches, then an Execute instruction has executed an MCP or MCW instruction.

The indicator bits of the BSR register which correspond to the position of "ones" in the R-field of the BEC instruction are reset to "zero" by the BEC instruction. Bit positions of BSR which are not tested by "ones" in R are not reset by the BEC instruction. Only the two LSB's of the BSR register are used by the Branch on Execute Condition instruction. The remaining two unused bits of BSR will be tied to "zero". The 2 MSB's of the R-field of the BEC instruction are "don't cares" as a result of the 2 MSB's of BSR being forced to "zero".

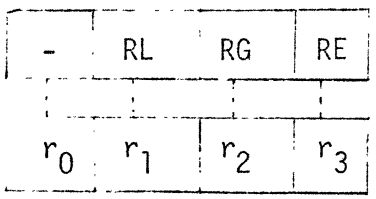
CC - Compare Code. One of the flip-flops CL, CG, or CE is set by an arithmetic or logical compare instruction and the bit so set will remain set until another compare instruction modifies the setting. Thus, the compare code indicators always reflect the outcome of the last compare instruction executed before being tested by a Comparison Code Branch instruction. The compare code indicators are not affected by a Comparison Code Branch instruction.



The arithmetic and logical properties with their corresponding comparison code are listed below for the general case of one operand (x) compared with another operand (y). The definitions of x and y are given with the instructions defining the various comparison operations.

Arithmetic compare (x) : (y)	CC, compare code CL	CG	CE	Logical compare $[(x)_j$ Boolean $(y)_j]$
(x) < (y)	1	0	0	Mixed "1's" and "0's"
(x) > (y)	0	1	0	All bits are "1"
(x) = (y)	0	0	1	All bits are "0"

RC - Result Code - One of the flip-flops RL, RG, or RE is set according to the properties of the arithmetic or logical result emerging from the arithmetic unit and the bit so set will remain set until another result from the AU modifies the setting. Thus, the result code indicators reflect the current status of the most recently referenced register. The result code indicators are not affected by a Result Code Branch instruction.



The arithmetic and logical properties of the AU result with their corresponding result code are listed below.

Arithmetic result from AU			RC, Result code RL	RG	RE	Logical result from AU
(x)	<	0	1	0	0	Mixed "1's" and "0's"
(x)	>	0	0	1	0	All bits are "1"
(x)	=	0	0	0	1	All bits are "0"

NOTE: Load and Store instructions cause the result code to be set as if the operand were an arithmetic result from the AU.

CP MEMORY USAGE BITS

Bits 16 through 19 (labeled CP MEM USAGE) of the second word of the Program Status Doubleword are reserved for Central Processor memory usage information. The CP MEM USAGE bits indicate that the Central Processor has been placed under the following usage mode by the Peripheral Processing Unit.

bit 16, M	0 ~ Mapped 1 ~ Not Mapped
bit 17, P	0 ~ Memory Protection 1 ~ No Memory Protection
bit 18, B	0 ~ No Breakpoint 1 ~ Breakpoint Active
bit 19, V	0 ~ Actual Breakpoint 1 ~ Virtual Breakpoint

STORE AND LOAD PROGRAM STATUS

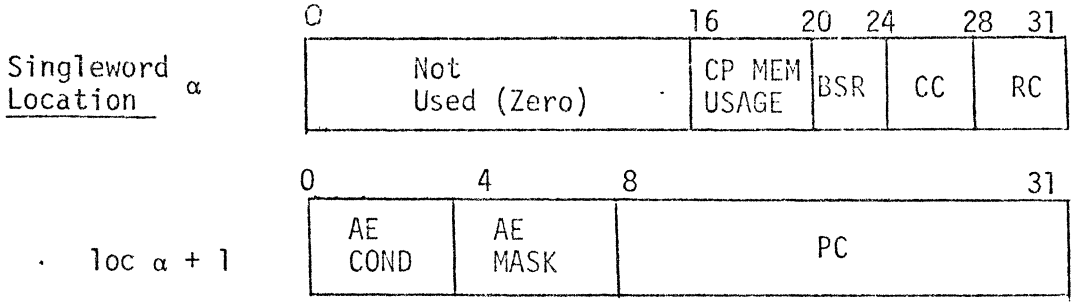
Three CCR signals from the PPU to the CP store, load, or exchange the program status doubleword as follows.

Store program status doubleword - The CP permits all instructions which are currently in process to go to compleiton. No new instructions are fetched by the instruction fetch unit after receiving this signal. After all instructions have been completed, the program status doubleword and all register files are stored at the location specified by the contents of memory location 14.

Load program status doubleword - The CP immeidately loads the program status doubleword and all register files beginning at the location specified by the contents of memory location 15 and then proceeds with execution.

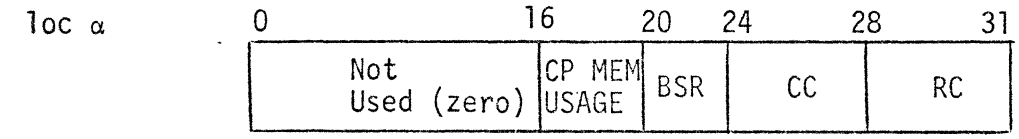
Exchange program status - The CP first performs the Store operation, then it performs the Load operation.

The program status doubleword has the following format:

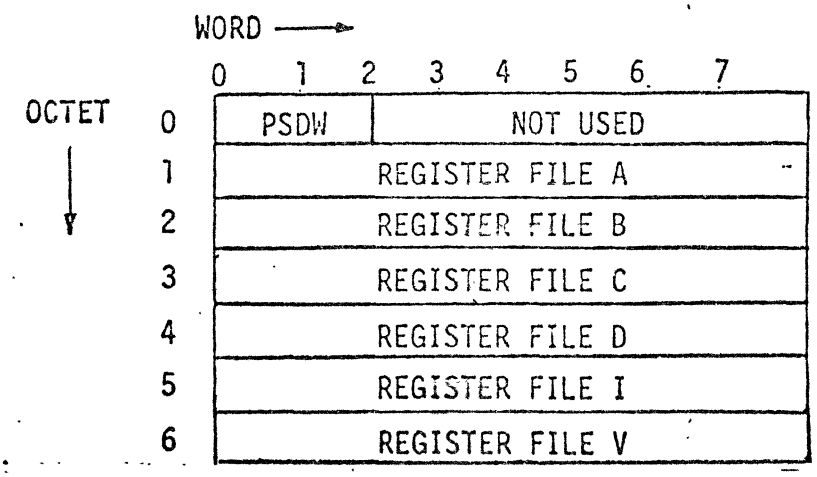


It should be clear from the preceding descriptions that the CP status information exists as a collection of separate registers and flip-flops and that the fields designated here only indicate how the respective CP status information is formatted in central memory. The designation names of these registers have been retained in this memory format.

A separate CP instruction is provided to store the first word of the program status doubleword into central memory singleword location α. The Store Program Status Word, SPS, instruction stores the CP MEM USAGE, BSR, CC and RC status information according to the format.



The load, store, or exchange status memory map appears as follows:



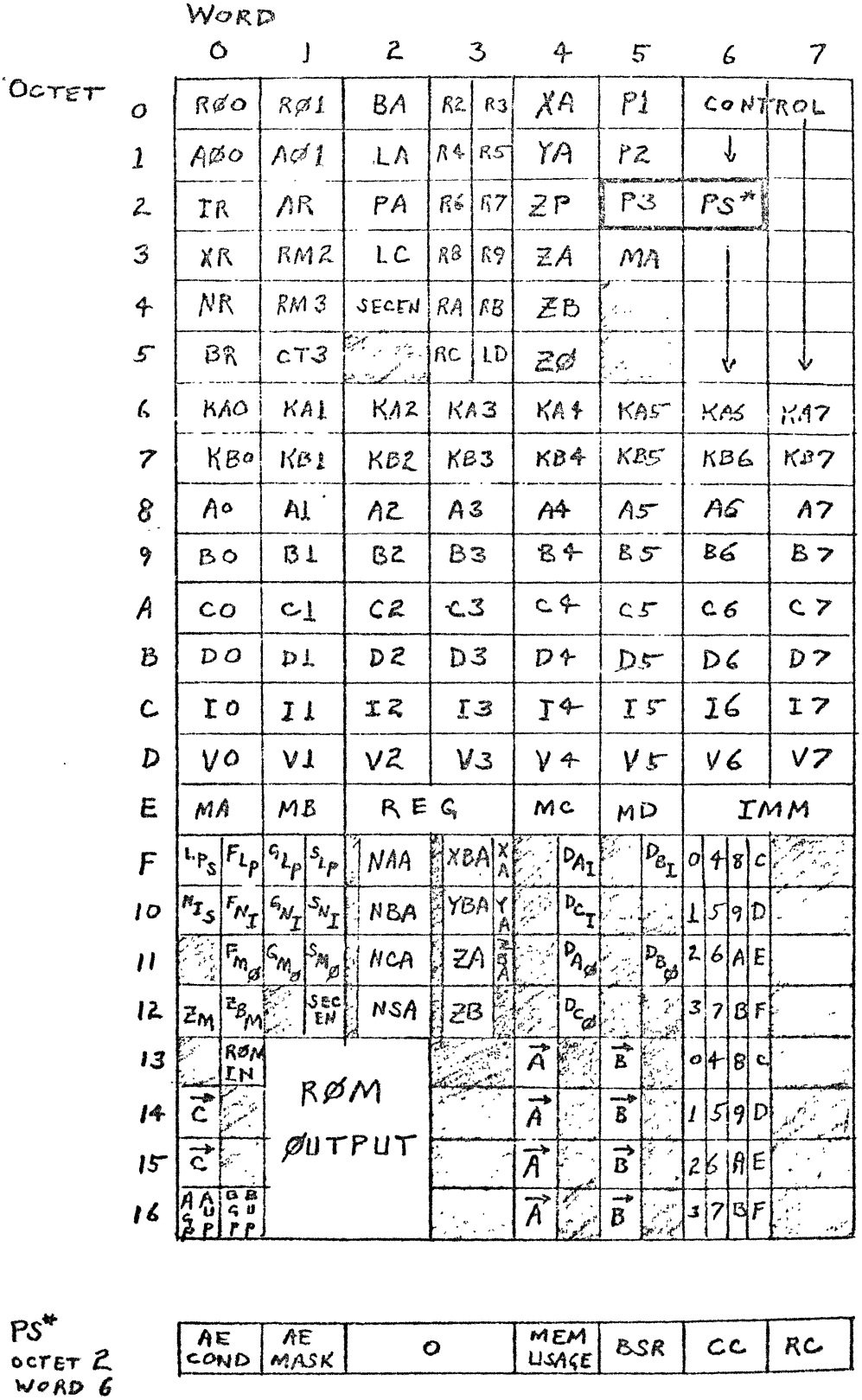


FIGURE 1. INTERMEDIATE LEVEL CP MAP

When it is intended that the CP start a new assignment through the use of Loading or Exchanging at the Intermediate level, and there is no prior CP intermediate level map available to load the internal registers of the CP, then the starting address of the instructions to be executed by the CP must be entered into the P3 word of an intermediate map. This is octet 2, word 5 (numbers begin with 0) of the intermediate map. The AE Condition, AE Mask, CP Memory Usage, BSR, CC, and RC bits must also be loaded. They are in octet 2, word 6 of the intermediate map. All other words of the map should be zero, initially. Entering of data into these locations must, of course, be done prior to an Exchange Intermediate (either by CCR code or automatic context switching) or a Load Intermediate (by CCR code). See Figure 1 for Intermediate Level CR Map page 22B

ASSEMBLER MNEMONICS

The Central Processor (CP) instructions are described using their assembler mnemonics for each instruction. Instructions which have the same mnemonics but different operation codes are distinguished from each other by the register designation. For example, the assembler will recognize L as the mnemonic for an instruction which loads an index register even though the same mnemonic is also used to load base registers and arithmetic registers. All three instructions have different operation codes and the one selected can be determined from the register designation. Thus, the volume of instruction mnemonics to learn is reduced.

The special characters @ and = are used in certain ASC instruction procedures. Whenever both of these symbols are used with the same expression, the order of their appearance is irrelevant. However, their appearance will be flagged in error by the assembler if they are used incorrectly. Their use anywhere but immediately preceding an expression is illegal.

The CP instructions are written in the form:

Label	Command	Operand	Remarks

In all instructions, Label is optional. When used, it will be assigned the value of the location counter of that instruction.

The operand format is dictated by the command and is described for each instruction on the following pages. In general, the following characters are used to represent special information in the operand formats:

R is the name of the register involved in the operation.

N is the central memory reference in the operand and may be represented by using a symbol which is the label of a memory location or by using base and displacement if N is replaced by (D,B) where,

D is the displacement value and,

B is the base register.

X is the index register name and is optional. It is used to modify the N field.

Examples: R, N, X or R, (D,B), X

When the N field is used for immediate referencing (to the instruction itself), the symbol I will be used.

"@" means an "@" can be used to indicate indirect addressing.

= means an "=" can be used to cause the Assembler to create a literal to be generated from N and replace N in this instruction with the literal's location.

For Branch instructions, "=" may be used only in pair with a "@". Using the "@" does not require the "=".

CP Operand Format Types:

1. R,@=N,X
2. R,@N,X
3. @N,X
4. R,I,X
5. I,X
6. R,R,N where the first R is for the R-field, the second for the X-field.
7. R,@=N,X
8. @=N,X
9. M,@=N,X
10. I

All values of registers must be represented by using symbolic constants which are familiar to the assembler via automatic initialization. For example, the CP registers are represented as

```
Location   0 = B0
           1 = B1
           2 = B2
           3 = B3
             .
             .
           E = B14
           F = B15
          10 = A0
          11 = A1
             .
             .
          1F = A15
          20 = X0
          21 = X1
             .
             .
          27 = X7
          28 = V0
          29 = V1
             .
             .
          2F = V7
```

If the register is not represented symbolically (i.e., X7 is specified by using a 27), an error is indicated by the Assembler unless it is in the second list item of the operand field (the Address Parameter). In this parameter, a number will be assumed to be an absolute address. The above reserved symbols may not be assigned as labels by the user.

In the following lists, for scalar Central Processor instructions, the columns contain the following information: MNEMONIC CODE heads the column listing the mnemonics recognized and interpreted in the command field of the Assembler Language statement; INSTRUCTION heads the column containing a brief description of the operation initiated by the command; OPERATION CODE heads the column listing the machine (CP) code produced by the Assembler from information in the command and operand fields of the Assembler Language statement; TYPE FORMAT heads the column which gives the format type showing the maximum complexity permitted for the particular command; OPERAND FORMAT heads the column which contains the symbolic representation of an operand of maximum complexity for the particular command; and ASSEMBLER SUPPLIED R FIELD, for those commands where it is applicable, heads the column which contains the value the Assembler supplies to the R field of the resultant machine code.

MNEMONIC CODE	INSTRUCTION	OPERATION CODE	TYPE FORMAT	OPERAND FORMAT
ST	Store arithmetic register, single length	24	2	R,@N,X
ST	Store base register, single length	28	2	R,@N,X
ST	Store index register or vector parameter register, single length	2C	2	R,@N,X
STH	Store half length, arithmetic register	25	2	R,@N,X
STR	Store register right half into memory right half, arithmetic register	2D	2	R,@N,X
STL	Store register left half into memory right half, arithmetic register	29	2	R,@N,X
SPS	Store program status word	22	3	@N,X
STD	Store arithmetic register, double length	27	2	R,@N,X
STN	Store negative, single length	34	7	R,@=N,X
STNH	Store negative, half length	35	7	R,@=N,X
STNF	Store negative, floating point	36	7	R,@=N,X
STND	Store negative, double length	37	7	R,@=N,X
STO	Store ones complement	2E	7	R,@=N,X
STOH	Store ones complement, half length	2A	7	R,@=N,X

MNEMONIC CODE	INSTRUCTION	OPERATION CODE	TYPE FORMAT	OPERAND FORMAT
STZ	Store zero, single length	20	3	@N,X
STZH	Store zero, half length	21	3	@N,X
STZD	Store zero, double length	23	3	@N,X
STF	Store base register file, registers $1\text{-}7_H$, M=0	2B	9	M,@N,X
STF	Store base register file, registers $8\text{-}F_H$, M=1	2B	9	M,@N,X
STF	Store arithmetic register file, registers $10\text{-}17_H$, M=2	2B	9	M,@N,X
STF	Store arithmetic register file, registers $18\text{-}1F_H$, M=3	2B	9	M,@N,X
STF	Store index register file, registers $20\text{-}27_H$, M=4	2B	9	M,@N,X
STF	Store vector parameter register file, registers $28\text{-}2F_H$, M=5	2B	9	M,@N,X
STFM	Store all register files, registers $1\text{-}2F_H$	2F	3	@N,X
L	Load arithmetic register single length word	14	1	R,@=N,X
L	Load base register single length	18	1	R,@=N,X
L	Load index register or vector parameter register single length	1C	1	R,@=N,X
LI	Load immediate into arithmetic register single length	54	4	R,I,X
LI	Load immediate into index register, or vector parameter register single length	5C	4	R,I,X
LH	Load arithmetic register half length word	15	1	R,@=N,X
LIH	Load immediate into arithmetic register half length	55	4	R,I,X
LR	Load memory right halfword into arithmetic register right halfword	1D	1	R,@=N,X
LL	Load memory right halfword into arithmetic register left halfword	19	1	R,@=N,X
LD	Load arithmetic register double length word	17	1	R,@=N,X

MNEMONIC CODE	INSTRUCTION	OPERATION CODE	TYPE FORMAT	OPERAND FORMAT
LM	Load magnitude fixed point single length - arithmetic register	3C	1	R,@=N,X
LMH	Load magnitude fixed point half length - arithmetic register	3D	1	R,@=N,X
LMF	Load magnitude floating point single length - arithmetic register	3E	1	R,@=N,X
LMD	Load magnitude floating point double length - arithmetic register	3F	1	R,@=N,X
LN	Load negative fixed point single length (load twos complement) arithmetic register	30	1	R,@=N,X
LNH	Load negative fixed point half length - arithmetic register	31	1	R,@=N,X
LNF	Load negative floating point single length - arithmetic register	32	1	R,@=N,X
LND	Load negative floating point double length - arithmetic register	33	1	R,@=N,X
LNM	Load negative magnitude fixed point single length - arithmetic register	38	1	R,@=N,X
LNMH	Load negative magnitude fixed point half length - arithmetic register	39	1	R,@=N,X
LNMF	Load negative magnitude floating point single length - arithmetic register	3A	1	R,@=N,X
LNMD	Load negative magnitude floating point double length - arithmetic register	3B	1	R,@=N,X
LF	Load base register file, registers $1\text{-}7_H$, M=0	1B	9	M,@N,X
LF	Load base register file, registers $8\text{-}F_H$, M=1	1B	9	M,@N,X
LF	Load arithmetic register file, registers $10\text{-}17_H$, M=2	1B	9	M,@N,X
LF	Load arithmetic register file, registers $18\text{-}1F_H$, M=3	1B	9	M,@N,X

MNEMONIC CODE	INSTRUCTION	OPERATION CODE	TYPE FORMAT	OPERAND FORMAT
LF	Load index register file, registers 20-27_H, M=4	1B	9	M,@N,X
LF	Load vector parameter register file, registers 28-$2F_H$, M=5	1B	9	M,@N,X
LFM	Load all register files	1F	3	@N,X
XCH	Exchange - arithmetic register	1A	2	R,@N,X
LAM	Load arithmetic mask	12	8	@=N,X
LAC	Load arithmetic exception condition	13	8	@=N,X
LLA	Load look ahead	16	10	I
LO	Load arithmetic register with ones complement, single length	1E	1	R,@=N,X
A	Add to arithmetic register, fixed point, single length	40	1	R,@=N,X
A	Add to base register, fixed point, single length	60	1	R,@=N,X
A	Add to index or vector parameter register, fixed point, single length	62	1	R,@=N,X
AI	Add immediate to arithmetic register, fixed point, single length	50	4	R,I,X
AI	Add immediate to base register, fixed point, single length	70	4	R,I,X
AI	Add immediate to index or vector parameter register, fixed point, single length	72	4	R,I,X
AH	Add fixed point, half length - arithmetic register	41	1	R,@=N,X
AIH	Add immediate fixed point, half length - arithmetic register	51	4	R,I,X
AF	Add floating point, single length - arithmetic register	42	1	R,@=N,X
AFD	Add floating point, double length - arithmetic register	43	1	R,@=N,X
AM	Add magnitude fixed point, single length - arithmetic register	44	1	R,@=N,X
AMH	Add magnitude fixed point, half length - arithmetic register	45	1	R,@=N,X

MNEMONIC CODE	INSTRUCTION	OPERATION CODE	TYPE FORMAT	OPERAND FORMAT
AMF	Add magnitude floating point, single length - arithmetic register	46	1	R,@=N,X
AMFD	Add magnitude floating point, double length - arithmetic register	47	1	R,@=N,X
S	Subtract fixed point, single length - arithmetic register	48	1	R,@=N,X
SI	Subtract immediate fixed point, single length - arithmetic register	58	4	R,I,X
SH	Subtract fixed point, half length - arithmetic register	49	1	R,@=N,X
SIH	Subtract immediate fixed point, half length - arithmetic register	59	4	R,I,X
SF	Subtract floating point, single length - arithmetic register	4A	1	R,@=N,X
SFD	Subtract floating point, double length - arithmetic register	4B	1	R,@=N,X
SM	Subtract magnitude fixed point, single length - arithmetic register	4C	1	R,@=N,X
SMH	Subtract magnitude fixed point, half length - arithmetic register	4D	1	R,@=N,X
SMF	Subtract magnitude floating point, single length - arithmetic register	4E	1	R,@=N,X
SMFD	Subtract magnitude floating point, double length - arithmetic register	4F	1	R,@=N,X
M	Multiply fixed point, single length - arithmetic register	6C	1	R,@=N,X
M	Multiply base register	68	1	R,@=N,X
M	Multiply index or vector parameter register	6A	1	R,@=N,X
MI	Multiply immediate fixed point, single length - arithmetic register	7C	4	R,I,X
MI	Multiply immediate to base register	78	4	R,I,X
MI	Multiply immediate to index or vector parameter register	7A	4	R,I,X
MH	Multiply fixed point, half length - arithmetic register	6D	1	R,@=N,X

MNEMONIC CODE	INSTRUCTION	OPERATION CODE	TYPE FORMAT	OPERAND FORMAT
MIH	Multiply immediate fixed point, half length - arithmetic register	7D	4	R,I,X
MF	Multiply floating point, single length - arithmetic register	6E	1	R,@=N,X
MFD	Multiply floating point, double length - arithmetic register	6F	1	R,@=N,X
D	Divide fixed point, single length - arithmetic register	64	1	R,@=N,X
DI	Divide immediate fixed point, single length - arithmetic register	74	4	R,I,X
DH	Divide fixed point, half length - arithmetic register	65	1	R,@=N,X
DIH	Divide immediate fixed point, half length - arithmetic register	75	4	R,I,X
DF	Divide floating point, single length - arithmetic register	66	1	R,@=N,X
DFD	Divide floating point, double length - arithmetic register	67	1	R,@=N,X
AND	AND - arithmetic register	E0	1	R,@=N,X
ANDI	Immediate AND - arithmetic register	F0	4	R,I,X
OR	OR - arithmetic register	E4	1	R,@=N,X
ORI	Immediate OR - arithmetic register	F4	4	R,I,X
XOR	Exclusive OR - arithmetic register	E8	1	R,@=N,X
XORI	Immediate Exclusive OR - arithmetic register	F8	4	R,I,X
EQC	Equivalence - arithmetic register	EC	1	R,@=N,X
EQCI	Immediate equivalence - arithmetic register	FC	4	R,I,X
ANDD	AND - arithmetic register (double length)	E1	1	R,@=N,X
ORD	OR - arithmetic register (double length)	E5	1	R,@=N,X
XORD	Exclusive OR - arithmetic register (double length)	E9	1	R,@=N,X
EQCD	Equivalence - arithmetic register (double length)	ED	1	R,@=N,X
SA	Arithmetic shift, fixed point, single length - arithmetic register	C0	4	R,I,X

MNEMONIC CODE	INSTRUCTION	OPERATION CODE	TYPE FORMAT	OPERAND FORMAT
SAH	Arithmetic shift, fixed point, half length - arithmetic register	C1	4	R,I,X
SAD	Arithmetic shift, fixed point, double length - arithmetic register	C3	4	R,I,X
SL	Logical shift, single length - arithmetic register	C4	4	R,I,X
SLH	Logical shift, half length - arithmetic register	C5	4	R,I,X
SLD	Logical shift, double length - arithmetic register	C7	4	R,I,X
SC	Circular shift, single length - arithmetic register	CC	4	R,I,X
SCH	Circular shift, half length - arithmetic register	CD	4	R,I,X
SCD	Circular shift, double length - arithmetic register	CF	4	R,I,X
RVS	Bit reversal, single length - arithmetic register	C6	4	R,I,X
C	Compare fixed point, single length - arithmetic register	C8	1	R,@=N,X
C	Compare index register, single length	CE	1	R,@=N,X
CI	Compare immediate, fixed point, single length - arithmetic register	D8	4	R,I,X
CI	Compare immediate, index register, single length	DE	4	R,I,X
CH	Compare fixed point, half length - arithmetic register	C9	1	R,@=N,X
CIH	Compare immediate, fixed point, half length - arithmetic register	D9	4	R,I,X
CF	Compare floating point, single length - arithmetic register	CA	1	R,@=N,X
CFD	Compare floating point, double length - arithmetic register	CB	1	R,@=N,X
CAND	Compare logical AND - arithmetic register (single length)	E2	1	R,@=N,X

MNEMONIC CODE	INSTRUCTION	OPERATION CODE	TYPE FORMAT	OPERAND FORMAT
CANDI	Compare immediate logical AND - arithmetic register (single length)	F2	4	R,I,X
COR	Compare logical OR, single length - arithmetic register	E6	1	R,@=N,X
CORI	Compare immediate logical OR, single length - arithmetic register	F6	4	R,I,X
CANDD	Compare logical AND, double length - arithmetic register	E3	1	R,@=N,X
CORD	Compare logical OR, double length - arithmetic register	E7	1	R,@=N,X
IBZ	Increment, test and branch on zero - arithmetic register	88	7	R,@=N,X
IBZ	Increment, test index, and branch on zero	8C	7	R,@=N,X
IBNZ	Increment, test, and branch on non-zero - arithmetic register	89	7	R,@=N,X
IBNZ	Increment, test index, and branch on non-zero	8D	7	R,@=N,X
DBZ	Decrement, test, and branch on zero - arithmetic register	8A	7	R,@=N,X
DBZ	Decrement, test index, and branch on zero	8E	7	R,@=N,X
DBNZ	Decrement, test, and branch on non-zero - arithmetic register	8B	7	R,@=N,X
DBNZ	Decrement, test index, and branch on non-zero	8F	7	R,@=N,X
ISE	Increment, test, and skip on equal - arithmetic register	80	1	R,@=N,X
ISNE	Increment, test, and skip on not equal - arithmetic register	81	1	R,@=N,X
DSE	Decrement, test, and skip on equal - arithmetic register	82	1	R,@=N,X
DSNE	Decrement, test, and skip on not equal - arithmetic register	83	1	R,@=N,X
BCLE	Branch on arithmetic register less than or equal to	84	6	R,R,N

MNEMONIC CODE	INSTRUCTION	OPERATION CODE	TYPE FORMAT	OPERAND FORMAT
BCLE	Branch on index less than or equal to	86	6	R,R,N
BCG	Branch on arithmetic register greater than	85	6	R,R,N
BCC	Branch on index greater than	87	6	R,R,N
PSH	Push word - arithmetic register	93	2	R,@N,X
PUL	Pull word - arithmetic register	97	2	R,@N,X
MOD	Modify - arithmetic register	9F	2	R,@N,X
BLB	Branch and load register with PC	98	7	R,@=N,X
BLX	Branch and load index register or vector parameter register	99	7	R,@=N,X
LEA	Load effective address - index register	56	1	R,@=N,X
LEA	Load effective address into base register	52	1	R,@=N,X
INT	Interpret - arithmetic register	92	1	R,@=N,X
XEC	Execute	96	8	@=N,X
FLFX	Convert floating point single length to fixed point single length - arithmetic register	A0	2	R,@N,X
FLFH	Convert floating point single length to fixed point half length - arithmetic register	A1	2	R,@N,X
FDFX	Convert floating point double length to fixed point single length	A2	2	R,@N,X
FXFL	Convert fixed point single length to floating point single length	A8	2	R,@N,X
FXFD	Convert fixed point single length to floating point double length	AA	2	R,@N,X
FHFL	Convert fixed point half length to floating point single length	A9	2	R,@N,X
FHFD	Convert fixed point half length to floating point double length	AB	2	R,@N,X
NFX	Normalize fixed point single length - arithmetic register	AC	2	R,@N,X
NFH	Normalize fixed point half length - arithmetic register	AD	2	R,@N,X

MNEMONIC CODE	INSTRUCTION	OPERATION CODE	ASSEMBLER SUPPLIES R FIELD	TYPE FORMAT	OPERAND FORMAT
MCP	Monitor call and proceed	90		5	I,X
MCW	Monitor call and wait	94		5	I,X
VECT	Vector	B0	R = 1	3	@N,X
VECTL	Vector after loading vector file	B0	R = 0	3	@N,X

Compare Code Branch Operation Code = 91

BCC	Branch on compare code	91		9	M,@=N,X
NOP	Take next instruction	91	R = 0	8	@=N,X

Comment: Execution of data values or indirect address constants will have the effect of a no-operation if the first four bits of the word (operation code) are zeros.

BE	(R) = (α)	91	R = 1	8	@=N,X
BG	(R) > (α)	91	R = 2	8	@=N,X
BGE	(R) $\geq$ (α)	91	R = 3	8	@=N,X
BL	(R) < (α)	91	R = 4	8	@=N,X
BLE	(R) $\leq$ (α)	91	R = 5	8	@=N,X
BNE	(R) $\neq$ (α)	91	R = 6	8	@=N,X
B	Unconditional branch		R = 7	8	@=N,X

Logical Branch Operation Code = 91

BCZ	All bits are zero	91	R = 1	8	@=N,X
BCO	All bits are one	91	R = 2	8	@=N,X
BCNM	Not mixed	91	R = 3	8	@=N,X
BCM	Mixed zeros and ones	91	R = 4	8	@=N,X
BCNO	Not all ones	91	R = 5	8	@=N,X
BCNZ	Not all zeros	91	R = 6	8	@=N,X

MNEMONIC CODE	INSTRUCTION	OPERATION CODE	ASSEMBLER SUPPLIED R FIELD	TYPE FORMAT	OPERAND FORMAT
Result Code Branch Operation Code = 95					
BRC	Branch on result code	95		9	M,@=N,X
BZ	(R) = 0	95	R = 1	8	@=N,X
BPL	(R) > 0	95	R = 2	8	@=N,X
BZP	(R) $\geq$ 0	95	R = 3	8	@=N,X
BMI	(R) < 0	95	R = 4	8	@=N,X
BZM	(R) $\leq$ 0	95	R = 5	8	@=N,X
BNZ	(R) $\neq$ 0	95	R = 6	8	@=N,X
Logical Result Branch Operation Code = 95					
BRZ	All bits are zero	95	R = 1	8	@=N,X
BRO	All bits are one	95	R = 2	8	@=N,X
BRNM	Not mixed	95	R = 3	8	@=N,X
BRM	Mixed zeros and ones	95	R = 4	8	@=N,X
BRNO	Not all ones	95	R = 5	8	@=N,X
BRNZ	Not all zeros	95	R = 6	8	@=N,X
Arithmetic Exception Branch Operation Code = 9D					
BAE	Branch on arithmetic exception	9D		9	M,@=N,X
BU	Floating point EXP underflow	9D	R = 1	8	@=N,X
BO	Floating point EXP overflow	9D	R = 2	8	@=N,X
BUO	Floating point EXP underflow or overflow	9D	R = 3	8	@=N,X
BX	Fixed point overflow	9D	R = 4	8	@=N,X
BXU	Fixed point overflow or floating EXP underflow	9D	R = 5	8	@=N,X

MNEMONIC CODE	INSTRUCTION	OPERATION CODE	ASSEMBLER SUPPLIED R FIELD	TYPE FORMAT	OPERAND FORMAT
Arithmetic Exception Branch Operation Code = 9D (continued)					
BXO	Fixed point overflow or floating point EXP overflow	9D	R = 6	8	@=N,X
BXUO	Fixed point overflow or floating point EXP overflow or underflow	9D	R = 7	8	@=N,X
BD	Divide check	9D	R = 8	8	@=N,X
BDU	Divide check or floating point EXP underflow	9D	R = 9	8	@=N,X
BDO	Divide check or floating point EXP overflow	9D	R = A	8	@=N,X
BDUO	Divide check or floating point EXP underflow or overflow	9D	R = B	8	@=N,X
BDX	Divide check or fixed point overflow	9D	R = C	8	@=N,X
BDXU	Divide check or fixed point overflow or floating point EXP underflow	9D	R = D	8	@=N,X
BDXO	Divide check or fixed point overflow or floating point EXP overflow	9D	R = E	8	@=N,X
BDXUO	Divide check or fixed point overflow or floating point EXP overflow or underflow	9D	R = F	8	@=N,X

MNEMONIC CODE	INSTRUCTION	OPERATION CODE	ASSEMBLER SUPPLIED R FIELD	TYPE FORMAT	OPERAND FORMAT
Branch on Execute Condition Operation Code = 9C					
BXEC	Branch on Execute branch condition true	9C	R = 1 or odd	3	@N,X

LOAD INSTRUCTIONS

LOAD WORD (L)

Operation Code	14, 18, 1C
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	α, word level
Symbolic Notation	$(\alpha) \rightarrow R$

There are three forms of the load word instruction indicated by the OP codes. One of these forms has two classes distinguished by the R field. In each case, the contents of the address indicated by the T, M, and N fields is loaded into the register indicated by the R field. In the case of OP code 1C, an R-field value from 0 to 7 (hexadecimal) indicates index registers (XR) and from 8 to F vector registers (VR).

OP Code	R-Field Destination	Register Loaded
14	AR (0 thru F)	Arithmetic
18	BR (1 thru F)	Base
1C	R Range 0 thru 7 addresses XR 0 thru 7.	Index
1C	R Range 8 thru F addresses VR 0 thru 7.	Vector

Programming Note: A Load Word instruction which specified base register zero (B0) will set the Result Code to the value of the α addressed operand, but otherwise appears as a no operation since base register zero is a fixed "all zeros" register.

Result Code: The result code register is set according to the arithmetic value of the operand in the register indicated by the R-field after the load operation is complete. The three allowable values of the result code are as follows:

Contents of R after load	Result Code (RL, RG, RE)
Negative, (R) < 0	(1, 0, 0)
Positive, (R) > 0	(0, 1, 0)
Zero, (R) = 0	(0, 0, 1)

Program Interruption: None.

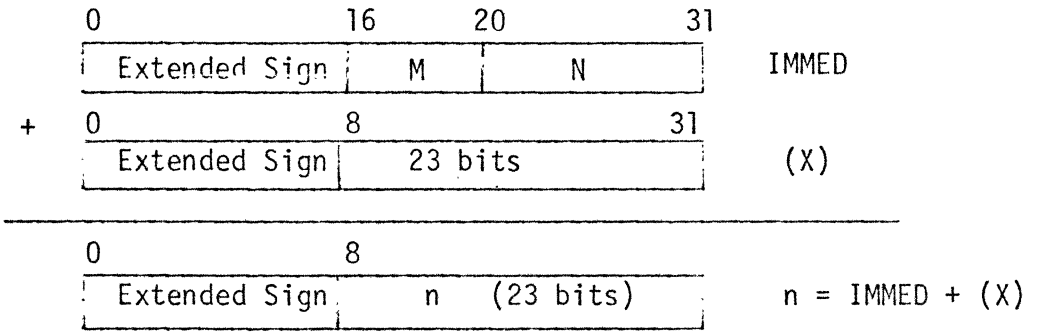
LOAD WORD IMMEDIATE (LI)

Operation Code	54, 5C
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate
Symbolic Notation	n → R

The immediate operand is entered into the register indicated by the R-field. In OP code 54, an R-field range from 0 thru F addresses arithmetic registers 0 thru F. In OP code 5C, an R-field range from 0 thru 7 addresses index registers 0 thru 7 and an R-field range from 8 thru F addresses vector registers 0 thru 7.

OP Code	R-Field Destination	Register Loaded
54	AR (0 thru F)	Arithmetic
5C	R range 0 thru 7 addresses XR 0 thru 7.	Index
5C	R range 8 thru F addresses VR 0 thru 7.	Vector

Programming Note: Whole word immediate operands for load instructions are formed from the combined M and N fields of the instruction word with extended sign (2's complement representation for negative numbers). The left half of IMMED consists of the extended sign of the most significant bit of the right half of IMMED. This immediate operand can be modified by an index register when $X \neq 0$. For this case, the contents of index register X is interpreted as a signed number (2's complement representation for negative numbers) within the range $-2^{23} \leq (X) \leq 2^{23} - 1$. In effect, the sign bit in the eighth bit position of the contents of index register X is extended into the most significant eight bits (bit position 0 through 7) before being added to IMMED. The true 32-bit value contained in index register X remains unchanged; the sign extension occurs in the index unit hardware and not in the register file. The modified immediate operand, n, is restricted to the range $-2^{23} \leq n \leq 2^{23} - 1$ by virtue of the fact that the parallel adder in the index unit is only 24-bits wide. The sign bit in the eighth bit position of n is extended into the most significant eight bits of n before being used as a modified immediate operand by the arithmetic unit.



Result code for load immediate instructions: The result code register is set according to the arithmetic value of the operand in the register indicated by the R-field after the load operation is complete. The three allowable values of the result code are as follows:

Contents of R after load	Result code (RL, RG, RE)
Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: None.

LOAD HALFWORD (LH)

Operation Code	15
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	α_h, halfword level
Symbolic Notation	$(\alpha_h) \rightarrow AR_{1h}$ where AR_{1h} indicates the left half of register AR.

The halfword (16 bits) from location α_h is entered into the left half of arithmetic register AR. The right half of register AR remains unchanged. Note that α_h represents an address for which displacement indexing is used and as such denotes a halfword address. An odd index value selects halfwords from the least significant half (right half) of a central memory or register whole word. An even index value addresses the left halfword of a central memory or register whole word. The left halfword is selected when not indexed.

Programming Note: Halfword memory operand selection for normal (not reversed) halfword address uses the LSB of the index register as shown below:

T-field	Contents of Index Register selected by T	Halfword is selected from Central Memory
0	None	Left half
1-7	Even Value	Left half
1-7	Odd Value	Right half
8-F	-	Depends on LSB of index register contents specified by terminal indirect address

Result code for load halfword instruction: The result code register is set according to the arithmetic value of the operand in the left half of the arithmetic register indicated by the R-field after the load operation is complete. The three allowable values of the result code are as follows:

Contents of AR&h after load	Result code (RL, RG, RE)
Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

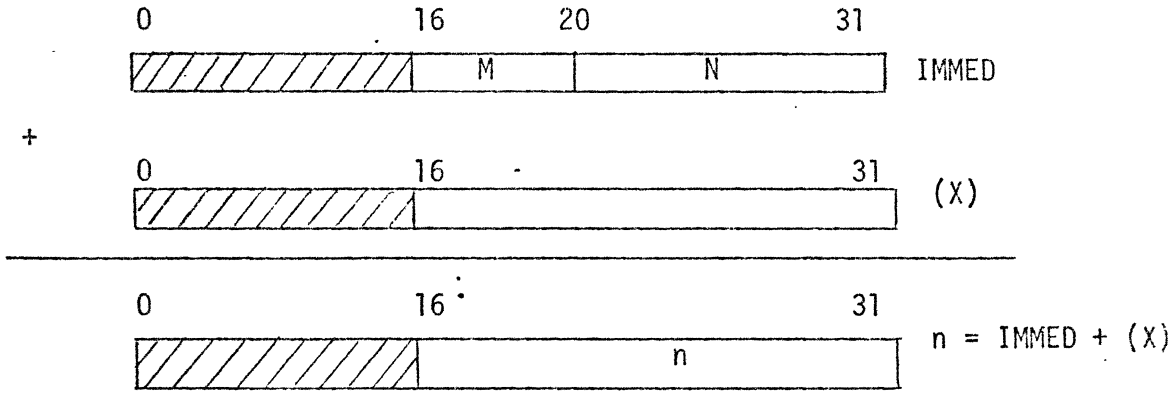
Program Interruption: None

LOAD IMMEDIATE HALFWORD (LIH)

The least significant 16-bits of the immediate operand is loaded into the left half of arithmetic register AR. The right half of register AR remains unchanged.

Operation Code	55
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate
Symbolic Notation	$n_{16-31} \rightarrow AR_{lh}$;

Programming Note: The combined M and N fields form the immediate operand for halfword instructions. The MSB of the right half of the instruction word is the sign bit. Negative numbers are represented in 2's complement form. This immediate operand can be modified by the right half of index register X. If X ≠ 0, the index register specified by X is added to the halfword immediate operand. For this case, the 16th bit position of index register X is a sign bit. If X = 0, no modification occurs.



Result code for load immediate halfword instructions: The result code register is set according to the arithmetic value of the operand in the left half of the arithmetic register indicated by the R-field after the load operation is complete. The three allowable values of the result code are as follows:

Contents of AR_{lh} after load	Result Code (RL, RG, RE)
Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: None.

LOAD MEMORY RIGHT HALFWORD INTO ARITHMETIC REGISTER RIGHT HALFWORD (LR)

Operation Code	1D
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	α_{hr} halfword level reversed
Symbolic Notation	$(\alpha_{hr}) \rightarrow AR_{rh}$

For this instruction, the right half of a central memory or register whole word is selected when not indexed. If indexed, an even index value selects words from the right half of a central memory or register whole word. An odd index value addresses the left halfword of the next consecutive singleword. This convention is just opposite to that of the LH instruction previously described. The operand selected is entered into the right half of arithmetic register AR. The left half of register AR remains unchanged.

Programming Note: When an array is addressed consecutively by indexing with this instruction (or with LL), an even index value addresses the right half of a memory or register whole word as in the preceding paragraph. But, it should be noted that when this even index value is incremented by unity (forming an odd index value), the operand acquired by this instruction is from the left half of the next consecutive central memory or register whole word.

T-field	Contents of Index Register selected by T	Halfword is selected from (Reversed halfword addressing)
0	None	Right half
1-7	Even Value	Right half
1-7	Odd Value	Left half of next consecutive singleword
8-F	-	Depends on index register contents specified by terminal indirect address

Result code for load right halfword instruction: The result code register is set according to the arithmetic value of the operand in the right half of the arithmetic register indicated by the R-field after the load operation is complete. The three allowable values of the result code are as follows:

Contents of AR_{rh} after load	Result Code (RL, RG, RE)
Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: None.

LOAD MEMORY RIGHT HALFWORD
INTO ARITHMETIC REGISTER
LEFT HALFWORD (LL)

The memory operand or register operand is selected as in an LR instruction. The operand selected is entered into the left half of arithmetic register AR.

Operation Code	19
Type Format	1
Operand Format	R, @ = N,X
Type Addressing	α_{hr}, halfword level reversed
Symbolic notation	$(\alpha_{hr}) \rightarrow AR_{1h}$

Programming Note: See programming note under LR instruction.

Result code for load left halfword instruction: The result code register is set according to the arithmetic value of the operand in the left half of the arithmetic register indicated by the R-field after the load operation is complete. The three allowable values of the result code are as follows:

Contents of AR_{1h} after load	Result Code (RL, RG, RE)
Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: None.

LOAD DOUBLEWORD (LD)

The doubleword from location ∝d is entered into the doubleword register designated by the R-field.

Operation Code	17
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	∝d, doubleword level
Symbolic notation	(∝d) → ARD where ARD denotes an arithmetic doubleword register from an even-odd address pair.

Programming notes: Doublewords are restricted to even-odd memory address pairs and register address pairs. The index register is displaced one bit position to the left when addressing doublewords so that the K^{th} doubleword in a data array is addressed by an index value equal to K.

Result code for doubleword load: The result code register is set according to the arithmetic value of the doubleword operand in register ARD (composed of whole word registers AR and AR + 1). The three allowable values of the result code are as follows:

Contents of ARD after load	Result Code (RL, RG, RE)
Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: Specification error if R-field is odd.

LOAD MAGNITUDE FIXED POINT (LM)

Load register AR with the magnitude of the contents of address α.

Operation Code	3C
Type Format	1
Operand Format	R, @ = N,X
Type Addressing	α, singleword level
Symbolic Notation	|(α)| → AR

AR is loaded with the 2's complement of (α) if (α) is negative.

Result code: Set according to the arithmetic value of the register operand after the operation is complete. The three allowable indicators are as follows:

Contents of AR after load instruction	Result code (RL, RG, RE)
Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: Overflow is possible with this instruction. When the operand is the largest negative number, i.e., $8000\ 0000_{Hex}$ overflow will result from complementing. The result in register AR will be $8000\ 0000_{Hex}$.

LOAD MAGNITUDE FIXED POINT HALFWORD (LMH)

Load register AR_{1h} with the magnitude of the halfword contained in location α_h.

Operation Code	3D
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	α_h halfword leve
Symbolic Notation	$\|(\alpha_h)\| \rightarrow AR_{1h}$

AR_{1h} is loaded with the 2's complement of (&) if (α) is negative.

Programming Note: The right halfword or left halfword source operand is selected according to the contents of the index register as in the LH instruction.

Result code for load magnitude instruction: The result code register is set according to the arithmetic value of the operand in the left half of the arithmetic register indicated by the R-field after the load operation is complete. The three allowable values of the result code are as follows:

Contents of AR_{1h} after load	Result Code (RL, RG, RE)
Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: Overflow is possible with this instruction. When the operand is the largest negative number, i.e., 8000_H, overflow will result from complementing. The result in register AR_{1h} will be 8000_H.

LOAD MAGNITUDE FLOATING POINT (LMF)

Load register AR with the magnitude of the contents of address α.

Operation Code	3E
Type Format	1
Operand Format	R, @ = N,X
Type Addressing	α, singleword level
Symbolic Notation	|(α)| → AR

Result code for load magnitude instructions: The result code register is set according to the arithmetic value of the operand in the register indicated by the R-field after the load operation is complete. The two allowable values of the result code are as follows:

Contents of AR after load	Result Code
Negative	Not possible
Positive	0 1 0
Zero	0 0 1

Program Interruption: None.

LOAD MAGNITUDE FLOATING POINT DOUBLEWORD (LMD)

Load register ARD with the magnitude of the contents of doubleword address αd.

Operation Code	3F
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	αd, doubleword level
Symbolic Notation	$\lvert(\alpha_d)\rvert \rightarrow$ ARD

Programming Note: Doublewords are restricted to even-odd memory address pairs and register address pairs.

Result code for doubleword load: The result code register is set according to the arithmetic value of the doubleword operand in register ARD (composed of whole word registers AR and AR + 1). The two allowable values of the result code are as follows:

Contents of ARD after load	Result Code (RL, RG, RE)
Negative	Not possible
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: Specification error if R-field is odd.

LOAD NEGATIVE WORD
FIXED POINT (LN)

Load register AR with the negative of the contents of address α.

Operation Code	30
Type Format	1
Operand Format	R, @ = N,X
Type Addressing	α, singleword level
Symbolic Notation	-(α) → AR

Programming Note: Two's complement representation is used for negative numbers in fixed point instructions.

Result code for load negative instructions: The result code register is set according to the arithmetic value of the operand in the register indicated by the R-field after the load operation is complete. The three allowable values of the result code are as follows:

Contents of AR after load	Result code (RL, RG, RE)
Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: Overflow possible on LN. When the operand is the largest negative number, i.e., $8000\ 0000_H$ overflow will result from complementing. The result in register R will be $8000\ 0000_H$.

LOAD NEGATIVE HALFWORD
FIXED POINT (LNH)

Load the left half of register AR with the negative of the contents of address α_h.

Operation Code	31
Type Format	1
Operand Format	R, (a) = N,X
Type Addressing	α_h, halfword level
Symbolic Notation	$-(\alpha_h) \rightarrow AR_{LH}$

Programming Note: Two's complement representation is used for negative numbers in fixed point instructions. The right halfword or left halfword source operand is selected according to the contents of the index register as in the LH instruction.

Result code for load negative instructions: The result code register is set according to the arithmetic value of the operand in the left half of the arithmetic register indicated by the R-field after the load operation is complete. The three allowable values of the result code are as follows:

Contents of AR_{LH} after Load	Result code (RL, RG, RE)
Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: Overflow possible. When the operand is the largest negative number, i.e., 8000_H overflow will result from complementing. The result in register R will be 8000_H.

LOAD NEGATIVE FLOATING POINT WORD (LNF)

Load register AR with the negative of the contents of address α.

Operation Code	32
Type Format	1
Operand Format	R, ⓐ = N,X
Type Addressing	α, singleword level
Symbolic Notation	$-(\alpha) \rightarrow$ AR

Programming Note: The negative form of floating point numbers involves a change of sign only.

Result code for load negative instructions: The result code register is set according to the arithmetic value of the operand in the register indicated by the R-field after the load operation is complete. The three allowable values of the result code are as follows:

Contents of AR after load	Result code (RL, RG, RE)
Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: None.

LOAD NEGATIVE FLOATING POINT DOUBLEWORD (LND)

Load doubleword register ARD with the negative of the contents of address α_d.

Operation Code	33
Type Format	1
Operand Format	R, ⓐ = N, X
Type Addressing	α_d, doubleword level
Symbolic Notation	$-(\alpha_d) \rightarrow$ ARD

Programming Note: The negative form of floating point numbers involves a change of sign only. Also, the R-field must be even, specifying an even-odd singleword register address pair.

Result Code for Load Negative Doubleword: The result code register is set according to the arithmetic value of the doubleword operand in register ARD (composed of whole word registers AR and AR + 1). The three allowable values of the result code are as follows:

Contents of ARD after load	Result code (RL, RG, RE)
Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: Specification error if R-field is odd.

LOAD NEGATIVE MAGNITUDE
FIXED POINT SINGLEWORD (LNM)

Load register AR with the negative of the magnitude of (α).

Operation code	38
Type Format	1
Operand Format	R, (a) = N, X
Type Addressing	α, singleword level
Symbolic Notation	$-\lvert(\alpha)\rvert \rightarrow$ AR

Programming Note: Two's complement representation is used for negative numbers in fixed point instructions.

Result code for load negative magnitude instructions: The result code register is set according to the arithmetic value of the operand in the register indicated by the R-field after the load operation is complete. The two allowable values of the result code are as follows:

Contents of AR after load	Result code (RL, RG, RE)
Negative	(1, 0, 0)
Positive	Not possible
Zero	(0, 0, 1)

Program Interruption: None.

LOAD NEGATIVE MAGNITUDE
HALFWORD FIXED POINT (LNMH)

Load halfword register AR_{1h} with the negative of the magnitude of (α_h).

Operation Code	39
Type Format	1
Operand Format	R, (a) = N, X
Type Addressing	α_h, halfword level
Symbolic Notation	$-\lvert(\alpha_h)\rvert \rightarrow AR_{1h}$

Programming Notes: Right halfword or left halfword from central memory or register file is selected according to the LSB of the index register specified by the T-field as in the LH instruction. Negative numbers are represented in two's complement notation.

Result code for load negative magnitude instructions: The result code register is set according to the arithmetic value of the operand in the left half of the register indicated by the R-field after the load operation is complete. The two allowable values of the result code are as follows:

Contents of AR&h after load	Result code (RL, RG, RE)
Negative	(1, 0, 0)
Positive	Not possible
Zero	(0, 0, 1)

Program Interruption: None.

LOAD NEGATIVE MAGNITUDE
FLOATING POINT SINGLEWORD (LNMF)

Load singleword register AR with the negative of the magnitude of (α).

Operation Code	3A
Type Format	1
Operand Format	R, ⓐ = N, X
Type Addressing	α, singleword level
Symbolic Notation	$-\|(\alpha)\| \rightarrow$ AR

Programming Note: The negative form of floating point numbers involves a change of sign only.

Result Code for Load Negative Magnitude Instructions: The result code register is set according to the arithmetic value of the operand in the register indicated by the R-field after the load operation is complete. The two allowable values of the result code are as follows:

Contents of AR after load	Result code (RL, RG, RE)
Negative	(1, 0, 0)
Positive	Not Possible
Zero	(0, 0, 1)

Program Interruption: None.

LOAD NEGATIVE MAGNITUDE DOUBLEWORD (LNMD)

Load doubleword register ARD with the negative of the magnitude of (αd).

Operation Code	3B
Type Format	1
Operand Format	R, @ . X
Type Addressing	αd, eword level
Symbolic Notation	- ARD

Programming Notes: The negative form of floating point numbers in ess a change of sign only. Also, the R-field must be even, specifying an even-odd singleword register address pair.

Result Code for Load Negative Magnitude Instruction: The result code register is set according to the arithmetic value of the doubleword operand register ARD (composed of whole word registers AR and AR + 1). The two allowable values of the result code are as follows:

Contents of ARD after load	Result code (RL, RE, EE)
Negative	(1, 0, 0)
Positive	Not possible
Zero	(0, 0, 1)

Program Interruption: Specification error if R-field is odd.

LOAD FILE (LF)

Operation Code	1B
Type Format	9
Operand Format	M, (a), N, X
Type Addressing	α , octet level
Symbolic Notation	(α)oct. → RF

The contents of central memory octet α are entered into the eight word register file designated by the R-field. There are six forms of the LF instruction having OP code 1B. The distinction is made according to the contents of the R-field: OP code 1B may be broken down as follows:

R-field	Designation register file designated by R-field	Hexadecimal locations
X000	Base register file A	0-7
X001	Base register file B	8-F
X010	General arithmetic reg. file C	10-17
X011	General arithmetic reg. file D	18-1F
X100	Index register file X	20-27
X101	Vector register file V	28-2F
X11X	No operation, no registers loaded	

Programming Notes: The three low order bits of α are ignored so that octet α means the octet in which word α is located. Also, if $\alpha \leq 2F$ and M = 0, then α references the register file which contains α . An R-field bit of X indicates that the bit is ignored (a don't care).

Result code: Not affected.

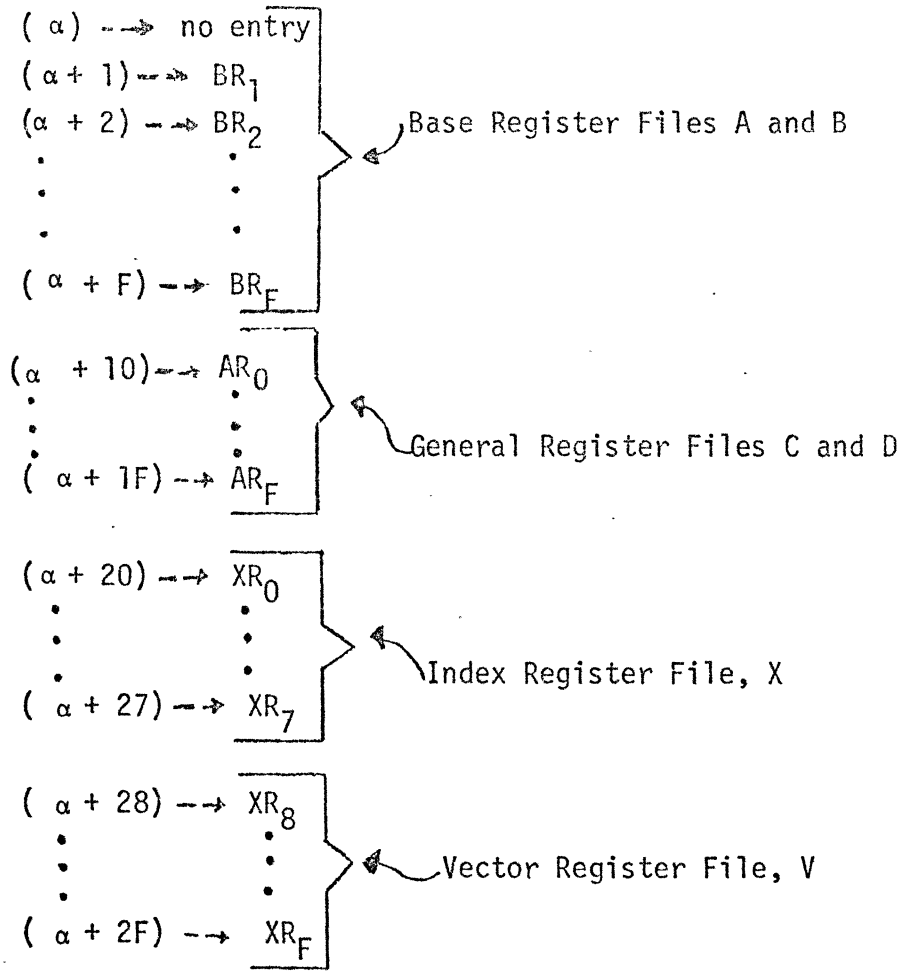
Program Interruption: None.

LOAD FILE MULTIPLE (LFM)

The contents of six consecutive memory octets starting with location α are entered into all six eight-word register files (A, B, C, D, X, and V) in physical locations 1 through 2F (hexadecimal).

Operation Code	1F
Type Format	3
Operand Format	ⓐN,X
Type Addressing	α , octet level
Symbolic Notation	(α)oct. → all RF

Programming Note: The three low order bits of α are ignored so that location α is on a full octet boundary.



Result Code: Not affected.

Program Interruption: Specification error if $\alpha \leq 2F$ and M = 0.

EXCHANGE WORDS (XCH)

An exchange instruction stores the contents of arithmetic register AR into location α and stores the previous contents of location α into register AR. It exchanges the contents of AR and α. Only a single length exchange instruction exists in the ASC.

Operation Code	1A
Type Format	2
Operand Format	R, (a) N, X
Type Addressing	α, singleword level
Symbolic Notation	$(AR) \rightarrow \alpha$ $(\alpha) \rightarrow AR$

Programming Note: If $\alpha \leq 2F$ and M = 0, then two registers are exchanged. One is always an arithmetic register and the other may be selected from any register of the register file.

Result Code: Not affected.

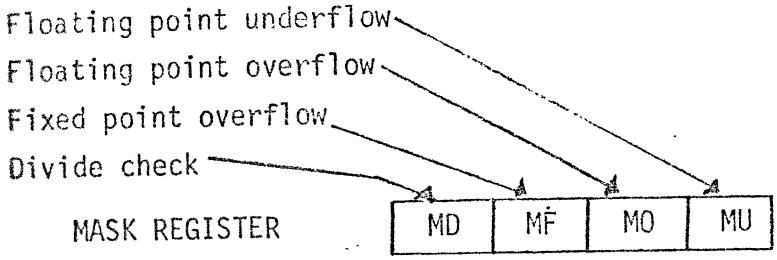
Program Interruption: None.

LOAD ARITHMETIC
EXCEPTION MASK (LAM)

Load bits 4 through 7 of the contents of location α into the four bit arithmetic exception mask register.

Operation Code	12
Type Format	8
Operand Format	(a) = N, X
Type Addressing	α , singleword level
Symbolic Notation	$(\alpha)_{4-7}$ → AE Mask

MASKABLE AE CONDITIONS:



An interrupt signal from the CP to the PP will become true if an arithmetic exception is detected and the mask bit corresponding to that arithmetic exception has been set to a "one".

1 ~ masked ON ~ interrupt possible

0 ~ masked OFF ~ no interrupt possible

Programming Note: Address α may address the base or index registers which may contain a mask stored from a previous BLB or BLX instruction. The R-field is not used.

Result Code: Not affected.

AE Condition Code: Not affected.

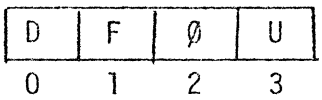
Program Interruption: Alteration of the AE mask register will cause an arithmetic exception program interruption if the corresponding bits of the AE condition register and AE mask register are both "one" after the LAM instruction is completed.

LOAD ARITHMETIC
EXCEPTION CONDITION (LAC)

Load bits 0 through 3 of the contents of location α into the four bit arithmetic exception condition code register.

Operation Code	13
Type Format	8
Operand Format	' (a) = N, X
Type Addressing	α , singleword level
Symbolic Notation	$(\alpha)_{0-3}$ → AE Cond.

AE CONDITION REGISTER



D - Divide check

F - Fixed point overflow

Ø - Floating point overflow

U - Floating point underflow

Bit (D,F,Ø,U) zero, indicates no arithmetic exception condition.

Bit (D,F,Ø or U) equal one, indicates an active AE condition.

Programming Note: Address α may address the base or index registers which may contain an arithmetic exception condition code stored from a previous BLB or BLX instruction. The R-field is not used.

Result Code: Not affected.

AE Condition Code: Changed to the state of $(\alpha)_{0-3}$.

Program Interruption: Alteration of the AE condition register will cause an arithmetic exception program interruption if the corresponding bits of the AE condition register and AE mask register are both "one" after the LAC instruction is completed.

LOAD LOOK AHEAD (LLA)

Operation Code	16
Type Format	10
Operand Format	I
Type Addressing	Immediate
Symbolic Notation	$N_{24-31} \rightarrow BC$
	$(PC) \rightarrow BR$

This instruction provides the instruction look-ahead unit in the CP control hardware with advance address information relating to a subsequent Branch instruction for which it is known that the branch path will normally be taken. The LLA instruction does not influence the decision that is made by a Branch instruction, it only increases the execution speed of a closed instruction loop.

The LLA instruction loads the 8 least significant bits (bit positions 24 through 31) of the N-field of this instruction into the branch counter (BC) internal to the CP control. Also, the program counter (PC) is entered into the branch address register (BR) internal to the CP control. These internal registers (BC and BR) are not addressable by CP program.

Programming Notes: The value, N, which is entered into the branch counter should be equal to the difference of instruction locations between this Load Look-Ahead instruction and the Branch instruction for which the LLA is intended. For example, if the LLA instruction is stored in location 401 and the Branch instruction is stored in location 429, then the value of N should be equal to 28. If any other Branch instructions occur between these locations and if one of the branch paths is taken, then the information in the branch look-ahead hardware will be dicarded. Such intermediate Branch instructions proceed normally when the branch paths are not taken and the look-ahead information remains current while the branch counter continues its count down. Regardless of whether or not an intermediate Skip instruction results in the skip being taken, the computation to determine the value for N should include both the skip instruction and the instruction following the Skip instruction.

The maximum applicable loop size is 255 instructions including the LLA instruction. The LLA instruction must be included at the top of the program loop so that the branch counter and branch address register can be re-initialized each time the program returns to the top of the loop.

The R-field is not used.

Result Code: Not affected.

Program Interruption: None.

LOAD ONE'S COMPLEMENT (LO)

The one's complement of location α is entered into arithmetic register AR. Bit positions with 1's in (α) are loaded as 0's in AR and vice versa.

Operation Code	1E
Type Format	1
Operand Format	R, (a) = N, X
Type Addressing	α, singleword level
Symbolic Notation	$(\alpha)_j \rightarrow AR_j$ where j ranges from 0 through 31

Result Code for Load One's Complement Instruction: The result code register is set according to the arithmetic value of the operand in the register indicated by the AR field after the load operation is complete. The three allowable values of the result code are as follows:

Contents of AR after load	Result Code (RL, RG, RE)
Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: None.

Duplicate of pg 51

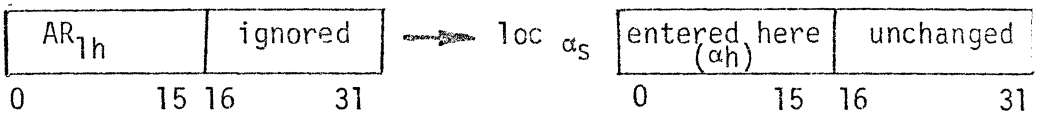
STORE HALFWORD (STH)

The contents of the left half of arithmetic register AR is stored into halfword location α_h.

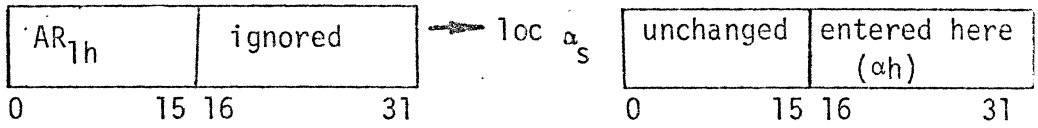
Operation Code	25
Type Format	2
Operand Format	R, @ N, X
Type Addressing	αh, halfword level
Symbolic Notation	$(AR_{1h}) \rightarrow \alpha_h$

Programming Note: α_h represents an address for which displacement indexing is used and as such denotes the proper halfword address. In particular, an odd index value selects the least significant half (right half) of a singleword location. An even index value addresses the left halfword of a singleword location. The left half word is selected when not indexed.

No index or even index value.



Odd index value.



Result Code for Store Halfword Instruction: The result code register is set according to the arithmetic value of the operand in halfword location αh after the store operation is complete. The three allowable values of the result code are as follows:

Contents of address αh after store	Result code (RL, RG, RE)
Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

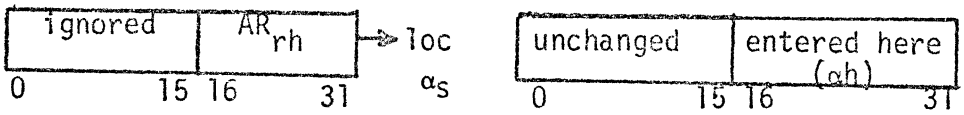
Program Interruption: None.

STORE ARITHMETIC REGISTER HALFWORD INTO RIGHT HALFWORD (STR)

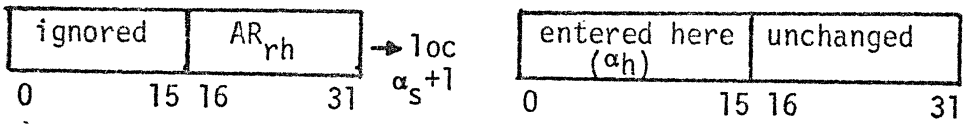
Operation Code	2D
Type Format	2
Operand Format	R, @ N, X
Type Addressing	α_{hr}, halfword reversed
Symbolic Notation	$(AR_{rh}) \rightarrow \alpha_{hr}$

The right half of arithmetic register AR is stored into the right half of a singleword location when not indexed. If indexed, an even index value selects the right half of a singleword location for storage. An odd index value addresses the left halfword of the next consecutive singleword. This convention is reversed from normal addressing.

No index or even index value.



Odd index value



Programming Note: When an array is addressed consecutively by indexing with this instruction (or with STL), an even index value addresses the right half of a singleword location as in the preceding paragraph, but when this even index value is incremented by unity (forming an odd index value), the register operand is entered into the left half of the next consecutive singleword location.

Also, when $\alpha_h \leq$ 5F (halfword address) and M = 0, then the operand is stored into a halfword register using the same addressing convention as is used for memory.

Result code for store instructions: The result code register is set according to the arithmetic value of the operand in halfword location α_h after the store operation is complete. The three allowable values of the result code are as follows:

Contents of address α_h after store	Result code (RL, RG, RE)
Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

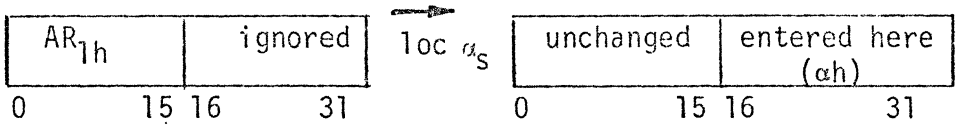
Program Interruption: None

STORE ARITHMETIC
REGISTER LEFT HALFWORD
INTO RIGHT HALFWORD (STL)

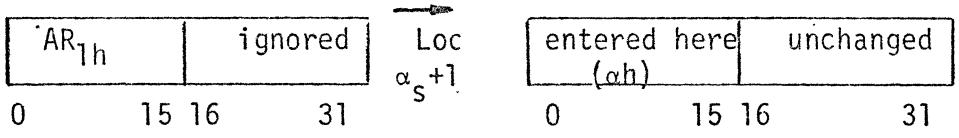
Operation Code	29
Type Format	2
Operand Format	R, @ N, X
Type Addressing	α_{hr}, halfword reversed
Symbolic Notation	$(AR_{1h}) \rightarrow \alpha_{hr}$

The left half of arithmetic register AR is stored into the right half of a singleword location α_s when not indexed. If indexed, an even index value selects the right half of a singleword location for storage. An odd index value addresses the left halfword of the next consecutive singleword. This convention is reversed from normal addressing.

No index or even index value.



Odd index value.



Programming Note: See programming note under STR instruction.

Result code for Store Instructions: The result code register is set according to the arithmetic value of the operand in the central memory address, α_h, after the store operation is complete. The three allowable values of the result code are as follows:

Contents of address α_h after store	Result code (RL, RG, RE)
Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: None.

STORE DOUBLEWORD (STD)

The contents of the doubleword register ARD is stored into the doubleword location specified by α_d.

Operation Code	27
Type Format	2
Operand Format	R, @ N, X
Type Addressing	α_d, doubleword level
Symbolic Notation	(ARD) → α_d

Programming Note: Doubleword registers and doubleword locations are restricted to even-odd singleword address pairs.

Result code for store instructions: The result code register is set according to the arithmetic value of the operand in location α_d after the store operation is complete. The three allowable values of the result code are as follows:

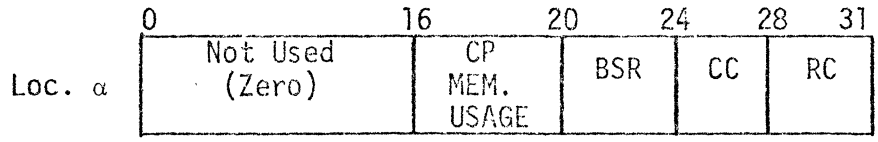
Contents of address α_d after store	Result code (RL, RG, RE)
Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: Possible specification error if R is odd.

STORE PROGRAM STATUS WORD (SPS)

Operation Code	22
Type Format	3
Operand Format	@ N, X
Type Addressing	α , singleword level
Symbolic Notation	PSW → α

The full program status doubleword (64 bits) is stored in central memory only on special signal from the PPU. This instruction (SPS) stores only the first half (32 bits) of the doubleword. The last half of the program status doubleword is stored into singleword location α. This instruction stores the MEM USAGE, BSR, CC and RC status information into memory according to the format:



where MEM USAGE refers to Central Memory Usage information.

BSR refers to the Branch or Skip condition register.

CC refers to the Condition Code.

RC refers to the Result Code.

See Program Status Doubleword for more detailed information on the meaning of these bit designations.

Programming Note: The R-field is not used.

Result Code: Unaffected.

Program Interruption: None.

STORE ZERO IN WORD (STZ)

Operation Code	20
Type Format	3
Operand Format	@ N, X
Type Addressing	α , singleword level
Symbolic Notation	0 → α

Zero is stored into location α. The R-field is not used.

Result Code Setting for Store Zero Instructions: The result code is set only to the value: (RL, RG, RE) = (0, 0, 1).

Program Interruption: None.

STORE ZERO IN
HALFWORD (STZH)

Zero is stored into location α_h. The R-field is not used. Displacement indexing selects normal halfword addresses, i.e., left halfwords are selected when the index value is even.

Operation Code	21
Type Format	3
Operand Format	@ N, X
Type Addressing	α_h, halfword level
Symbolic Notation	$0 \rightarrow \alpha_h$

Result Code Setting for Store Zero Instructions: The result code is set only to the value: (RL, RG, RE) = (0, 0, 1).

Program Interruption: None.

STORE ZERO IN
DOUBLEWORD (STZD)

Zero is stored into location α_d. The R-field is not used.

Operation Code	23
Type Format	3
Operand Format	@ N, X
Type Addressing	α_d, doubleword level
Symbolic notation	$0^d \rightarrow \alpha_d$

Result Code Setting for Store Zero Instructions: The result code is set only to the value: (RL, RG, RE) = (0, 0, 1).

Program Interruption: None.

STORE NEGATIVE FIXED
PQINT SINGLEWORD (STN)

Store the negative of the contents of singleword arithmetic register AR into location α. The 2's complement of the value in AR is stored.

Operation Code	34
Type Format	2
Operand Format	R, (a) N, X
Type Addressing	α , singleword level
Symbolic Notation	- (AR) → α

Result Code Setting for Store Negative Instructions: The result code is set according to the arithmetic value in location α after the store is complete. The three possible values of the result code are:

Contents of α after store	Result Code (RL, RG, RE)
Zero	(0, 0, 1)
Positive	(0, 1, 0)
Negative	(1, 0, 0)

Program Interruption: Fixed point overflow will occur if the arithmetic register contains the largest negative value $(8000\ 0000)_{hex}$. The result stored into location α is $(8000\ 0000)_{hex}$ if the largest negative value is stored.

STORE NEGATIVE
FIXED POINT HALFWORD (STNH)

Store the negative of the contents of the left half of arithmetic register AR into halfword location α_h. The 2's complement of the halfword value is stored.

Operation Code	35
Type Format	2
Operand Format	R, (a) N, X
Type Addressing	α_h, halfword level
Symbolic Notation	$-(AR_{1h}) \rightarrow \alpha_h$

Result Code Setting for Store Negative Instructions: The result code is set according to the arithmetic value in halfword location α_h after the store is complete. The three possible values of the result code are:

Contents of α_h after store	Result code (RL, RG, RE)
Zero	(0, 0, 1)
Positive	(0, 1, 0)
Negative	(1, 0, 0)

Program Interruption: Fixed point overflow will occur if the arithmetic register contains the largest negative value $(8000)_{hex}$. The result stored in location α is $(8000)_{hex}$ if the largest negative value is stored.

STORE NEGATIVE FLOATING
POINT SINGLEWORD (STNF)

Store the negative of the contents of singleword arithmetic register AR into location α. This involves a change of sign in floating point representation.

Operation Code	36
Type Format	2
Operand Format	R, (a) N, X
Type Addressing	α, Singleword le
Symbolic Notation	$-(AR) \rightarrow \alpha$

Result Code Setting for Store Negative Instructions: The result code is set according to the arithmetic value in location α after the store is complete. The three possible values of the result code are:

Contents of α after store	Result Code (RL, RG, RE)
Zero	(0, 0, 1)
Positive	(0, 1, 0)
Negative	(1, 0, 0)

Program Interruption: None.

STORE NEGATIVE FLOATING POINT DOUBLEWORD (STND)

Store the negative of the contents of doubleword arithmetic register ARD into location α_d. This involves a change of sign in floating point representation.

Operation Code	37
Type Format	2
Operand Format	R, ⓐ N, X
Type Addressing	α_d, Doubleword level
Symbolic Notation	$-(ARD) \rightarrow \alpha_d$

Result Code Setting for Store Negative Instructions. The result code is set according to the arithmetic value in doubleword location α_d after the store is complete. The three possible values of the result code are:

Contents of α_d after store	Result Code (RL, RG, RE)
Zero	(0, 0, 1)
Positive	(0, 1, 0)
Negative	(1, 0, 0)

Program Interruption: Specification error if R-field is odd.

STORE ONE'S COMPLEMENT SINGLEWORD (STO)

Store the one's complement of the contents of singleword arithmetic register AR into location α. Zero bits in AR are stored as ones in α and vice versa.

Operation Code	2E
Type Format	2
Operand Format	R, ⓐ N, X
Type Addressing	α, singleword level
Symbolic Notation	$\overline{(AR)_j} \rightarrow \alpha_j$ for j range 0 thru 3

Result Code Setting for Store One's Complement: The result code is set according to the arithmetic value in singleword location α after the store is complete. The three possible values of the result code are:

Contents of α after store	Result Code (RL, RG, RE)
Zero	(0, 0, 1)
Positive	(0, 1, 0)
Negative	(1, 0, 0)

Program Interruption: None.

STORE ONE'S COMPLEMENT HALFWORD (STOH)

Store the one's complement of the contents of the left half of arithmetic register AR into halfword location α_h. Zero bits in AR_{lh} are stored as ones in α and vice versa.

Operation Code	2A
Type Format	2
Operand Format	R, (a) N, X
Type Addressing	α_h, halfword level
Symbolic Notation	$(AR_{lh})_j \rightarrow \alpha_{hj}$ for j range 0 thru 15

Result Code Setting for Store One's Complement: The result code is set according to the arithmetic value in halfword location α_h after the store is complete. The three possible values of the result code are:

Contents of α_h after store	Result Code (RL, RG, RE)
Zero	(0, 0, 1)
Positive	(0, 1, 0)
Negative	(1, 0, 0)

Program Interruption: None.

STORE REGISTER FILE (STF)

The contents of eight consecutive registers, from the register file designated by the R-field, are stored into central memory octet α .

Operation Code	2B
Type Format	9
Operand Format	M,@N,X
Type Addressing	α, octet level
Symbolic Notation	RF → α octet

R-field	Source Register File Designated by the R-Field	Hexadecimal Location in Register File
X000	Base register File A	0-7
X001	Base register File B	8-F
X010	Arithmetic register File C	10-17
X011	Arithmetic register File D	18-1F
X100	Index Register File X	20-27
X101	Vector File V	28-2F
X11X	Octet of zeros	

Programming Notes: The three least significant bits of singleword address α are ignored when an octet referenced is the one which contains singleword address α.

Also, if $\alpha \leq 2F$ and M = 0, then α references an octet of the register file. Register files may be moved or loaded with an octet of zeros by using such an address with an R-field value of 6 or 7.

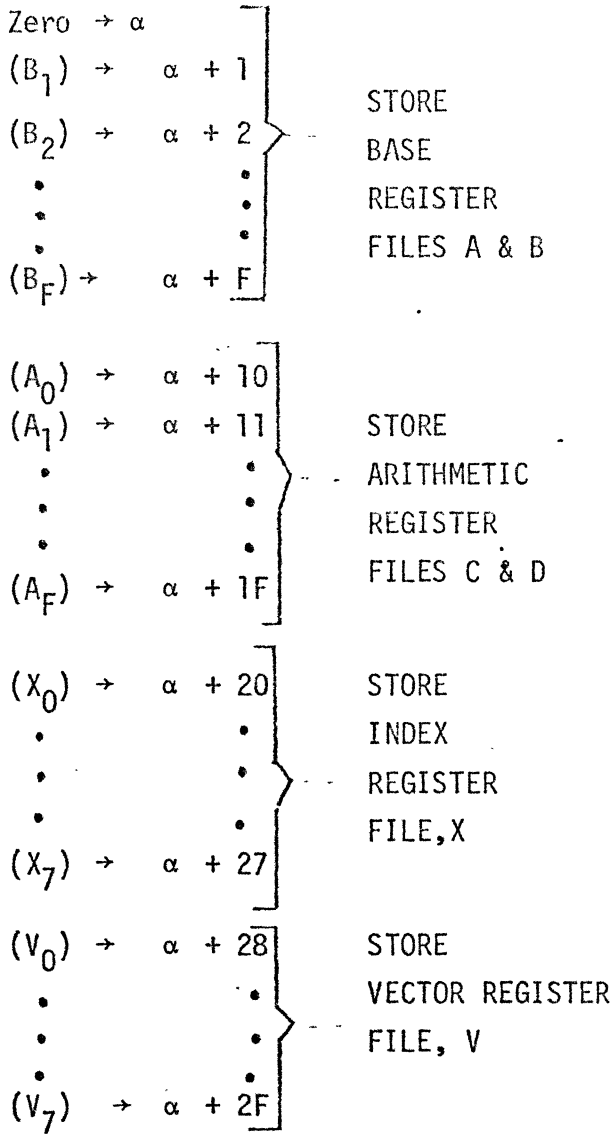
Result Code: Not affected.

Program Interruption: None.

STORE REGISTER FILES, MULTIPLE (STFM)

Operation Code	2F
Type Format	3
Operand Format	(a) N, X
Type Addressing	α , octet level
Symbolic Notation	All RF→ α(6 octets)

The contents of six consecutive register octets (register files A, B, C, D, X, and V) are stored into six consecutive memory octets starting with location α.



Programming Note: If α ≤2F and M = 0, then an illegal operation is specified. This results in program interruption.

Result Code: Not affected.

Program Interruption: Specification error if α ≤2F and M = 0.

ARITHMETIC INSTRUCTIONS

Operation Code	40, 60, 62
Type Format	1
Operand Format	R, ⓐ = N, X
Type Addressing	α, singleword level
Symbolic Notation	(R) + (α) → R

ADD WORD (A)

The four forms of the add word instruction indicated by OP codes are listed as follows:

OP Code	R-Field	Register Involved
40	AR	Arithmetic
60	BR	Base
62	Range 0 thru 7 addresses XR 0 thru 7.	Index
62	Range 8 thru F addresses VR 0 thru 7.	Vector

OP Code 40 - The whole word fixed point value in location α is added to arithmetic register AR specified by the R-field. Location α may be in central memory or in one of the registers of the register file. The result is stored into arithmetic register AR.

OP Code 60: Add the whole word contents of location α to the contents of base register BR, specified by the R-field, and store the result into base register BR.

OP Code 62 - Add the whole word contents of location α to the contents of index or vector register XR or VR, specified by the R-field and store the result into index or vector register XR or VR.

Index register if $0 \leq R \leq 7$

Vector register if $8 \leq R \leq F$.

Result Code Setting: The result code is set according to the result of the operation as follows:

Arithmetic Operation Result	Result Code (RL, RG, RE)
(R) < 0	(1, 0, 0)
(R) > 0	(0, 1, 0)
(R) = 0	(0, 0, 1)

Program Interruption: Fixed point overflow is possible.

ADD WORD IMMEDIATE (AI)

The four forms of the add word immediate instruction indicated by OP codes are listed as follows:

Operation Code	50, 70, 72
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate
Symbolic Notation	(R) + n → R

OP Code	R-Field	Register Involved
50	AR	Arithmetic
70	BR	Base
72	Range 0 thru 7 addresses XR 0 thru 7.	Index
72	Range 8 thru F addresses VR 0 thru 7.	Vector

OP Code 50 - Add the singleword arithmetic immediate operand to the contents of arithmetic register AR, specified by the R-field, and store the result into arithmetic register AR.

OP Code 70 - Add the singleword arithmetic immediate operand to base register BR, specified by the R-field, and store the result into base register BR.

OP Code 72 - Add the singleword arithmetic immediate operand to the index or vector register specified by the R-field, and store the result into index or vector register XR or VR.

Result Code Setting: The result code is set according to the result of the operation as follows:

Arithmetic Operation Result	Result Code (RL, RG, RE)
(R) < 0	(1, 0, 0)
(R) > 0	(0, 1, 0)
(R) = 0	(0, 0, 1)

Program Interruption: Fixed point overflow is possible.

ADD HALFWORD (AH)

The halfword fixed point value in location α_h is added to the left half of singleword arithmetic register AR specified by the R-Field. The result is stored into the left half of arithmetic register AR.

Operation Code	41
Type Format	1
Operand Format	R, (a) = N, X
Type Addressing	α_h, halfword level
Symbolic Notation	$(AR_{1h}) + (\alpha_h) \rightarrow AR_{1h}$

Programming Notes: Halfword fixed point arithmetic operations acquire the register operand from the left half of arithmetic register AR. The second operand is a halfword from location α_h where α_h specifies a halfword address by normal displacement indexing. The result of a halfword arithmetic operation is stored into the left half of arithmetic register AR.

Location α_h may be in central memory or in the register file. If $\alpha_h \leq 5F$ and M = 0, then α_h addresses one of the 96 (decimal) halfword registers of the register file.

Result Code Setting: The result code is set according to the result of the operation as follows:

Arithmetic Operation Result	Result Code (RL, RG, RE)
$(AR_{1h}) < 0$	(1, 0, 0)
$(AR_{1h}) > 0$	(0, 1, 0)
$(AR_{1h}) = 0$	(0, 0, 1)

Program Interruption: Fixed point overflow is possible.

ADD HALFWORD IMMEDIATE (AIH)

The halfword immediate operand is added to the left half of single-word arithmetic register AR specified by the R-field. The result is stored into the left half of register AR.

Operation Code	51
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate
Symbolic Notation	$(AR_{lh}) + n \rightarrow AR_{lh}$

Programming Notes: The combined M&N fields form the immediate (IMMED) operand for halfword instructions. The MSB of the M-field is the sign bit. This immediate operand can still employ the indexing option to effect an operand modification, of the form IMMED + (X), where IMMED is defined above.

Result Code Setting: The result code is set according to the result of the operation as follows:

Arithmetic Operation Result	Result Code (RL, RG, RE)
(R) < 0	(1, 0, 0)
(R) > 0	(0, 1, 0)
(R) = 0	(0, 0, 1)

Program Interruption: Fixed point overflow is possible.

ADD FLOATING POINT WORD (AF)

The singleword floating point operand in location α is added to arithmetic register AR specified by the R-field. The result is stored into register AR.

Operation Code	42
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	α singleword level
Symbolic Notation	(AR) + (α) → AR

Programming Note: Floating point inputs must be hexadecimally normalized.

Result Code Setting: The result code is set according to the result of the operation as follows:

Arithmetic Operation Result	Result Code (RL, RG, RE)
(R) < 0	(1, 0, 0)
(R) > 0	(0, 1, 0)
(R) = 0	(0, 0, 1)

Program Interruption: Floating point overflow and underflow are possible.

ADD FLOATING POINT DOUBLEWORD (AFD)

Operation Code	43
Type Format	1
Operand Format	R, (a) = N, X
Type Addressing	α_d, doubleword level
Symbolic Notation	(ARD) + (α_d) → ARD

The doubleword floating point operand in location α_d is added to arithmetic register ARD specified by the even R-field value. The result is stored into doubleword register ARD.

Programming Notes: Doubleword floating point arithmetic operations involve two doubleword operands. One operand is from registers AR and AR + 1, considered as a 64-bit floating point number, where AR + 1 contains the low order bits of the number. The other operand (memory operand) is a doubleword from singleword memory locations α and α + 1. The result is stored into registers AR and AR + 1, where the R-field ranges from 0 through E (hexadecimal). Only even-odd register address pairs and memory address pairs are permissible for doubleword operations.

Floating point inputs must be hexadecimally normalized.

Result Code Setting: The result code is set according to the result of the operation as follows:

Arithmetic Operation Result	Result Code (RL, RG, RE)
(R) < 0	(1, 0, 0)
(R) > 0	(0, 1, 0)
(R) = 0	(0, 0, 1)

Program Interruption: Floating point overflow and underflow are possible. Program specification error if R-field is odd.

ADD MAGNITUDE FIXED POINT WORD (AM)

The magnitude of the singleword fixed point value in location α is added to the arithmetic register specified by the R-field. The result is stored into arithmetic register AR designated by the R-field.

Operation Code	44
Type Format	1
Operand Format	R, (a) = N, X
Type Addressing	α , singleword
Symbolic Notation	(AR) + |(α)| → AR

Result Code Setting: The result code (RL, RG, RE) is set according to the result of the operation as follows:

(AR) < 0	(1, 0, 0)
(AR) > 0	(0, 1, 0)
(AR) = 0	(0, 0, 1)

Program Interruption: Overflow is possible. Special cases are shown below when the operand from Central Memory is the largest negative value. Fixed point overflow is possible when (α) is not the largest negative value and (AR) is positive.

Value of (AR)	Value of (α)	Result of (AR) + |(α)|	Result Code Setting	Fixed Point Overflow
Zero	8000 0000	(AR) + 8000 0000	Negative	Yes
Positive	8000 0000	(AR) + 8000 0000	Negative	Yes
Negative	8000 0000	(AR) + 8000 0000	Positive	No

ADD MAGNITUDE FIXED
POINT HALFWORD (AMH)

Operation Code	45
Type Format	1
Operand Format	R, (a) = N, X
Type Addressing	α_h, halfword
Symbolic Notation	$(AR_{1h}) + \|(\alpha_h)\| \rightarrow AR$

The magnitude of the fixed point value in halfword location α_h is added to the left half of singleword arithmetic register AR specified by the R-field. The result is stored into the left half of whole word arithmetic register AR designated by the R-field.

Result Code Setting: The result code (RL, RG, RE) is set according to the halfword result of the operation as follows:

$(AR_{1h}) < 0$	(1, 0, 0)
$(AR_{1h}) > 0$	(0, 1, 0)
$(AR_{1h}) = 0$	(0, 0, 1)

Program Interruption: Overflow is possible. Special cases are shown below when the operand from Central Memory is the largest negative value. Fixed point overflow is possible when (α) is not the largest negative value and (AR_{1h}) is positive.

Value of (AR_{1h})	Value of (α_h)	Result of $(AR_{1h}) + \|(\alpha_h)\|$	Result Code Setting	Fixed Point Overflow
Zero	8000	(AR_{1h}) + 8000	Negative	Yes
Positive	8000	(AR_{1h}) + 8000	Negative	Yes
Negative	8000	(AR_{1h}) + 8000	Positive	No

ADD MAGNITUDE FLOATING POINT WORD (AMF)

Operation Code	46
Type Format	1
Operand Format	R, (a) = N, X
Type Addressing	α, singleword
Symbolic Notation	(AR) + (α) → AR

The magnitude of the singleword floating point value in location α is added to arithmetic register AR specified by the R-field. The result is stored into arithmetic register AR.

Programming Note: Floating point inputs must be hexadecimally normalized.
Result Code Setting: The result code (RL, RG, RE) is set according to the result of the operation as follows:

(AR) < 0	(1, 0, 0)
(AR) > 0	(0, 1, 0)
(AR) = 0	(0, 0, 1)

Program Interruption: Floating point overflow is possible.

ADD MAGNITUDE FLOATING POINT DOUBLEWORD (AMFD)

Operation Code	47
Type Format	1
Operand Format	R, (a) = N, X
Type Addressing	αd, doubleword
Symbolic Notation	(ARD) + (α_d) → ARD

The magnitude of the doubleword floating point value in location α_d is added to doubleword arithmetic register ARD specified by the even R-field value. The result is stored into doubleword arithmetic register ARD. Only even-odd register and memory address pairs may be used.

Programming Note: Floating point inputs must be hexadecimally normalized.
Result Code Setting: The result code (RL, RG, RE) is set according to the result of the operation as follows:

(ARD) < 0	(1, 0, 0)
(ARD) > 0	(0, 1, 0)
(ARD) = 0	(0, 0, 1)

Program Interruption: Floating point overflow is possible. Specification error if R-field is odd.

SUBTRACT WORD (S)

The singleword fixed point value in location is subtracted from arithmetic register AR specified by the R-field. The result is stored into arithmetic register AR.

Operation Code	48
Type Format	1
Operand Format	R, (a) = N, X
Type Addressing	, singleword
Symbolic Notation	(AR) - (α) → AR

Result Code Setting: The result code (RL, RG, RE) is set according to the result of the operation as follows:

(AR) < 0	(1, 0, 0)
(AR) > 0	(0, 1, 0)
(AR) = 0	(0, 0, 1)

Program Interruption: Fixed point overflow is possible.

SUBTRACT WORD IMMEDIATE (SI)

Subtract the singleword arithmetic immediate operand from the contents of arithmetic register AR specified by the R-field. The result is stored into register AR.

Operation Code	58
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate
Symbolic Notation	(AR) - n → AR

Programming Note: For arithmetic singleword immediate operand instructions, the sign bit is extended into the most significant half of the word.

This immediate operand can still employ the indexing option to effect an operand modification of the form IMMED + (X).

Result Code Setting: Fixed point arithmetic instructions set the result code (RL, RG, RE) according to the result of the operation as follows:

(AR) < 0	(1, 0, 0)
(AR) > 0	(0, 1, 0)
(AR) = 0	(0, 0, 1)

Program Interruption: Fixed point overflow is possible.

SUBTRACT HALFWORD (SH)

The halfword fixed point value in location α_h is subtracted from the left half of arithmetic register AR specified by the R-field. The result is stored into the left half of register AR.

Operation Code	49
Type Format	1
Operand Format	R, (a) = N, X
Type Addressing	α_h, halfword
Symbolic Notation	$(AR_{1h}) - (\alpha_h) \rightarrow AR_{1h}$

Result Code Setting: The result code (RL, RG, RE) is set according to the result of the operation as follows:

$(AR_{1h}) < 0$	(1, 0, 0)
$(AR_{1h}) > 0$	(0, 1, 0)
$(AR_{1h}) = 0$	(0, 0, 1)

Program Interruption: Fixed point overflow is possible.

SUBTRACT HALFWORD IMMEDIATE (SIH)

Subtract the halfword immediate operand from the contents of the left half of arithmetic register AR specified by the R-field. The result is stored into the left half of register AR.

Operation Code	59
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate
Symbolic Notation	$(AR_{1h}) - n \rightarrow AR_{1h}$

Programming Note: The combined M&N fields form the immediate operand for halfword instructions. The MSB of the M-field is the sign bit. This immediate operand can still employ the indexing option to effect an operand modification of the form IMMED + (X).

Result Code Setting: The result code (RL, RG, RE) is set according to the result of the operation as follows:

$(AR_{1h}) < 0$	(1, 0, 0)
$(AR_{1h}) > 0$	(0, 1, 0)
$(AR_{1h}) = 0$	(0, 0, 1)

Program Interruption: Fixed point overflow is possible.

SUBTRACT FLOATING POINT WORD (SF)

Operation Code	4A
Type Format	1
Operand Format	R, (a) = N, X
Type Addressing	α , singleword
Symbolic Notation	(AR) - (α) → AR

The singleword floating point value in location α is subtracted from arithmetic register AR specified by the R-field. The result is stored into arithmetic register AR.

Programming Note: Floating point inputs must be hexadecimally normalized.

Result Code Setting: The result code (RL, RG, RE) is set according to the result of the operation as follows:

AR < 0	(1, 0, 0)
AR > 0	(0, 1, 0)
AR = 0	(0, 0, 1)

Program Interruption: Floating point overflow and underflow are possible.

SUBTRACT FLOATING POINT DOUBLEWORD (SFD)

Operation Code	4B
Type Format	1
Operand Format	R, (a) = N, X
Type Addressing	α_d, doubleword
Symbolic Notation	(ARD) - (αd) → ARD

The doubleword floating point value in location α_d is subtracted from arithmetic register ARD specified by the even R-field value. The result is stored into register ARD.

Programming Notes: Double floating point arithmetic operations involve two doubleword operands. One operand is from register AR and AR + 1, considered as a 64-bit floating point number, where AR + 1 contains the low order bits of the number. The other operand is a doubleword from singleword locations α and α + 1. The result is stored into registers AR and AR + 1, where AR ranges fr 0 through E (hexadecimal). Only even-odd register address pairs and memory address pairs are permissible for doubleword operations.

Floating point inputs must be hexadecimally normalized

Result Code Setting: The result code (RL, RG, RE) is set according to the result of the operation as follows:

ARD < 0	(1, 0, 0)
ARD > 0	(0, 1, 0)
ARD = 0	(0, 0, 1)

Program Interruption: Floating point overflow and underflow are possible. Specification error if R-field is odd.

SUBTRACT MAGNITUDE FIXED POINT WORD (SM)

The magnitude of the singleword fixed point value in location α is subtracted from arithmetic register AR specified by the R-field. The result is stored into arithmetic register AR.

Operation Code	4C
Type Format	1
Operand Format	R, ⓐ = N, X
Type addressing	α , singleword
Symbolic Notation	(AR) - |(α)| → AR

Programming Note: Fixed point magnitude involves taking the 2's complement if the number is negative.

Result Code Setting: The result code (RL, RG, RE) is set according to the result of the operation as follows:

(AR) < 0	(1, 0, 0)
(AR) > 0	(0, 1, 0)
(AR) = 0	(0, 0, 1)

Program Interruption: Overflow is possible. Special cases are shown below when the operand from central memory is the largest negative value.

Value of (AR)	Result of (AR) - |(α)|	Result Code Setting	Fixed Point Overflow
Zero	(AR) - 8000 0000	Negative	No
Positive	(AR) - 8000 0000	Negative	No
Negative	(AR) - 8000 0000	Positive	Yes

Also, fixed point overflow is possible when (α) is not the largest negative value and (AR) is negative.

SUBSTRACT MAGNITUDE FIXED POINT HALFWORD (SMH)

The magnitude of the halfword fixed point value in location α_h is subtracted from the left half of singleword arithmetic register AR. The result is stored into the left half of register AR.

Operation Code	4D
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	α_h, halfword
Symbolic Notation	$(AR_{1h}) - \lvert(\alpha_h)\rvert \rightarrow AR_{1h}$

Programming Note: See programming note under SM instruction.

Result Code Setting: The result code (RL, RG, RE) is set according to the result of the operation as follows:

$(AR_{1h}) < 0$	(1, 0, 0)
$(AR_{1h}) > 0$	(0, 1, 0)
$(AR_{1h}) = 0$	(0, 0, 1)

Program Interruption: Overflow is possible. Special cases are shown below when the operand from central memory is the largest negative value.

Value of (AR_{1h})	Result of $(AR_{1h}) - \lvert(\alpha_h)\rvert$	Result Code Setting	Fixed Point Overflow
Zero	(Ar_{1h}) - 8000	Negative	No
Positive	(AR_{1h}) - 8000	Negative	NO
Negative	(AR_{1h}) - 8000	Positive	YES

Also, fixed point overflow is possible when (α_h) is not the largest negative value and (AR) is negative.

SUBTRACT MAGNITUDE FLOATING POINT WORD (SMF)

The magnitude of the single-word floating point value in location α is subtracted from arithmetic register AR specified by the R-field. The result is stored into register AR.

Operation Code	4E
Type Format	1
Operand Format	R, ⓐ = N, X
Type Addressing	α, singleword
Symbolic Notation	(AR) - $\lvert(\alpha)\rvert \rightarrow$ AR

Programming Notes: Floating point magnitude involves changing the sign of the fraction if the number is negative. Floating point inputs must be hexadecimally normalized.

Result Code Setting: The result code (RL, RG, RE) is set according to the result of the operation as follows:

(AR) $<$ 0	(1, 0, 0)
(AR) $>$ 0	(0, 1, 0)
(AR) = 0	(0, 0, 1)

Program Interruption: Floating point underflow is possible.

SUBTRACT MAGNITUDE FLOATING POINT DOUBLEWORD (SMFD)

The magnitude of the doubleword floating point value in location α_d is subtracted from doubleword arithmetic register ARD specified by the even R-field value. The result is stored into doubleword register ARD.

Operation Code	4F
Type Format	1
Operand Format	R, ⓐ = N, X
Type Addressing	α_d, doubleword
Symbolic Notation	(ARD) - $\lvert(\alpha_d)\rvert \rightarrow$ ARD

Programming Note: Floating point magnitude involves changing the sign of the fraction if the number is negative. Floating point inputs must be hexadecimally normalized.

Result Code Setting: The result code (RL, RG, RE) is set according to the result of the operation as follows:

(ARD) $<$ 0	(1, 0, 0)
(ARD) $>$ 0	(0, 1, 0)
(ARD) = 0	(0, 0, 1)

Program Interruption: Floating point underflow is possible.
Specification error if R-field is odd.

MULTIPLY WORD
FIXED POINT WORD (M)

Operation Code	6C, 68, 6A
Type Format	1
Operand Format	R, (a) = N, X
Type Addressing	α_S, singleword
Symbolic Notation	(R) * (α_S) → R

The three forms of the M instruction, indicated by OP codes are listed in the following table:

OP Code	R-Field	Register Addressed
6C	AR	Arithmetic
68	BR	Base
6A	Range 0 thru 7 address R 0 thru 7	Index
6A	Range 8 thru F address VR 0 thru 7	Vector

OP Code 6C: Multiple the contents of singleword arithmetic register AR by the contents of singleword location α . If the singleword register operand (AR) is selected from an even register address, the full 64-bit signed integer product is stored into an even-odd address pair (registers AR and AR+1). If the singleword register operand is selected from an odd register address (R-field is odd), then the least significant 32-bits of the 64-bit signed integer product is stored into the odd register address specified by R.

(AR) * (α_S) → ARD if R is even

(AR) * (α_S) → AR if R is odd.

OP Code 68: Multiply the singleword contents of base register BR by the singleword contents of location α . The 32 least significant bits of the 64-bit signed integer product are stored into base register BR. There is no product length option -

(BR) * (α_S) → BR for R even or odd.

OP Code 6A: Multiply the singleword contents of index register XR or vector register VR by the singleword contents of location α . The 32 least significant bits of the 64-bit signed integer product are stored into index register XR or vector register VR. There is no product length option.

(XR) * (α_S) → XR for R range 0 thru 7.

(VR) * (α_S) → VR for R range 8 thru F.

Result Code Setting: The result code (RL, RG, RE) is set according to the results of the operation as follows:

(R) < 0	(1, 0, 0)
(R) > 0	(0, 1, 0)
(R) = 0	(0, 0, 1)

Program Interruption: Fixed point overflow is possible for OP code 6C when the R-field is odd and the product cannot be expressed in 32-bits. Fixed point overflow is possible for OP codes 68 and 6A if the product cannot be expressed in 32-bits. Fixed point overflow is indicated if the 33 most significant bits of the 64-bit product are not all zeros or not all ones.

MULTIPLY FIXED POINT
WORD IMMEDIATE (MI)

Operation Code	7C, 78, 7A
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate
Symbolic Notation	(R) * n → R

The three forms of the MI instruction, indicated by OP codes are as follows:

OP Code	R-Field	Register Addressed
7C	AR	Arithmetic
78	BR	Base
7A	Range 0 thru 7 addresses XR 0 thru 7	Index
7A	Range 8 thru F addresses VR 0 thru 7	Vector

OP Code 7C: Multiply the contents of singleword arithmetic register AR by the singleword arithmetic immediate operand. If the singleword register operand (AR) is selected from an even register address, then the full 64-bit signed integer product is stored into an even-odd register address pair (registers AR and AR+1). If the singleword register operand is selected from an odd register address (R-field is odd), then the least significant 32-bits of the 64-bit signed integer product is stored into the odd register address specified by R.

(AR) * n → ARD if R is even

(AR) * n → AR is R is odd.

OP Code 78: Multiply the singleword contents of base register BR by the singleword arithmetic immediate operand. The 32 least significant bits of the 64 bit signed integer product are stored into base register BR. There is no product length option

(BR) * n → BR for R even or odd

OP Code 7A: Multiply the singleword contents of index register XR or vector register VR by the singleword arithmetic immediate operand. The 32 least significant bits of the 64-bit signed integer product are stored into index register XR or vector register VR. There is no product length option.

(XR) * n → XR for R range 0 thru 7

(VR) * n → VR for R range 8 thru F.

Result Code Setting: The result code (RL, RG, RE) is set according to the results of the operation as follows:

(R) < 0	(1, 0, 0)
(R) > 0	(0, 1, 0)
(R) = 0	(0, 0, 1)

Program Interruption: Fixed point overflow is possible for OP Code 7C when the R-field is odd and the product cannot be expressed in 32-bits. Fixed point overflow is possible for OP codes 78 and 7A if the product cannot be expressed in 32-bits.

FIXED POINT MULTIPLY
HALFWORD (MH)

Multiply the contents of the left half of singleword arithmetic register AR by the operand from halfword location α h. The full 32-bit signed integer product is stored into singleword arithmetic register AR. There is no product length option.

Operation Code	6D
Type Format	1
Operand Format	R, (a) = N, X
Type Addressing	α_h, halfword
Symbolic Notation	$(AR_{1h}) * (\alpha h) \rightarrow AR$

Result Code Setting: The result code (RL, RG, RE) is set according to the singleword result of the operation as follows:

(AR) < 0	(1, 0, 0)
(AR) > 0	(0, 1, 0)
(AR) = 0	(0, 0, 1)

Program Interruption: None.

MULTIPLY FIXED POINT
HALFWORD IMMEDIATE (MIH)

Multiply the contents of the left half of singleword arithmetic register AR by the halfword immediate operand. The full 32-bit signed integer product is stored into singleword arithmetic register AR. There is no product length option.

Operation Code	7D
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate
Symbolic Notation	$(AR_{1h}) * n \rightarrow AR$

Result Code Setting: The result code (RL, RG, RE) is set according to the singleword result of the operation as follows:

(AR) < 0	(1, 0, 0)
(AR) > 0	(0, 1, 0)
(AR) = 0	(0, 0, 1)

Program Interruption: None.

MULTIPLY FLOATING POINT WORD (MF)

Operation Code	6E
Type Format	1
Operand Format	R, (a) = N, X
Type Addressing	α_s, singleword
Symbolic Notation	(AR) * (αs) → AR

Multiply the floating point contents of singleword arithmetic register AR by the contents of singleword location αs. The singleword floating point product is stored into arithmetic register AR.

Result Code Setting: The result code (RL, RG, RE) is set according to the result of the operation as follows:

(AR) < 0	(1, 0, 0)
(AR) > 0	(0, 1, 0)
(AR) = 0	(0, 0, 1)

Program Interruption: Floating point overflow and underflow are possible.

MULTIPLY FLOATING POINT DOUBLEWORD (MFD)

Operation Code	6F
Type Format	1
Operand Format	R, (a) = N, X
Type Addressing	α_d, doubleword
Symbolic Notation	(ARD) * (αd) → ARD

Multiply the floating point contents of doubleword arithmetic register ARD by the contents of doubleword location αd. The doubleword floating point product is stored into arithmetic register ARD.

Result Code Setting: The result code (RL, RG, RE) is set according to the result of the operation as follows:

(ARD) < 0	(1, 0, 0)
(ARD) > 0	(0, 1, 0)
(ARD) = 0	(0, 0, 1)

Program Interruption: Floating point overflow and underflow are possible. Specification error if R-field is odd.

DIVIDE FIXED POINT WORD (D)

Operation Code	64
Type Format	1
Operand Format	R, (a) = N, X
Type Addressing	α_s, singleword
Symbolic Notation	(ARD/(α s)→AR

This division is of the form: arithmetic register operand divided by location α The fixed point dividend is from the register operand and the divison is a singleword from location α s. If the dividend is selected from an even register address, then the dividend (a 64-bit signed integer) is acquired from the even-odd register address pair AR and AR+1. The 32-bit signed integer quotient is stored into the even register, AR. The odd register address, AR + 1, retains the low order 32-bits of the double length dividend.

If the dividend is selected from an odd register address, then the dividend (a 32-bit signed integer) is acquired from the odd register AR specified by R. The 32-bit signed integer quotient is stored into register AR.

(ARD)/(αs) → AR if R is even

(AR)/(αs) → AR if R is odd.

Result Code Setting: The result code (RL, RG, RE) is set according to the result of the operation as follows:

(AR) < 0 : (1, 0, 0)

(AR) > 0 (0, 1, 0)

(AR) = 0 (0, 0, 1)

Program Interruption: Fixed point overflow is indicated if the quotient cannot be expressed in 32-bits of register AR when the R-field is even. Also, a fixed point Divide Check is indicated if the divisor is equal to zero. In either case, an AU result is stored into register AR.

DIVIDE FIXED POINT
IMMEDIATE WORD (DI)

Operation Code	74
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate
Symbolic Notation	(ARD)/n → AR

This division is of the form: arithmetic register operand divided by the immediate operand. The fixed point dividend is from the register operand and the divisor is a singleword arithmetic immediate operand. If the dividend is selected from an even register address, then the dividend (a 64-bit signed integer) is acquired from the even-odd register address pair AR and AR + 1. The 32-bit signed integer quotient is stored into the even register address, AR. The odd register address, AR + 1, retains the low order 32-bits of the double length dividend. If the dividend is selected from an odd register address, then the dividend (a 32-bit signed integer) is acquired from the odd register AR specified by R. The 32-bit signed integer quotient is stored into register AR.

(ARD)/n → AR if R is even

(AR)/n → AR if R is odd

Result Code Setting: The result code (RL, RG, RE) is set according to the result of the operation as follows:

(AR) < 0 (1, 0, 0)

(AR) > 0 (0, 1, 0)

(AR) = 0 (0, 0, 1)

Program Interruption: Fixed point overflow is indicated if the quotient cannot be expressed in 32-bits of register AR when the R-field is even. Also, a fixed point divide check is indicated if the divisor is equal to zero. In either case, an AU result is stored into register AR.

DIVIDE FIXED POINT
HALFWORD (DH)

Operation Code	65
Type Format	1
Operand Format	R, ⓐ = N, X
Type Addressing	α_h, halfword
Symbolic Notation	$(AR)/(\alpha h) \rightarrow AR_{1h}$

This division is of the form: arithmetic register operand divided by location αh. The fixed point dividend is a 32-bit signed integer from register AR and the division is a 16-bit signed integer from halfword location αh. A 16-bit signed integer quotient is formed in the left half of register AR. The right half of register AR retains the 16 low order bits of the dividend.

Result Code Setting: The result code (RL, RG, RE) is set according to the result of the operation as follows:

$(AR_{1h}) < 0$ (1, 0, 0)

$(AR_{1h}) > 0$ (0, 1, 0)

$(AR_{1h}) = 0$ (0, 0, 1)

Program Interruption: A fixed point overflow is indicated if the quotient cannot be expressed in 16 bits. A fixed point divide check is indicated if the divisor equals zero. In either case, an AU result is stored into the left half of register AR.

DIVIDE FIXED POINT
HALFWORD IMMEDIATE (DIH)

Operation Code	75
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate
Symbolic Notation	$(AR)/n \rightarrow AR_{1h}$

This division is of the form: arithmetic register operand divided by the halfword immediate operand. The fixed point dividend is a 32-bit signed integer from register AR and the divisor is a 16-bit signed integer from the halfword immediate operand. A 16-bit signed integer quotient is formed in the left half of register AR. The right half of register AR retains the 16 low order bits of the dividend.

Result Code Setting: The result code (RL, RG, RE) is set according to the result of the operation as follows:

$(AR_{1h}) < 0$ (1, 0, 0)

$(AR_{1h}) > 0$ (0, 1, 0)

$(AR_{1h}) = 0$ (0, 0, 1)

Program Interruption: A fixed point overflow is indicated if the quotient cannot be expressed in 16 bits. A fixed point divide check is indicated if the divisor equals zero. In either case, an AU result is stored into the left half of register AR.

DIVIDE FLOATING POINT WORD (DF)

Operation Code	66
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	α_s, singleword
Symbolic Notation	(AR) / (αs) → AR

This division is of the form: arithmetic register operand divided by location α_s. The floating point dividend is from the singleword arithmetic register AR and the division is a singleword from location αs. The floating point quotient is stored into singleword register AR.

Programming Note: Floating point division results in a quotient which is the same length as the dividend and divisor from which the quotient was obtained.

Result Code Setting: The result code (RL, RG, RE) is set according to the result of the operation as follows:

(AR)<0	(1, 0, 0)
(AR)>0	(0, 1, 0)
(AR)=0	(0, 0, 1)

Program Interruption: Floating point exponent overflow or underflow is possible. Divide check and exponent overflow is indicated if the divisor equals zero. See the table of infinite and indefinite forms (pages 13-16) for definitions of the register result following these program interruption conditions.

DIVIDE FLOATING POINT DOUBLEWORD (DFD)

Operation Code	67
Type Format	1
Operand Format	R, @ = N, X
Type addressing	αd, doubleword
Symbolic Notation	(ARD) / (αd) → ARD

This division is of the form arithmetic register operand divided by location αd. The floating point dividend is from the doubleword arithmetic register ARD and the divisor is a doubleword from location αd. The floating point quotient is stored into doubleword register ARD.

Programming Note: Floating point division results in a quotient which is the same length as the dividend and divisor from which the quotient was obtained.

Result Code Setting: The result code (RL, RG, RE) is set according to the result of the operation as follows.

(ARD) < 0	(1, 0, 0)
(ARD) > 0	(0, 1, 0)
(ARD) = 0	(0, 0, 1)

Program Interruption: Floating point exponent overflow or underflow is possible. Divide check and exponent overflow is indicated if the divisor equals zero. Specification error if R-field is odd. See the table of infinite and indefinite forms for definitions of the register result following these program interruption conditions.

LOGICAL INSTRUCTIONS

AND WORD (AND)

A logical AND operation is applied at each bit position (j) of two operand singlewords. One operand is from arithmetic register AR and the other operand is from location α. The result is stored into register AR.

Operation Code	E0
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	α, singleword
Symbolic Notation	$(AR)_j \wedge (\alpha)_j \rightarrow AR_j$
	for j range 0 thru 3

Result Code Setting: The result code (RL, RG, RE) is set after each logical operation according to the logical properties of the result as shown below.

All bits are "zero"	(0, 0, 1)
All bits are "one"	(0, 1, 0)
Mixed "ones" and "zeros"	(1, 0, 0)

Program Interruption: None.

AND WORD IMMEDIATE (ANDI)

The logical AND operation is applied at each bit position (j) of two operand singlewords. One operand is from arithmetic register AR and the other operand is the singleword logical immediate operand. The result is stored into register AR.

Operation Code	F0
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate Logical
Symbolic Notation	$(AR)_j \; n \; n_j \rightarrow AR_j$
	for j range 0 thru 31

Programming Note: Singleword logical immediate operands are formed from the combined M and N fields of the instruction word. Zeros are located in the left halfword and the M and N fields make up the right halfword. This immediate operand may be modified (prior to the logical operation) by adding to it, the 24 LSB's of an index register designated by the X-field. If X = 0, no modification occurs.

Result Code Setting: The result code (RL, RG, RE) is set after each logical operation according to the logical properties of the result as shown below.

All bits are "zero"	(0, 0, 1)
All bits are "one"	(0, 1, 0)
Mixed "ones" and "zeros"	(1, 0, 0)

Program Interruption: None.

OR WORD (OR)

A logical OR operation is applied at each bit position (j) of two operand singlewords. One operand is from arithmetic register AR and the other operand from location α. The result is stored into register AR.

Operation Code	E4
Type Format	1
Operand Format	R, (a)= N, X
Type Addressing	α, singleword
Symbolic Notation	$(AR)_j \vee (\alpha)_j \rightarrow AR_j$
	for j range 0 thru 31

Result Code Setting: The result code (RL, RG, RE) is set after each logical operation according to the logical properties of the result as shown below.

All bits are "zero"	(0, 0, 1)
All bits are "one"	(0, 1, 0)
Mixed "ones" and "zeros"	(1, 0, 0)

Program Interruption: None.

OR WORD IMMEDIATE (ORI)

A logical OR operation is applied at each bit position (j) of two operand singlewords. One operand is from arithmetic register AR and the other operand is the singleword logical immediate operand. The result is stored into register AR.

Operation Code	F4
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate logical
Symbolic Notation	$(AR)_j \vee n_j \rightarrow AR_j$
	for j range 0 thru 31

Result Code Setting: The result code (RL, RG, RE) is set after each logical operation according to the logical properties of the result as shown below.

All bits are "zero"	(0, 0, 1)
All bits are "one"	(0, 1, 0)
Mixed "ones" and "zeros"	(1, 0, 0)

Program Interruption: None.

EXCLUSIVE OR WORD (XOR)

A logical EXCLUSIVE OR operation is applied at each bit position (j) of two operand singlewords. One operand is from arithmetic register AR and the other operand is from location α. The result is stored into register AR.

Operation Code	E8
Type Format	1
Operand Format	R, ⓐ = N, X
Type Addressing	α, singleword
Symbolic Notation	$(AR)_j \oplus (\alpha)_j \rightarrow AR_j$
	for j range 0 thru 31

Result Code Setting: The result code (RL, RG, RE) is set after each logical operation according to the logical properties of the result as shown below.

All bits are "zero"	(0, 0, 1)
All bits are "one"	(0, 1, 0)
Mixed "ones" and "zeros"	(1, 0, 0)

Program Interruption: None

EXCLUSIVE OR WORD IMMEDIATE (XORI)

A logical EXCLUSIVE OR operation is applied at each bit position (j) of two operand singlewords. One operand is from arithmetic register AR and the other operand is the singleword logical immediate operand. The result is stored into register AR.

Operation Code	F8
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate logical
Symbolic Notation	$(AR)_j \oplus n_j \rightarrow AR_j$
	for j range 0 thru 31

Result Code Setting: The result code (RL, RG, RE) is set after each logical operation according to the logical properties of the result as shown below.

All bits are "zero"	(0, 0, 1)
All bits are "one"	(0, 1, 0)
Mixed "ones" and "zeros"	(1, 0, 0)

Program Interruption: None.

EQUIVALENCE WORD (EQC)

A logical EQUIVALENCE operation is applied at each bit position (j) of two operand singlewords. One operand is from arithmetic register AR and the other operand is from location α. The result is stored into register AR.

Operation Code	EC
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	α, singleword
Symbolic Notation	$(AR)_j \odot (\alpha)_j \rightarrow AR_j$
	for j range 0 thru 31

Result Code Setting: The result code (RL, RG, RE) is set after each logical operation according to the logical properties of the result as shown below.

All bits are "zero"	(0, 0, 1)
All bits are "one"	(0, 1, 0)
Mixed "ones" and "zeros"	(1, 0, 0)

Program Interruption: None.

EQUIVALENCE WORD IMMEDIATE (EQCI)

A logical EQUIVALENCE operation is applied at each bit position (j) of two operand singlewords. One operand is from arithmetic register AR and the other operand is the singleword logical immediate operand. The result is stored into register AR.

Operation Code	FC
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate/logical
Symbolic Notation	$(AR)_j \odot n_j \rightarrow AR_j$
	for j range 0 thru 31

Result Code Setting: The result code (RL, RG, RE) is set after each logical operation according to the logical properties of the result as shown below.

All bits are "zero"	(0, 0, 1)
All bits are "one"	(0, 1, 0)
Mixed "ones" and "zeros"	(1, 0, 0)

Program Interruption: None.

AND DOUBLEWORD (ANDD)

A logical AND operation is applied at each bit position (j) of two operand doublewords. One operand is from arithmetic register ARD and the other operand is from location αd. The result is stored into register ARD.

Operation Code	E1
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	αd, doubleword
Symbolic Notation	$(ARD)_j \wedge (\alpha d)_j \rightarrow ARD_j$
	for j range 0 thru 63

Result Code Setting: The result code (RL, RG, RE) is set after each logical operation according to the logical properties of the result as shown below.

All bits are "zero"	(0, 0, 1)
All bits are "ones"	(0, 1, 0)
Mixed "ones" and "zeros"	(1, 0, 0)

Program Interruption: Specification error if R-field is odd.

OR DOUBLEWORD (ORD)

A logical OR operation is applied at each bit position (j) of two operand doublewords. One operand is from arithmetic register ARD and the other operand is from location αd. The result is stored into register ARD.

Operation Code	E5
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	αd, doubleword
Symbolic Notation	$(ARD)_j \vee (\alpha d)_j \rightarrow ARD_j$
	for j range 0 thru 63

Result Code Setting: The result code (RL, RG, RE) is set after each logical operation according to the logical properties of the result as shown below.

All bits are "zero"	(0, 0, 1)
All bits are "one"	(0, 1, 0)
Mixed "ones" and "zeros"	(1, 0, 0)

Program Interruption: Specification error if R-field is odd.

EXCLUSIVE OR DOUBLEWORD (XORD)

A logical EXCLUSIVE OR operation is applied at each bit position (j) of two operand doublewords. One operand is from arithmetic register ARD and the other operand is from location αd. The result is stored into register ARD.

Operation Code	E9
Type Format	1
Operand Format	R, (a) = N, X
Type Addressing	αd, doubleword
Symbolic Notation	$(ARD)_j \oplus (\alpha d)_j \rightarrow ARD_j$
	for j range 0 thru 63

Result Code Setting: The result code (RL, RG, RE) is set after each logical operation according to the logical properties of the result as shown below.

All bits are "zero"	(0, 0, 1)
All bits are "ones"	(0, 1, 0)
Mixed "ones" and "zeros"	(1, 0, 0)

Program Interruption: Specification error if R-field is odd.

EQUIVALENCE DOUBLEWORD (EQCD)

A logical EQUIVALENCE operation is applied at each bit position (j) of two operand doublewords. One operand is from arithmetic register ARD and the other operand is from location αd. The result is stored into register ARD.

Operation Code	ED
Type Format	1
Operand Format	R, (a) = N, X
Type Addressing	αd, doubleword
Symbolic Notation	$(ARD)_j \odot (\alpha d)_j \rightarrow ARD_j$
	for j range 0 thru 63

Result Code Setting: The result code (RL, RG, RE) is set after each logical operation according to the logical properties of the result as shown below.

All bits are "zero"	(0, 0, 1)
All bits are "ones"	(0, 1, 0)
Mixed "ones" and "zeros"	(1, 0, 0)

Program Interruption: Specification error if R-field is odd.

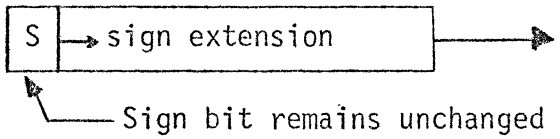
SHIFT INSTRUCTIONS

ARITHMETIC SHIFT WORD (SA)

Operation Code	CO
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate
Symbolic Notation	$(AR)_j \rightarrow AR_{j-sc}$ (Arith.)

The contents of the single-word arithmetic register AR, designated by the R-field, is shifted arithmetically either right or left and the result is entered into register AR. The direction and the amount of shift is determined by the shift count (SC) information in the 7 LSB's of the N-field of the instruction word. The direction and amount of shift may be modified by the index register specified by the X-field when X ≠ 0. When modification occurs, MN and (X) are added the same as for halfword immediate operands. The least significant 7 bits of the result are interpreted as a twos complement number to determine the direction and amount of shift. This is equivalent to 7 bit addition. Overflow of the 7 bit shift count is not detected. For example, a minus 60 indexed by a minus 7 gives a plus 61; plus 60 indexed by plus 6 gives a minus 62. Bit position 25 of the N-field and of the contents of X is interpreted as the sign position for shift instructions only. A positive sign of the resulting shift count causes a left shift of SC bit positions. A negative sign of the resulting shift count causes a right shift of SC bit positions. The value of SC is within the range: $-64 \leq SC \leq +63$.

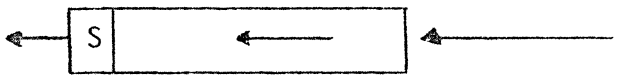
Arithmetic right shift (negative shift count)



bits shifted out of low-order bit position are lost

Arithmetic left shift (positive shift count)

bits shifted through sign position are lost



zeros are entered into low-order bit position

Programming Notes: The 7-bit shift count is used for all word sizes. If the resulting shift count should exceed the word size of 32-bits for singleword shift instructions, then the register result would appear as follows:

Arithmetic Left Shift - A register result of zero with overflow detection. There would be no overflow detection if the original value of the register before shifting was zero.

Arithmetic Right Shift - Either zero or minus one (fixed point, 2's complement) depending on whether the original register value was positive or negative, respectively. The M-field is not used but must be zero.

Also, the most significant bit of the T-field is not used, i.e., indirect shift counts are not possible. The M-field is not used, but must be zero.

Result Code Setting: The result code (RL, RG, RE) is set according to the arithmetic value of the result as follows:

(AR) $<$ 0	(1, 0, 0)
(AR) $>$ 0	(0, 1, 0)
(AR) = 0	(0, 0, 1)

Program Interruption: Fixed point overflow is detected, for arithmetic left shifts only, if the sign bit changes during the shift. The entire shift operation designated by the shift count is completed regardless of overflow conditions.

ARITHMETIC SHIFT HALFWORD (SAH)

The contents of the left half of singleword arithmetic register AR, designated by the R-field, is shifted arithmetically either right or left and the result is entered into the left half of register AR.

Operation Code	C1
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate
Symbolic Notation	$(ARlh)_j \rightarrow ARlh_{j-sc}$ (Arith.)

The direction and the amount of shift is determined in exactly the same manner as described in the SA instruction.

Programming Notes: The 7-bit shift count is used for all word sizes. If the resulting shift count should exceed the word size of 16-bits for half-words, then the register result would appear as described under programming notes of the SA instruction.

Also, the most significant bit of the T-field is not used, i.e., indirect shift counts are not possible. The M-field must be zero.

Result Code Setting: The result code (RL, RG, RE) is set according to the arithmetic value of the result as follows:

$(AR_{1h}) < 0$ (1, 0, 0)

$(AR_{1h}) > 0$ (0, 1, 0)

$(AR_{1h}) = 0$ (0, 0, 1)

Program Interruption: Fixed point overflow is detected, for arithmetic left shifts only, if the sign bit changes during the shift. The entire shift operation designated by the shift count is completed regardless of overflow conditions.

ARITHMETIC SHIFT DOUBLEWORD (SAD)

The contents of the doubleword arithmetic register ARD, designated by the even R-field value, is shifted arithmetically either right or left and the result is entered into doubleword register ARD.

Operation Code	C3
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate
Symbolic Notation	$(ARD)_j \rightarrow ARD_j\text{-sc}$ (Arith.)

The direction and the amount of shift is determined in the same manner as described in the SA instruction.

Programming Notes: The shift count (SC) cannot exceed the doubleword size of 64-bits. Also, the most significant bit of the T-field is not used, i.e., indirect shift counts are not possible. The M-field must be zero.

Result Code Setting: The result code (RL, RG, RE) is set according to the arithmetic value of the doubleword result as follows:

$(ARD) < 0$	(1, 0, 0)
$(ARD) > 0$	(0, 1, 0)
$(ARD) = 0$	(0, 0, 1)

Program Interruption: Fixed point overflow is detected, for arithmetic left shifts only, if the sign bit changes during the shift. The entire shift operation designated by the shift count is completed regardless of overflow conditions. Specification error if R-field is odd.

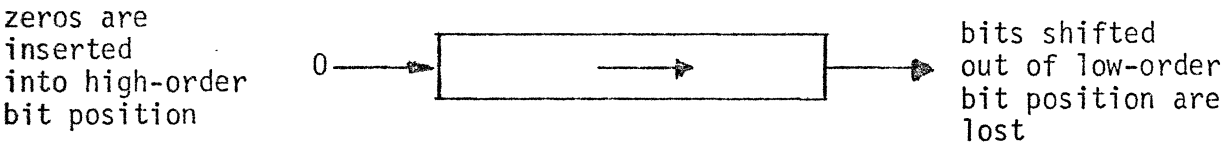
LOGICAL SHIFT WORD (SL)

The contents of the single-word arithmetic register AR, designated by the R-field, is shifted logically and the result is entered into arithmetic register AR.

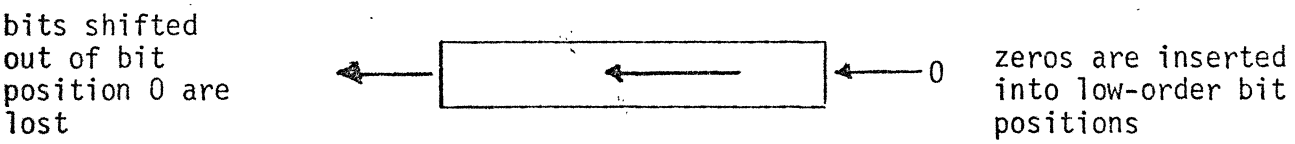
Operation Code	C4
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate
Symbolic Notation	$(AR)_j \longrightarrow AR_{j\text{-}sc}$ (logical)

The direction and the amount of shift is determined by the shift count (SC) information in the 7 LSB's of the N-field of the instruction word. The direction and amount of shift may be modified by the index register specified by the X-field when X ≠ 0. When modification occurs, MN and (X) are added the same as for halfword immediate operands. The least significant 7 bits of the result are interpreted as a twos complement number to determine the direction and amount of shift. This is equivalent to 7 bit addition. Overflow of the 7 bit shift count is not detected. For example, a minus 60 indexed by a minus 7 gives a plus 61; plus 60 indexed by plus 6 gives a minus 62. Bit position 25 of the N-field and of the contents of X is interpreted as the sign position for shift instructions only. A positive sign of the resulting shift count causes a left shift of SC bit positions. A negative sign of the resulting shift count causes a right shift of SC bit positions. The value of SC is within the range: $-64 \leq SC \leq +63$.

Logical right shift (negative shift count)



Logical left shift (positive shift count)



Logical left shifts are the same as arithmetic left shifts, except that overflows are not detected.

Programming Notes: The 7-bit shift count is used for all word sizes. If the resulting shift count should exceed the word size of 32-bits for singleword shift instructions, then the register result would appear as follows:

Logical left shift - all zeros and no overflow detection.

Logical right shift - all zeros.

Also, the most significant bit of the T-field is not used, i.e., indirect shift counts are not possible. The M-field must be zero.

Result Code Setting: The result code (RL, RG, RE) is set according to the logical properties of the singleword result as follows:

Mixed "ones" and "zeros"	(1, 0, 0)
All bits are "one"	(0, 1, 0)
All bits are "zero"	(0, 0, 1)

Program Interruption: None.

LOGICAL SHIFT HALFWORD (SLH)

Operation Code	C5
Type Format	4
Operand Format	R,I,X
Type Addressing	Immediate
Symbolic Notation	$(ARlh)_j \rightarrow ARlh_{j-sc}$ (logical)

The contents of the left half of singleword arithmetic register AR, designated by the R-field, is shifted logically and the result is entered into the left half of register AR. The direction and the amount of shift is determined in the same manner as described in the SL instruction.

Programming Notes: The 7-bit shift count is used for all word sizes. If the resulting shift count should exceed the word size of 16-bits for half-words, then the register result would appear as follows:

Logical left shift - all zeros and no overflow detection.

Logical right shift - all zeros.

Also, the most significant bit of the T-field is not used, i.e., indirect shift counts are not possible. The M-field must be zero.

Result Code Setting: The result code (RL, RG, RE) is set according to the logical properties of the halfword result as follows:

Mixed "ones" and "zeros"	(1, 0, 0)
All bits are "one"	(0, 1, 0)
All bits are "zero"	(0, 0, 1)

Program Interruption: None.

LOGICAL SHIFT DOUBLEWORD (SLD)

Operation Code	C7
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate
Symbolic Notation	$(ARD)_j \rightarrow ARD_{j\text{-}sc}$ (logical)

The contents of the doubleword arithmetic register ARD, designated by the even R-field value, is shifted logically and the result is entered into doubleword register ARD. The direction and the amount of shift is determined in the same manner as described in the SL instruction.

Programming Notes: The shift count (SC) cannot exceed the doubleword size of 64-bits, although SC may be set to minus 64 to get an all zeros result. Also, the most significant bit of the T-field is not used, i.e., indirect shift counts are not possible. The M-field must be zero.

Result Code Setting: The result code (RL, RG, RE) is set according to the logical properties of the doubleword result as follows:

Mixed "ones" and "zeros"	(1, 0, 0)
All bits are "one"	(0, 1, 0)
All bits are "zero"	(0, 0, 1)

Program Interruption: Specification error if R-field is odd.

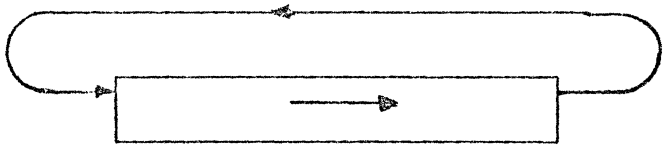
CIRCULAR SHIFT WORD (SC)

The contents of single-word arithmetic register AR, designated by the R-field, is shifted circularly and the result is entered into register AR.

Operation Code	CC
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate
Symbolic Notation	$(AR)_j \longrightarrow AR_{(j-sc)}$ Mod 32

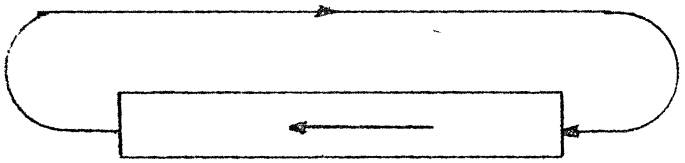
The direction and the amount of shift is determined by the shift count (SC) information in the 7 LSB's of the N-field of the instruction word. The direction and amount of shift may be modified by the index register specified by the X-field when $X \neq 0$. When modification occurs, MN and (X) are added the same as for halfword immediate operands. The least significant 7 bits of the result are interpreted as a twos complement number to determine the direction and amount of shift. This is equivalent to 7 bit addition. Overflow of the 7 bit shift count is not detected. For example, a minus 60 indexed by a minus 7 gives a plus 61; plus 60 indexed by plus 6 gives a minus 62. Bit position 25 of the N-field and of the contents of X is interpreted as the sign position for shift instructions only. A positive sign of the resulting shift count causes a left shift of SC bit positions. A negative sign of the resulting shift count causes a right shift of SC bit positions. The value of SC is within the range: $-64 \leq SC \leq +63$.

Circular right shift (negative shift count)



Bits shifted out of low-order bit position are entered into high-order bit position.

Circular left shift (positive shift count)



Bits shifted out of high-order bit position are entered into the low-order bit position. Overflows are not detected.

Programming Notes: The 7-bit shift count is used for all word sizes. If the resulting shift count should exceed the word size of 32-bits for singleword shift instructions, then the register result would appear as follows:

Circular right shift - Actual right shift equals shift count plus 32. (SC is modulo 32).

Circular left shift - Actual left shift equals shift count minus 32. (SC is modulo 32).

Also, the most significant bit of the T-field is not used, i.e., indirect shift counts are not possible. The M-field must be zero.

Result Code Setting: The result code (RL, RG, RE) is set according to the logical properties of the singleword result as follows:

Mixed "ones" and "zeros"	(1, 0, 0)
All bits are "one"	(0, 1, 0)
All bits are "zero"	(0, 0, 1)

Program Interruption: None.

CIRCULAR SHIFT HALFWORD (SCH)

The contents of the left half of singleword arithmetic register AR, designated by the R-field, is shifted circularly and the result is entered into the left half of register AR. The direction and the amount of shift is determined in the same manner as described in the SC instruction.

Operation Code	CD
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate
Symbolic Notation	$(AR1h) \xrightarrow[j]{} AR1h\ (j\text{-}sc)$

Programming Notes: The 7-bit shift count is used for all word sizes. If the resulting shift count should exceed the word size of 16-bits for halfwords, then the register result would appear as follows:

Circular right shift - Actual right shift equals shift count plus nearest smaller multiple of 16 which brings SC into the range $-16 \leq SC \leq 0$ (SC is modulo 16).

Circular left shift - Actual left shift equals shift count minus nearest smaller multiple of 16 which brings SC into the range $0 \leq SC \leq 15$ (SC is modulo 16)

Also, the most significant bit of the T-field is not used, i.e., indirect shift counts are not possible. The M-field must be zero.

Result Code Setting: The result code (RL, RG, RE) is set according to the logical properties of the halfword result as follows:

Mixed "ones" and "zeros"	(1, 0, 0)
All bits are "one"	(0, 1, 0)
All bits are "zero"	(0, 0, 1)

Program Interruption: None.

CIRCULAR SHIFT DOUBLEWORD (SCD)

Operation Code	CF
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate
Symbolic Notation	$(ARD)_j \longrightarrow ARD_{(j-sc) \text{ Mod } 64}$

The contents of the doubleword arithmetic register ARD, designated by the even R-field value, is shifted circularly and the result is entered into doubleword register ARD. The direction and the amount of shift is determined in the same manner as described in the SC instruction.

Programming Notes: The shift count (SC) cannot exceed the doubleword size of 64-bits. The original value results if SC is zero or minus 64. Also, the most significant bit of the T-field is not used, i.e., indirect shift counts are not possible. The M-field must be zero.

Result Code Setting: The result code (RL, RG, RE) is set according to the logical properties of the doubleword result as follows:

Mixed "ones" and "zeros"	(1, 0, 0)
All bits are "one"	(0, 1, 0)
All bits are "zero"	(0, 0, 1)

Program Interruption: Specification error if R-field is odd.

BIT REVERSAL WORD (RVS)

Operation Code	C6
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate
Symbolic Notation	See Below

Reverse the right most n bits of the contents of arithmetic register AR. The other bits of (AR) remain unchanged. n is equal to IMMED + (X) and is restricted to the range $0 \leq n \leq -32$, where n is a 2's complement number.
Let m = -n
For m = (2,3,4,...32)

$(R)_{j+31-m} \longrightarrow R_{32-j}$ for j = (1,2,3,...n)

$(R)_k$ for k = (0,1,2,...31-m) remain unchanged.

For m = 0 or 1, the contents of R remain unchanged.

For example, suppose n = -7

Before REV Reg. R

	1234567

After REV Reg. R

unchanged	7654321

Result Code Setting: The result code (RL, RG, RE) is set according to the logical properties of the singleword result as follows:

Mixed "ones" and "zeros"	(1, 0, 0)
All bits are "one"	(0, 1, 0)
All bits are "zero"	(0, 0, 1)

Program Interruption: None.

COMPARE INSTRUCTIONS

COMPARE WORD (C)

Operation Code	C8, CE
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	α, singleword
Symbolic Notation	(R) : (α)

OP Code C8

This compare instruction tests the contents of singlword arithmetic register AR relative to the contents of location α and preserves the result of the comparison in the compare code bits (CL, CG, CE) as specified below. The contents of AR and α remain unchanged.

COMPARE Code Setting: (CL, CG, CE)

(AR) < (α)	(1, 0, 0)
(AR) > (α)	(0, 1, 0)
(AR) = (α)	(0, 0, 1)

OP Code CE

This instruction compares the contents of the index register XR or vector register VR, designated by R,relative to the contents of the fixed point single length operand in location α and preserves the result of the comparison in the compare code bits shown below. The contents of XR or VR and α remain unchanged.

XR for R-field range 0 thru 7
VR for R-field range 8 thru F

Compare Code Setting: (CL, CG, CE)

(R) < (α)	(1, 0, 0)
(R) > (α)	(0, 1, 0)
(R) = (α)	(0, 0, 1)

Result Code: Not Affected

Program Interruption: None

COMPARE WORD IMMEDIATE (CI)

OP CODE D8

This compare immediate singleword instruction tests the contents of arithmetic register AR relative to a single length arithmetic immediate operand and preserves the result of the comparison in the compare code bits. The contents of AR remain unchanged.

Operation Code	D8, DE
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate, arith, whole word
Symbolic Notation	(R):n

Compare Code Setting: (CL, CG, CE)

(AR) < IMMED	(1, 0, 0)
(AR) > IMMED	(0, 1, 0)
(AR) = IMMED	(0, 0, 1)

OP CODE DE

This compare immediate singleword instruction tests the contents of index register XR or vector register VR, designated by R, relative to a single length arithmetic immediate operand and preserves the result of the comparison in the compare code bits. The contents of XR and VR remain unchanged.

XR for R-field range 0 thru 7

VR for R-field range 8 thru F

Compare Code Setting: (CL, CG, CE)

(R) < IMMED	(1, 0, 0)
(R) > IMMED	(0, 1, 0)
(R) = IMMED	(0, 0, 1)

Result Code: Not Affected

Program Interruption: None

COMPARE HALFWORD (CH)

Operation Code	C9
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	αh, halfword
Symbolic Notation	(ARlh):(αh)

The compare halfword instruction tests the contents of the 16 most significant bits of arithmetic register AR relative to the contents of halfword location αh, and preserves the result of the comparison in compare code bits as specified below. The contents of AR and α remain unchanged.

Compare Code Setting: (CL, CG, CE)

(ARlh) < (αh)	(1, 0, 0)
(ARlh) > (αh)	(0, 1, 0)
(ARlh) = (αh)	(0, 0, 1)

Result Code: Not Affected

Program Interruption: None

COMPARE HALFWORD IMMEDIATE (CIH)

Operation Code	D9
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate halfword
Symbolic Notation	(AR1h):nrh

A compare immediate halfword instruction tests the most significant 16 bits of arithmetic register AR relative to a halfword immediate operand and preserves the result of the comparison in the compare code bits. The contents of register AR remains unchanged.

Compare Code Setting: (CL, CG, CE)

(AR1h) < nrh	(1, 0, 0)
(AR1h) > nrh	(0, 1, 0)
(AR1h) = nrh	(0, 0, 1)

Result Code: Not Affected

Program Interruption: None

COMPARE FLOATING POINT WORD (CF)

Operation Code	CA
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	α, singleword
Symbolic Notation	(AR):(α)

The compare floating point instruction tests the contents of the singleword arithmetic register AR relative to the contents of location α and preserves the result of the comparison in the compare code bits as specified below. The contents of AR and α remain unchanged.

PROGRAMMING NOTE: Floating point input arguments must be hexadecimally normalized prior to use in a floating point compare instruction.

Compare Code Setting: (CL, CG, CE)

(AR) < (α)	(1, 0, 0)
(AR) > (α)	(0, 1, 0)
(AR) = (α)	(0, 0, 1)

Result Code: Not Affected

Program Interruption: None

COMPARE FLOATING POINT DOUBLEWORD (CFD)

Operation Code	CB
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	αd, doubleword
Symbolic Notation	(ARD):(αd)

The compare instruction tests the contents of the doubleword arithmetic register ARD relative to the contents of location αd and preserves the result of the comparison in the compare code bits as specified below. The contents of ARD and αd remain unchanged.

PROGRAMMING NOTE: Floating point input arguments must be hexadecimally normalized prior to use in a floating point compare instruction.

Compare Code Setting: (CL, CG, CE)

(ARD) < (αd)	(1, 0, 0)
(ARD) > (αd)	(0, 1, 0)
(ARD) = (αd)	(0, 0, 1)

Result Code: Not Affected

Program Interruption: Specification error if R-field is odd.

COMPARE LOGICAL AND (CAND)

Operation Code	E2
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	α, singleword
Symbolic Notation	$(AR)_j \wedge (\alpha)_j$ for j range 0 thru 31

This whole word logical compare instruction first performs a logical "AND" operation on the contents of register AR and the contents of location α. The compare code bits are set according to the logical properties of the 32-bit result, but the result is not stored.

Compare Code Setting: (CL, CG, CE)

Mixed "ones" and "zeros"	(1, 0, 0)
All bits are "one"	(0, 1, 0)
All bits are "zero"	(0, 0, 1)

Result Code: Not Affected

Program Interruption: None

COMPARE LOGICAL "AND" IMMEDIATE (CANDI)

Operation Code	F2
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate, logical SW
Symbolic Notation	$(AR)_j \wedge n_j$ for j range 0 thru 31

This logical immediate instruction first performs a logical "AND" operation on the singleword contents of register AR and the singleword logical immediate operand. The compare code is set according to the logical properties of the 32-bit result, but the result is not stored. The contents of register AR remain unchanged by this instruction.

Compare Code Setting: (CL, CG, CE)

Mixed "1's" and "0's"	(1, 0, 0)
All bits are "1"	(0, 1, 0)
All bits are "0"	(0, 0, 1)

Result Code: Not Affected

Program Interruption: None

COMPARE LOGICAL OR (COR)

Operation Code	E6
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	α, singleword
Symbolic Notation	$(AR)_j \vee (\alpha)_j$ for j range 0 thru 31

This logical compare instruction first performs a logical "OR" operation on the contents of register AR and the contents of location α. The compare code bits are set according to the logical properties of the 32-bit result, but the result is not stored.

The logical properties and the respective compare codes are listed below.

Compare Code Setting: (CL, CG, CE)

Mixed "ones" and "zeros"	(1, 0, 0)
All bits are "one"	(0, 1, 0)
All bits are "zero"	(0, 0, 1)

Result Code: Not Affected

Program Interruption: None

COMPARE LOGICAL "OR" IMMEDIATE (CORI)

Operation Code	F6
Type Format	4
Operand Format	R, I, X
Type Addressing	Immediate, logical SW
Symbolic Notation	$(AR)_j \vee n_j$ for j range 0 thru 31

This logical immediate instruction first performs a logical "OR" operation on the contents of singleword register AR and the singleword logical immediate operand. The compare code is set according to the logical properties of the 32-bit result, but the result is not stored.

The contents of register AR remain unchanged by logical compare instructions.

Compare Code Setting: (CL, CG, CE)

Mixed "1's" and "0's"	(1, 0, 0)
All bits are "1"	(0, 1, 0)
All bits are "0"	(0, 0, 1)

Result Code: Not Affected

Program Interruption: None

COMPARE LOGICAL "AND" DOUBLEWORD (CANDD)

Operation Code	E3
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	αd, doubleword
Symbolic Notation	$(ARD)_j \wedge (\alpha d)_j$ for j range 0 thru 63

Doubleword logical compare instructions first perform the specified logical operation on the contents of doubleword register ARD and the contents of doubleword location αd. The compare code bits are set according to the logical properties of the 64-bit result, but the result is not stored. The logical properties and the respective compare codes are listed below.

Compare Code Setting: (CL, CG, CE)

Mixed "1's" and "0's"	(1, 0, 0)
All bits are "1"	(0, 1, 0)
All bits are "0"	(0, 0, 1)

Result Code: Not Affected

Program Interruption: Specification error if R-field is odd.

COMPARE LOGICAL "OR" DOUBLE WORD (CORD)

Operation Code	E7
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	αd, doubleword
Symbolic Notation	$(ARD)_j \vee (\alpha d)_j$ for j range 0 thru 63

Doubleword logical compare instructions first perform the specified logical operation on the contents of doubleword register ARD and the contents of doubleword location αd. The compare code bits are set according to the logical properties of the 64-bit result, but the result is not stored. The logical properties and the respective compare codes are listed below.

Compare Code Setting: (CL, CG, CE)

Mixed "1's" and "0's"	(1, 0, 0)
All bits are "1"	(0, 1, 0)
All bits are "0"	(0, 0, 1)

Result Code: Not Affected

Program Interruption: Specification error if R-field is odd.

CONDITIONAL BRANCH INSTRUCTIONS

BRANCH ON COMPARISON TRUE (BCC)

Operation Code	91
Type Format	9
Operand Format	M,@ = N, X
Type Addressing	β, branch
Symbolic Notation	β → PC if COND

The R-field of the instruction word is matched with the compare code indicators and a branch is taken to location β if the logical equation, COND = $r_1 \cdot CL + r_2 \cdot CG + r_3 \cdot CE$, is true, otherwise the next instruction in sequence is taken. Terms CL, CG, & CE in the logical equation are the compare code indicators, while r_1, r_2, & r_3 are the 3 LSB's of the R-field. The most significant bit of the R-field is ignored. The instruction mnemonic, R-field value, and branch condition are shown below for the case of a Branch on Comparison True instruction operating on the compare code setting of a previous Arithmetic Comparison instruction. These instructions include C, CT, CH, CIH, CF, CFD.

Mnemonic	R-field	Branch on Comparison Condition
NOP	X000	take next instruction
BE	X001	(R) = (α)
BG	X010	(R) > (α)
BGE	X011	(R) $\geq$ (α)
BL	X100	(R) < (α)
BLE	X101	(R) $\leq$ (α)
BNE	X110	(R) $\neq$ (α)
B	X111	unconditional branch

Compare Code: Not Affected

Result Code: Not Affected

Program Interruption: None

The branch address, β, for a Branch on Condition True instruction is a function of the T, M, and N-fields of the instruction word as follows:

T	M	Branch Address, β	
0	0	N*+(PC)	Relative to program counter
1-7	0	N*+(PC)+(T)	Relative to program counter plus index
0	1-F	N+(M)	Base plus displacement
1-7	1-F	N+(M)+(T)	Base plus displacement plus index
8	0	(N*+(PC))	Indirect relative to program counter
9-F	0	(N*+(PC)+(T-8))	Indirect relative to program counter plus index
8	1-F	(N+(M))	Indirect relative to base plus displacement
9-F	1-F	(N+(M)+(T-8))	Indirect relative to base plus displacement plus index

where N + (M) is Base address plus displacement (N is positive, 12-bit number) and N* ≡ Signed N-field, 11-bits plus sign bit, 2's complement.

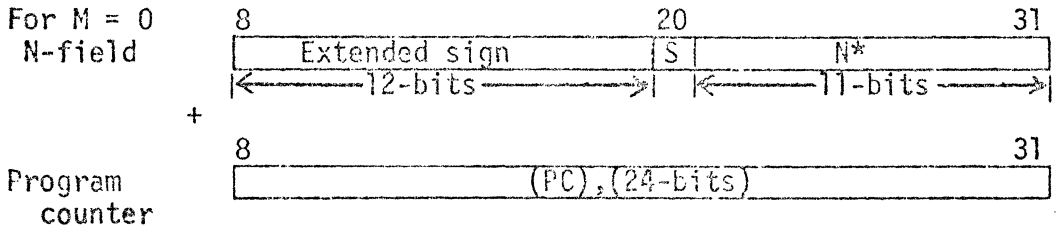
This branch address definition is used for all test and branch instructions.

These include:

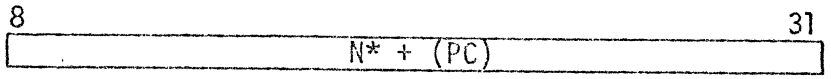
BE, BG, BGE, BL, BLE, BNE, B
BCZ, BCO, BCNM, BCM, BCNO, BCNZ
BZ, BPL, BZP, BMI, BZM, BNZ
BRZ, BRO, BRNM, BRM, BRNO, BRNZ
BU, BO, BUO, BX, BXU, BXO, BXUO, BD
BDU, BDO, BDUO, BDX, BDXU, BDXO, BDXUO
BXEC, BLB, BLX
IBZ, IBNZ, DBZ, DBNZ

When an indirect branch address is specified ($T > 8$), the indirect address format is the same as that used by indirect addressing, except that addresses ≤2F reference central memory regardless of M.
If a branch address is less than or equal to 2F ($\beta \leq 2F$), then the program branches to central memory location regardless of the M and T-field specifications. Branches cannot reference the register file.

Relative branch addresses are generated as follows:



Branch address, β



BRANCH ON COMPARISON AFTER LOGICAL COMPARISON INSTRUCTIONS (BCC)

The Branch on Comparison instruction described on the previous page also functions as a logical test to determine the outcome of a previous Logical Comparison instruction. These instructions include: CAND, COR, CANDD, CORD, CANDI, CORI.

Operation Code	91
Type Format	9
Operand Format	M, @ = N, X
Type Addressing	β, branch
Symbolic Notation	β → PC if COND

Mnemonic	R-field	Branch condition
NOP	X000	Do not branch
BCZ	X001	All bits are zero
BCO	X010	All bits are one
BCNM	X011	Not mixed
BCM	X100	Mixed zeros & ones
BCNO	X101	Not all ones
BCNZ	X110	Not all zeros
B	X111	Unconditional branch

Compare Code: Not Affected

Result Code: Not Affected

Program Interruption: None

BRANCH ON RESULT CODE TRUE (BRC)

Operation Code	95
Type Format	9
Operand Format	M, @ = N, X
Type Addressing	β, branch
Symbolic Notation	β → PC if R COND

The R-field of the instruction word is matched with the result code indicators and a branch is taken to location β if the logical equation, COND = $r_1 \cdot RL + r_2 \cdot RG + r_3 \cdot RE$, is true, otherwise the next instruction in sequence is taken. Terms RL, RG, & RE in the logical equation are the result code indicators, while r_1, r_2, & r_3 are the 3 LSB's of the R-field. The most significant bit of the R-field is ignored. The instruction mnemonic, R-field value, and branch condition are shown below for the case of a Branch on Result Code True instruction operating on the result code setting of a previous Load, Store, or Arithmetic instruction.

Mnemonic	R-field	Branch on REsult Code Condition
NOP	X000	take next instruction
BZ	X001	(R) = 0
BPL	X010	(R) > 0
BZP	X011	(R) $\geq$ 0
BMI	X100	(R) < 0
BZM	X101	(R) $\leq$ 0
BNZ	X110	(R) $\neq$ 0
B	X111	unconditional branch

Compare Code: Not Affected

Result Code: Not Affected

Program Interruption: None

BRANCH ON RESULT CODE INSTRUCTION AFTER A LOGICAL INSTRUCTION (BLR)

Operation Code	95
Type Format	9
Operand Format	M, @ = N, X
Type Addressing	β, branch
Symbolic Notation	β → PC if R COND

The branch on Result Code instruction described on the previous page also functions as a logical test to determine the outcome of a previous logical instruction. The result code setting is determined by the current logical properties of the most recently referenced register, providing that register was referenced by a logical instruction.

Mnemonic	R-field	Branch condition
NOP	X000	Do not branch
BRZ	X001	All bits are zero
BRO	X010	All bits are one
BRNM	X011	Not mixed
BRM	X100	Mixed zeros & ones
BRNO	X101	Not all ones
BRNZ	X110	Not all zeros
B	X111	Unconditional branch

Compare Code: The indicator code settings are not affected by any of the branch instructions just described.

Result Code: Not Affected

Program Interruption: None

BRANCH ON ARITHMETIC EXCEPTION (BAE)

Operation Code	9D
Type Format	9
Operand Format	M, @ = N, X
Type Addressing	β, branch
Symbolic Notation	β → PC if AE COND

The R-field of the instruction word is compared with the arithmetic exception code and a branch is taken to location β when the logical equation,

$$BRANCH = r_0 \cdot D + r_1 \cdot X + r_2 \cdot \emptyset + r_3 \cdot U$$

is true; otherwise the next instruction in sequence is taken. Terms D, X, Ø, and U in the logical equation are the arithmetic exception code bits, while r_0, r_1, r_2, and r_3 represent the bits of the R-field. The table below shows the branch conditions for a BAE - - instruction. The branch address, β, for a BAE - - instruction is defined identical to that of a Branch on comparison True instruction.

The arithmetic exception bits are set when the condition occurs and the bit so set will remain set until it is tested by a BAE - - instruction. Only the tested bit/s as indicated by "ones" in the R-field are reset upon execution of the AE test instruction. Bits not tested are not reset. Thus the AE bits (D, X, Ø, and U) are cumulative in indicating arithmetic exceptions.

Mnemonic	R-field	Branch on Condition
BU	0001	Floating point exp. underflow
BO	0010	Floating point exp. overflow
BUO	0011	Floating point exp. underflow or overflow
BX	0100	Fixed point overflow
BXU	0101	Fixed point overflow or floating point exp. underflow
BXO	0110	Fixed point overflow or floating point exp. overflow
BXUO	0111	Fixed point overflow or floating point exp. underflow or overflow
BD	1000	Divide check
BDU	1001	Divide check or floating point exp. underflow
BDO	1010	Divide check or floating point exp. overflow
BDUO	1011	Divide check or floating point exp. underflow or overflow

BDX	1100	Divide check or fixed point overflow
BDXU	1101	Divide check or fixed point overflow or floating point exp. underflow
BDXO	1110	Divide check or fixed point overflow or floating point exp. overflow
BDXUO	1111	Divide check or fixed point overflow or floating point exponent underflow or overflow

Compare Code: Not Affected

Result Code: Not Affected

Program Interruption: None

BRANCH ON EXECUTE CONDITION (BXEC)

Operation Code	9C
Type Format	3
Operand Format	@ N, X
Type Addressing	β, branch
Symbolic Notation	β → PC if BRANCH true

The R-field of the instruction word is compared with the Branch or Skip register (BSR) and a branch is taken to location β when the logical equation,

$$BRANCH = r_0 \cdot BSR_0 + r_1 \cdot BSR_1 + r_2 \cdot MCC + r_3 \cdot BSC,$$

is true; otherwise the next instruction in sequence is taken.

The BSC term in the logical equation is the Branch or Skip Condition bit. The MCC term is the Monitor Call Condition bit. Terms r_0, r_1, r_2, and r_3 represent the four bits of the R-field.

The BSC bit is set to a "one" when an Execute instruction executes any conditional branch or skip type instruction and the condition for branching or skipping is satisfied. The MCC bit is set to a "one" when an Execute instruction executes an MCP or MCW instruction. The branch or skip is not taken when BSC is set nor is a monitor call made to central memory and the PPU when the MCC bit is set.

If a BXEC instruction (one for which R=0001) branches, then the condition for branching was satisfied. If a BXEC instruction (one for which R=0010) branches, then an Execute instruction has executed an MCP or MCW instruction. Both conditions are tested by a BXEC instruction with an R-field of 0011.

The indicator bits of the BSR register which correspond to the position of "ones" in the R-field of the BXEC instruction are reset to "zero" by the BXEC instruction. Bit positions of BSR which are not tested by "ones" in R are not reset by the BXEC instruction. Only the two LSB's of the BSR register are used by the BXEC instruction. The two MSB bits of the 4-bit BSR register are spare indicator bits which are presently tied to "zero" and are unassigned.

COMPARE CODE: Not affected

RESULT CODE: Not affected

PROGRAM INTERRUPTION: None

INCREMENT, TEST AND BRANCH ON ZERO (IBZ)

Operation Code	88, 8C
Type Format	7
Operand Format	R, @ = N, X
Type Addressing	β, branch
Symbolic Notation	β → PC if = 0

OP Code 88

The contents of the arithmetic register specified by the R-field is incremented by unity and tested for zero. If the contents of register AR equal zero after modification, then a branch is taken to the branch address, β. If (AR) is non-zero the next instruction in sequence is taken.

	Test Condition after modification	
Modification	(AR) = 0	(AR) ≠ 0
(AR) + 1 → AR	β → PC	(PC) + 1 → PC

OP Code 8C

The contents of the index register or vector register (XVR) specified by the R-field is incremented by unity and tested for zero. If the contents of register XVR equal zero after modification, then a branch is taken to the branch address, β. If XVR is non-zero, the next instruction in sequence is taken.

XVR is an index register, XR, if the R-field value is 0 thru 7.
XVR is a vector register, VR, if the R-field value is 8 thru F.

	Test Condition after modification	
Modification	(XVR) = 0	(XVR) ≠ 0
(XVR) + 1 → XVR	β → PC	(PC) + 1 → PC

Result Code: (RL, RG, RE) is set according to the arithmetic value of the register after modification as follows:

Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: None

INCREMENT, TEST AND BRANCH ON NON-ZERO (IBNZ)

Operation Code	89, 8D
Type Format	7
Operand Format	R, @ = N, X
Type Addressing	β, branch
Symbolic Notation	β → PC if ≠ 0

OP Code 89

The contents of arithmetic register AR is incremented by unity and tested for non-zero. If (AR) is non-zero after modification, a branch is taken to β. If (AR) is zero, the next instruction is taken.

Test Condition after modification

Modification	(AR) = 0	(AR) ≠ 0
(AR) + 1 → AR	(PC) + 1 → PC	β → PC

OP Code 8D

The contents of the index register or vector register specified by the R-field is incremented by unity and tested for non-zero. If (XVR) is non-zero after modification, a branch is taken to β. If (XVR) is zero, the next instruction is taken.

Test Condition after modification

Modification	(XVR) = 0	(XVR) ≠ 0
(XVR) + 1 → XVR	(PC) + 1 → PC	β → PC

Result Code: (RL, RG, RE) is set according to the arithmetic value of the register after modification as follows:

Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: None

DECREMENT, TEST AND BRANCH ON ZERO (DBZ)

Operation Code	8A, 8E
Type Format	7
Operand Format	R, @ = N, X
Type Addressing	β, branch
Symbolic Notation	β → PC if = 0

OP Code 8A

The contents of the arithmetic register specified by the R-field is decremented by unity and tested for zero. If the contents of register AR equal zero after modification, then a branch is taken to the branch address, β. If (AR) is non-zero the next instruction in sequence is taken.

	Test Condition after modification	
Modification	(AR) = 0	(AR) ≠ 0
(AR) - 1 → AR	β → PC	(PC) + 1 → PC

OP Code 8E

The contents of the index register or vector register specified by the R-field is decremented by unity and tested for zero. If the contents of register XVR equal zero after modification, then a branch is taken to the branch address, β. If (XVR) is non-zero, the next instruction in sequence is taken.

	Test Condition after modification	
Modification	(XVR) = 0	(AR) ≠ 0
(XVR) - 1 → XVR	β → PC	(PC) + 1 → PC

Result Code: The result code (RL, RG, RE) is set according to the arithmetic value of the register after modification as follows:

Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: None

DECREMENT, TEST AND BRANCH ON NON-ZERO (DBNZ)

Operation Code	8B, 8F
Type Format	7
Operand Format	R, @ = N, X
Type Addressing	β, branch
Symbolic Notation	β→PC if ≠ 0

OP Code 8B

The contents of arithmetic register AR is decremented by unity and tested for non-zero. If (AR) is non-zero after modification, a branch is taken to β. If (AR) is zero, the next instruction is taken.

	Test Condition after modification	
Modification	(AR) = 0	(AR) ≠ 0
(AR) - 1 → AR	(PC) + 1 → PC	β → PC

OP Code 8F

The contents of the index register or vector register specified by the R-field is decremented by unity and tested for non-zero. If (XVR) is non-zero after modification, a branch is taken to β. If (XVR) is zero, the next instruction is taken.

	Test Condition after modification	
Modification	(XVR) = 0	(XVR) ≠ 0
(XVR) - 1 XVR	(PC) + 1 → PC	β → PC

Result Code: The result code (RL, RG, RE) is set according to the arithmetic value of the register after modification as follows:

Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: None

INCREMENT, TEST AND SKIP ON EQUAL (ISE)

The contents of arithmetic register AR is incremented by unity and compared relative to the contents of location α. If AR = (α) after (AR) has been modified, then the next instruction is skipped. If (AR) ≠ (α), then the next instruction is taken.

Operation Code	80
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	α, singleword
Symbolic Notation	PC+2→PC if =

	Test Condition after modification	
Modification	(AR) = (α)	(AR) ≠ (α)
(AR) + 1 → AR	(PC) + 2 → PC	(PC) + 1 → PC

Result Code: The result code (RL, RG, RE) is set according to the arithmetic value of the register after modification as follows:

Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: None

INCREMENT, TEST AND SKIP ON NOT EQUAL (ISNE)

The contents of arithmetic register AR is incremented by unity and compared relative to the contents of location α. If (AR) ≠ (α) after (AR) has been modified, then the next instruction is skipped. If (AR) = (α), then the next instruction is taken.

Operation Code	81
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	α, singleword
Symbolic Notation	PC+2→PC if ≠

Test Condition after modification

Modification	(AR) = (α)	(AR) ≠ (α)
(AR) + 1 → AR	(PC) + 1 → PC	(PC) + 2 → PC

Result Code: The result code (RL, RG, RE) is set according to the arithmetic value of the register after modification as follows:

Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: None

DECREMENT, TEST AND SKIP ON EQUAL (DSE)

The contents of arithmetic register AR is decremented by unity and compared relative to the contents of location α. If (AR) = (α) after (AR) has been modified, then the next instruction is skipped. If (AR) ≠ (α), then the next instruction is taken.

Operation Code	82
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	α, singleword
Symbolic Notation	PC+2→PC if =

Test Condition after modification

Modification	(AR) = (α)	(AR) ≠ (α)
(AR) - 1 → AR	(PC) + 2 → PC	(PC) + 1 → PC

Result Code: The result code (RL, RG, RE) is set according to the arithmetic value of the register after modification as follows:

Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: None

DECREMENT, TEST AND SKIP ON NON-EQUAL (DSNE)

The contents of arithmetic register AR is decremented by unity and compared relative to the contents of location α. If (AR) ≠ (α) after (AR) has been modified, then the next instruction is skipped. If (AR) = (α), then the next instruction is taken.

Operation Code	83
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	α, singleword
Symbolic Notation	PC+2→PC if ≠

Test Condition after modification

Modification	(AR) = (α)	(AR) ≠ (α)
(AR) - 1 → AR	(PC) + 1 → PC	(PC) + 2 → PC

Result Code: The result code (RL, RG, RE) is set according to the arithmetic value of the register after modification as follows:

Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: None

TEST AND BRANCH INSTRUCTIONS

BRANCH ON LESS THAN OR EQUAL TO (BCLE)

Operation Code	84, 86
Type Format	6
Operand Format	R, R, N
Type Addressing	β, branch
Symbolic Notation	See table below

OP Code 84 - Arithmetic Register

OP Code 86 - Index Register

The contents of the arithmetic or index register specified by the R-field is added to the contents of the arithmetic register specified by the T-field. The sum is stored into the arithmetic or index register specified by the R-field. This result is then compared relative to the contents of the arithmetic register specified by the T-field plus one. A branch to location β is taken if the result is less than or equal to the contents of arithmetic register T plus one. The T-field must be even. The increment and limit must be stored into an even-odd arithmetic register address pair.

OP Coade 84

Modification	(AR) ≤ (AT+1)	(AR) > (AT+1)	
(AR) + (AT) →ARR	β → PC	(PC) + 1 → PC	Test Condition after modification

OP Code 86

Modification	(XVR) ≤ (AT+1)	(XVR) > (AT+1)	
(XVR) + (AT) → XVR	β → PC	(PC) + 1 → PC	Test Condition after modification

The branch address, β, is relative to the program counter or relative to the base address depending on the M-field selection as shown in the table below. Indexed branch addressing is not possible. Also, indirect branch addressing is not possible.

M-field		Branch address, β
0	N* + (PC)	Relative to program counter
1-F	N + (M)	Relative to base address

where N is a positive, 12-bit number
and N* is a signed, 2's complement number (11-bits plus sign bit).

Result Code: The result code (RL, RG, RE) is set according to the arithmetic value of the register after modification as follows:

Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: Specification error if T-field is odd.

BRANCH ON GREATER THAN (BCG)

Operation Code	85, 87
Type Format	6
Operand Format	R, R, N
Type Addressing	β, branch
Symbolic Notation	See table below

OP Code 85 - Arithmetic Register

OP Code 87 - Index Register

The contents of the arithmetic or index register specified by the R-field is added to the contents of the arithmetic register specified by the T-field. The sum is stored into the arithmetic or index register specified by the R-field. This result is then compared relative to the contents of the arithmetic register specified by the T-field plus one. A branch to location β is taken if the result is greater than the contents of arithmetic register T plus one. The T-field must be even. The increment and limit must be stored into an even-odd arithmetic register address pair.

Op Code 85

Modification	(AR) ≤ (AT+1)	(AR) > (AT+1)	Test Condition after modification
(AR) + (AT) → AR	(PC) + 1 → PC	β → PC	

Op Code 87

Modification	(XVR) ≤ (AT+1)	(XVR) > (AT+1)	Test Condition after modification
(XVR) + (AT) → XVR	(PC) + 1 → PC	β → PC	

The branch address, β, is relative to the program counter or relative to the base address depending on the M-field selection as shown in the table below. Indexed branch addressing is not possible. Also, indirect branch addressing is not possible.

M-field	Branch address, β	
0	N* + (PC)	Relative to program counter
1-F	N + (M)	Relative to base address

where N is a positive, 12-bit number
and N* is a signed, 2's complement number (11-bits plus sign bit).

Result Code: The result code (RL, RG, RE) is set according to the arithmetic value of the register after modification as follows:

Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: Specification error if T-field is odd.

MISCELLANEOUS INSTRUCTIONS

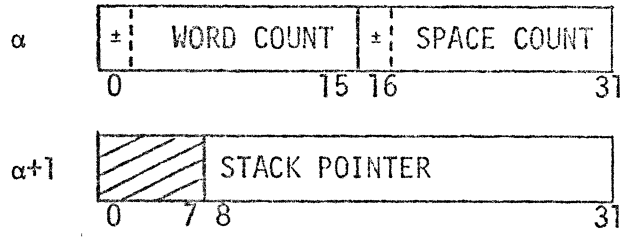
PUSH WORD INTO LIFO STACK (PSH)

Operation Code	93
Type Format	2
Operand Format	R, @ N, X
Type Addressing	αd, doubleword
Symbolic Notation	(AR) → (P) (P)+1 → P

The R-field designates the arithmetic register to be stored in a push operation.

The effective address, αd, specifies a doubleword location. The first word contains a positive 16-bit word count and a positive 16-bit space count in the left and right half word, respectively. Word and space counts are in 2's complement representation. The second word contains a 24-bit stack pointer as shown in the figure below.

Location



The doubleword locations shown above are restricted to an even - odd singleword address pair

In a push word instruction the first word at location α is read from memory. The space and word counts are tested to see if either would result in a negative value after one is added to the word count and one is subtracted from the space count. If either would become negative, then the operation is terminated, the word and space counts are not updated, nothing is pushed, and the next instruction in sequence is executed. If both the space and word counts are zero or positive after one is added to the word count and one is subtracted from the space count, then the updated word and space counts are stored back into location α and the stack pointer word is read from singleword location α+1. The stack pointer is a 24-bit central memory address designating the location into which the contents of register AR is then stored. The stack pointer is incremented by one and stored back into location α+1. The next instruction is executed from the location specified by the program counter plus two (skip). No overflow is indicated. Incrementing of the stack pointer is carried out using full 32-bit addition.

Result Code: Not Affected

Program Interruption: None

PULL WORD FROM LIFO STACK (PUL)

Operation Code	97
Type Format	2
Operand Format	R, @ N, X
Type Addressing	αd, doubleword
Symbolic Notation	(P)-1 → P ((P)) → AR

The R-field specifies the arithmetic register to be loaded in a pull operation.

In a pull word instruction the word from location α is read. The space and word counts are tested to see if either would result in a negative value after one is subtracted from the word count and one is added to the space count. If either would become negative, then the operation is terminated, the word and space counts are not updated, nothing is pulled, and the next instruction in sequence is executed. If both the space and word counts are zero or positive after one is subtracted from the word count and one is added to the space count, then the updated word and space counts are stored back into location α and the stack pointer is read from singleword location α+1. The stack pointer is decremented by one before being used to specify a 24-bit central memory address of an operand which is then read from memory and loaded into register AR. The decremented stack pointer is then stored back into location α+1. The next instruction is executed from the location specified by the program counter plus two (skip). No overflow is indicated. Decrementing of the stack pointer is carried out using 32-bits of the AU adder.

Result Code: Not Affected

Program Interruption: None

MODIFY WORD PAIR (MOD)

Operation Code	9F
Type Format	2
Operand Format	R, @ N, X
Type Addressing	αd, doubleword
Symbolic Notation	$(P)+(AR_{1h})\rightarrow P$

Modify operates on the word-pair described in PUSH. The contents of the left half of the arithmetic register specified by the R-field is added to the word count and subtracted from the space count. Execution is forbidden if the result in the word or space count goes negative. In this case, the operation terminates and the next successive instruction is executed. If both word and space counts result in values which are non-negative, the modified word and space counts replace the original values in central memory, the halfword arithmetic register value is added to the pointer value and the modified pointer is stored in memory and the next sequential instruction is skipped. No overflow is indicated. Modification of the stack pointer is carried out using full 32-bit addition.

If the halfword arithmetic register value is negative (2's complement), the most recent stack entries are deleted. If the halfword arithmetic register value is positive, a gap of unused stack locations is created.

Result Code: Not Affected

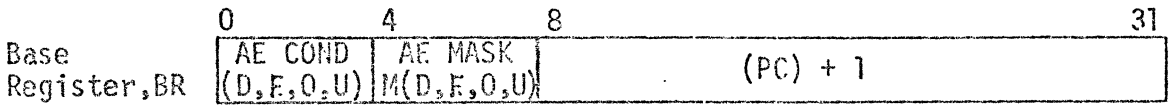
Program Interruption: None

BRANCH AND LOAD BASE REGISTER WITH PC (BLB)

Operation Code	98
Type Format	7
Operand Format	R, @ = N,X
Type Addressing	ß, branch
Symbolic Notation	(PC)+1 → BR
	ß → PC

The contents of the program counter plus one is stored into the base register specified by the R-field and then an unconditional branch is taken to the branch address, ß. The branch address is defined the same as for the branch on comparison instructions. Also, the Arithmetic Exception Condition bits (D, F, O, U) are stored into bit positions 0 thru 3 of base register BR and the Arithmetic Exception Mask bits (MD, MF, MO, MU) are stored into bit positions 4 thru 7 of base register BR.

Programming note: (BR) appears as shown below after the BLB instruction.



Result Code: Not Affected

Program Interruption: None

BRANCH AND LOAD INDEX OR VECTOR REGISTER (BLX)

Operation Code	99
Type Format	7
Operand Format	R, @ = N, X
Type Addressing	β, branch
Symbolic Notation	(PC)+1 → XVR
	β → PC

The contents of the program counter plus one is stored into the index register or vector register specified by the R-field and then an unconditional branch is taken to the branch address, β. The branch address is defined the same as for the branch on comparison instructions. Also, the Arithmetic Exception Condition bits (D, F, O, U) are stored into bit positions 0 thru 3 and the Arithmetic Exception Mask bits (MD, MF, MO, MU) are stored into bit positions 4 thru 7 of index or vector register (XR or VR).

Programming note: (XVR) appears as shown in the diagram for the BLB instruction on the preceeding page, with the exception of replacing BR with XR or VR depending upon the value of the R-field.

XVR is XR if the R-field value is 0 thru 7
XVR is VR if the R-field value is 8 thru F

Result Code: Not Affected

Program Interruption: None

LOAD EFFECTIVE ADDRESS (LEA)

Operation Code	56, 52
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	α, singleword
Symbolic Notation	Op Code 56 α → XR or VR
	Op Code 52 α → BR

OP Code 56

Load the effective address generated by this instruction into the index or vector register (XR or VR), designated by the R-field. The effective address contains 24-bits and is entered into bit positions 8 thru 31 of XR or VR. The eight MSB's (0 thru 7) are filled with zeros.

OP Code 52

Load the effective address of this instruction into base register BR, designated by the R-field. EA is entered into positions 8 thru 31 of BR. The eight MSB's (0 thru 7) are filled with zeros.

Programming notes: All effective addresses generated by this instruction are singleword addresses. When indirect addressing is specified (T-field greater than or equal to 8), then multi-level indirect addressing is possible and the terminal indirect address will refer to a singleword address (i.e., there is no displacement indexing for halfword or doubleword addresses). The indirect address format follows the normal indirect format. Indirect address requests are tagged as execute requests when transmitted to central memory.

After an LEA instruction for which $\alpha \leq 2F$, there is no way to tell whether the contents of register R contains a central memory address or a register address, except by examining the M-field of the LEA instruction which placed the address into register R.

Result Code: Not Affected

Program Interruption: None

INTERPRET (INT)

Operation Code	92
Type Format	1
Operand Format	R, @ = N, X
Type Addressing	α, singleword
Symbolic Notation	(see below)

This instruction loads the operation code and the R-field of the instruction at location α into arithmetic register AR (even). It also loads the T, M, & N-fields of the instruction at location α into register AR + 1 (odd register location).

$$(\alpha)_{0-11} \rightarrow AR_{20-31} \qquad 0 \rightarrow AR_{0-19}$$

$$(\alpha)_{12-31} \rightarrow AR + 1_{12-31} \qquad 0 \rightarrow AR + 1_{0-11}$$

Result Code: Not Affected

Program Interruption: Specification error if R is odd.

EXECUTE (XEC)

Operation Code	96
Type Format	8
Operand Format	@ = N, X
Type Addressing	α, singleword
Symbolic Notation	(α) → IR

This instruction executes the instruction at location α. The program counter is incremented by one following the execution of this instruction.

If the instruction being executed is a branch or skip type instruction and the condition for branching or skipping is satisfied, the BSC-bit of the BSRregister is set to "one" indicating that the branch would have occurred and the instruction following the XEC is executed. A program branch will not occur.

The memory protection hardware will interpret the request for the instruction at location α as an execute request. Also, any indirect memory requests are tagged as execute requests (this is true regardless of instruction type).

Compare Code: Depends on instruction being executed.

Result Code: Depends on instruction being executed.

Program Interruption: Depends on instruction being executed.

Programming Note: If an XEC instruction executes a Branch on Execute Condition instruction (BXEC), for which the LSB of the R-field is one, the BSC-bit of the BSR register is reset to zero if BSC was previously true (one) and remains a zero if BSC was previously false (zero). If the LSB of the R-field of the BXEC instruction is zero, then the BSC bit remains unchanged.

MONITOR CALL AND PROCEED (MCP)

Operation Code	90
Type Format	5
Operand Format	I, X
Type Addressing	Immediate
Symbolic Notation	n → 07

An MCP instruction stores a single length logical immediate operand into actual memory location 07. The most significant eight bits of location 07 are stored as "zeros" in accord with the logical immediate operand format. The immediate operand can be modified by the contents of index register X when X is not equal to zero.

The Central Processor signals the PPU that it has executed an MCP instruction via the CP/PPU communications area of the CR-file of the PPU, and then the CP proceeds to the next instruction.

RESULT CODE: Not Affected

PROGRAM INTERRUPTION: None

MONITOR CALL AND WAIT (MCW)

Operation Code	94
Type Format	5
Operand Format	I, X
Type Addressing	Immediate
Symbolic Notation	n → 07

An MCW instruction stores a single length logical immediate operand into actual memory location 07. The most significant eight bits of location 07 are stored as "zeros" in accord with the logical immediate operand format. The immediate operand can be modified by the contents of index register X when X is not equal to zero.

The Central Processor signals the PPU that it has executed an MCW instruction via the CP/PPU communications area of the CR-file of the PPU. If the PPU is prepared for context switching, then the CP performs an automatic exchange intermediate operation thereby exchanging the CP program. If the PPU is not prepared for context switching, then the CP halts until such time that the PPU initiates a context switch operation.

RESULT CODE: Not Affected

PROGRAM INTERRUPTION: None

FLOATING TO FIXED POINT CONVERSION INSTRUCTIONS

Floating point wholeword numbers can be converted to whole word or halfword fixed point numbers. Floating point doubleword numbers can be converted to whole word fixed point numbers.

Scalar conversion instructions acquire the floating point whole word operand from arithmetic register AR specified by the R-field of the instruction format. Doubleword floating point operands are read from doubleword register ARD specified by the R-field with the LSB ignored. Register ARD is the even-odd register address pair AR and AR+1.

The scale factor or Q-point is supplied as one of the arguments for the conversion process and is obtained from halfword location a_h. Displacement indexing is applied in the standard manner when addressing halfwords. If M = 0 and $a_h \leq$ 2F.1, the scale factor is read from an absolute halfword register address. The scale factor is a 16-bit signed integer and is represented in 2's complement notation for negative numbers. The scale factor locates the fixed point result relative to the decimal point to the right of the least significant bit (fixed point integer format).

A fixed point wholeword signed integer result is stored into general arithmetic register AR. Register AR+1 remains unchanged. A fixed point halfword signed integer result is stored into the left half of arithmetic register AR. The right half of register AR remains unchanged.

The algorithm for converting from floating to fixed point is as follows:

1) Record the sign of the floating point fraction.

2) Subtract 64_{10} $(40)_{hex}$ from the biased hexadecimal exponent to obtain the unbiased hexadecimal exponent, HE.

3) Multiply HE by 4 (shift left 2 bit positions) to obtain the equivalent binary exponent, BE. BE is 9-bits including sign.

4) Consider the floating point fraction to be aligned such that its MSB (bit position 8) is located in bit position 1 of the fixed point output register.

5) Insert a zero into the sign position (bit position 0) of the fixed point output register.

6) Add 31 to the scale factor and subtract from this sum the value of BE obtained in step 3 above. This gives the number of bit positions that the quantity, G, in the fixed point output register is to be shifted.

   If $H = 31 + SF - BE \geq 0$, then shift G right H bit positions. Insert zeros into the vacated positions at the left end of the fixed point output register. Truncation is possible in this step when shifting right.

   If $H = 31 + SF - BE < 0$, then shift G left H bit positions. Insert zeros into the vacated positions at the right end of the fixed point output register. Overflow is possible on this step when shifting left.

7) If the sign recorded from step 1 was negative, then take the 2's complement of the number in the fixed point output register.

8) Store the entire 32-bit fixed point output register into the whole word arithmetic register specified by the AR-field if the instruction specified a whole word result. Store the least significant 16-bits of the fixed point output register into the left half of arithmetic register AR if the instruction specified a halfword.

Several examples are given in Figure 1 for both whole and half word results. Half word results are chosen from the sixteen LSB's of the fixed point integer. In the example, half length conversions would overflow for any scale factor from -256 to +3. Figure 2 shows the same examples for a negative floating point input. The results in Figure 2 are the two's complement of the results in Figure 1. It should be noted however in Figure 2, the result of two separate conversions with a scale factor difference by one does not of necessity yield identical results shifted one position with respect to each other. This is due to the loss of some significance past the LSB and is related to a round-off problem. Scale factors of 000000000 and 000000001 in Figure 2 demonstrate this fact.

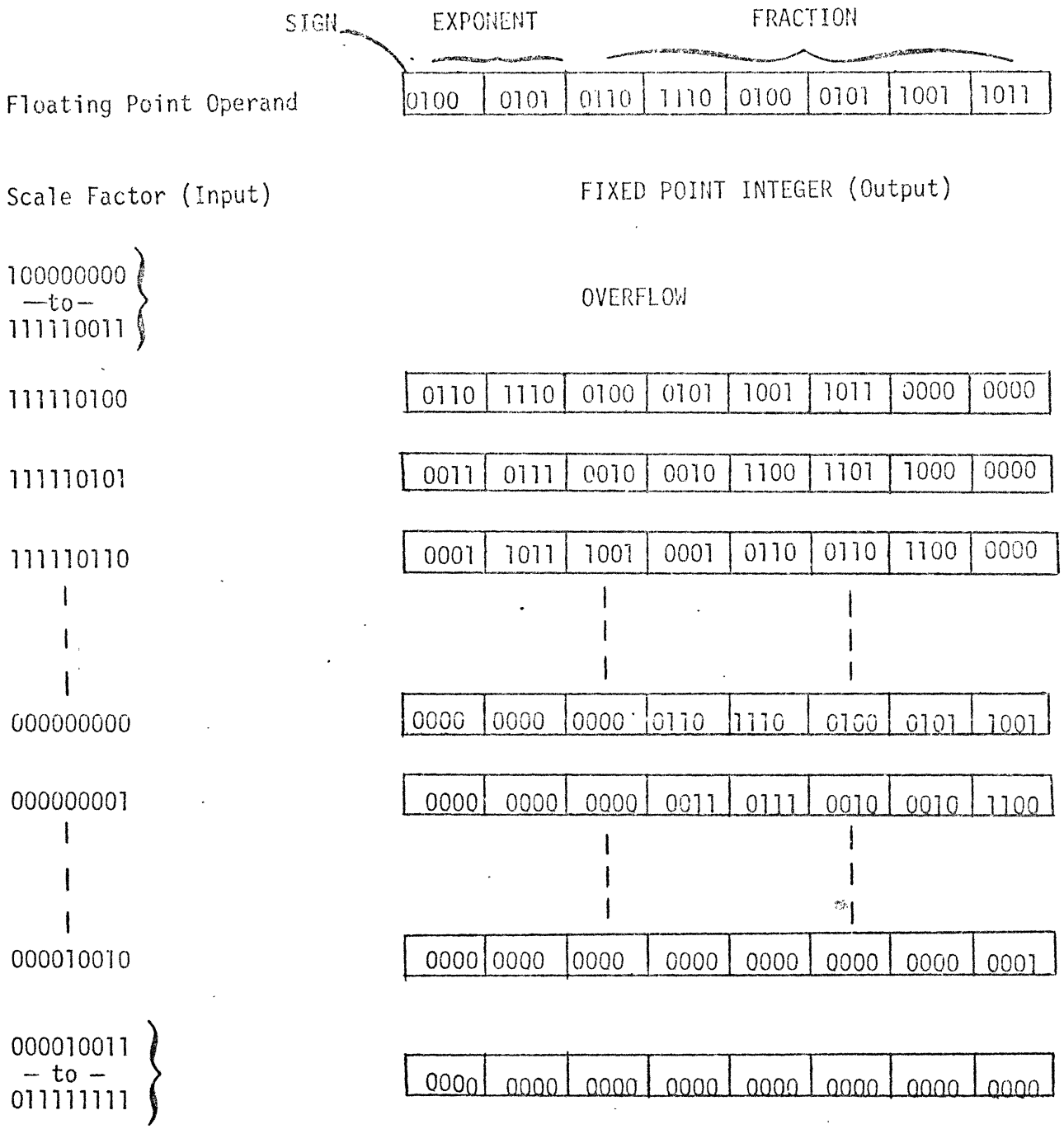

SIGN
EXPONENT
FRACTION
Floating Point Operand
0100 0101 0110 1110 0100 0101 1001 1011
Scale Factor (Input)
FIXED POINT INTEGER (Output)
100000000
—to—
111110011
OVERFLOW
111110100
0110 1110 0100 0101 1001 1011 0000 0000
111110101
0011 0111 0010 0010 1100 1101 1000 0000
111110110
0001 1011 1001 0001 0110 0110 1100 0000
000000000
0000 0000 0000 0110 1110 0100 0101 1001
000000001
0000 0000 0000 0011 0111 0010 0010 1100
000010010
0000 0000 0000 0000 0000 0000 0000 0001
000010011
— to —
011111111
0000 0000 0000 0000 0000 0000 0000 0000

Figure 1

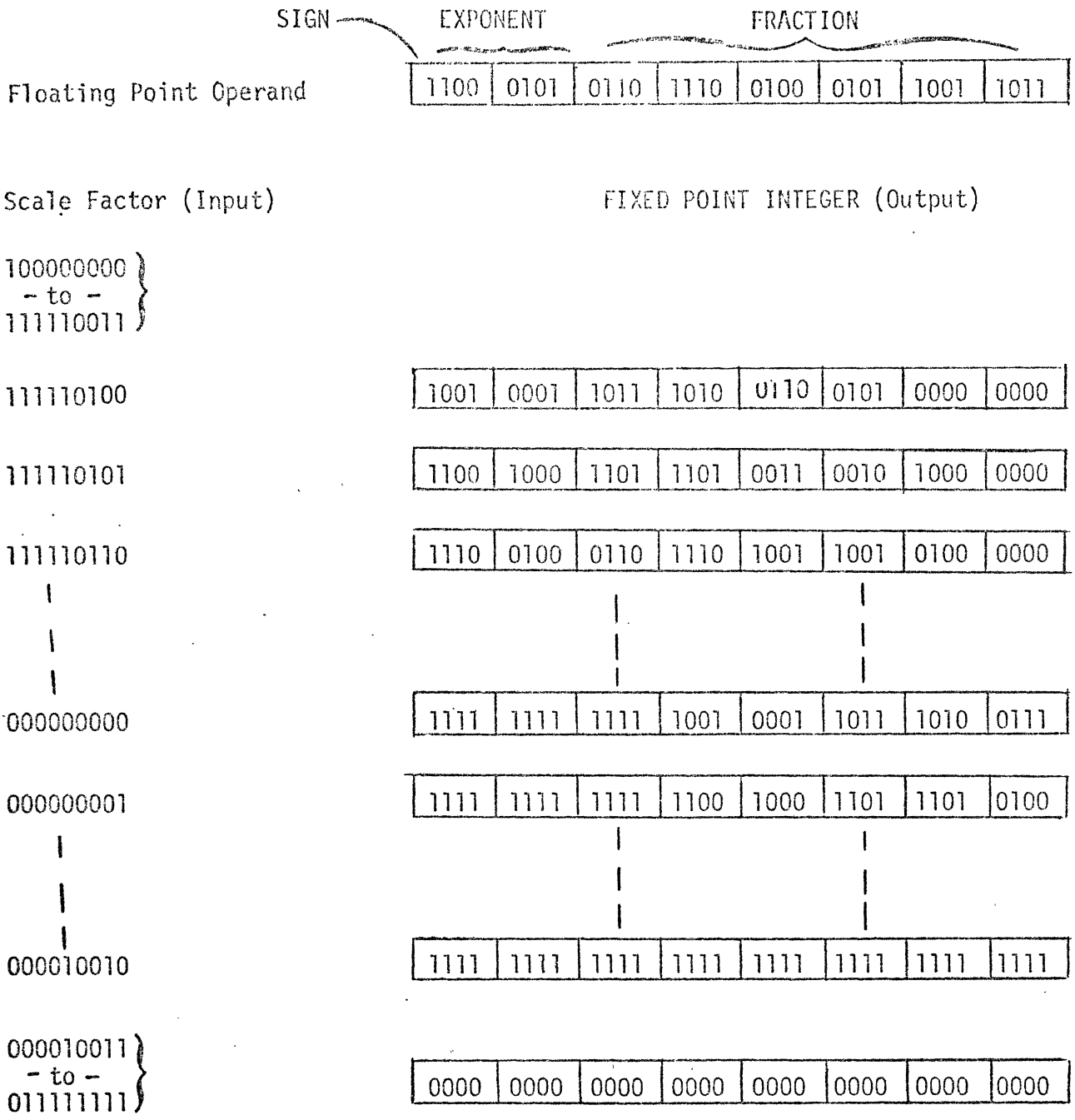

SIGN
EXPONENT
FRACTION
Floating Point Operand
1100 0101 0110 1110 0100 0101 1001 1011
Scale Factor (Input)
FIXED POINT INTEGER (Output)
100000000
- to -
111110011
111110100
1001 0001 1011 1010 0110 0101 0000 0000
111110101
1100 1000 1101 1101 0011 0010 1000 0000
111110110
1110 0100 0110 1110 1001 1001 0100 0000
000000000
1111 1111 1111 1001 0001 1011 1010 0111
000000001
1111 1111 1111 1100 1000 1101 1101 0100
000010010
1111 1111 1111 1111 1111 1111 1111 1111
000010011
- to -
011111111
0000 0000 0000 0000 0000 0000 0000 0000

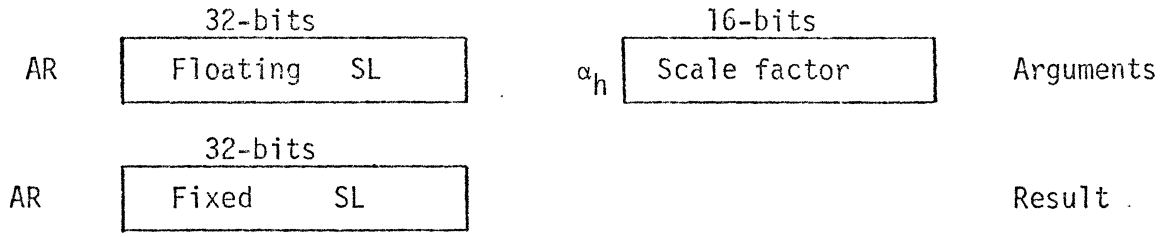
Figure 2

CONVERT FLOATING POINT TO FIXED POINT WORD (FLFX)

Operation Code	A0
Type Format	2
Operand Format	R, @ N, X
Type Addressing	α_h, halfword

The floating whole word operand to be converted is read from arithmetic register AR. The scale factor used in the conversion process is read from halfword address α_h.

After conversion, the fixed point whole word signed integer is stored into arithmetic register AR.



Result Code: The result code (RL, RG, RE) is set according to the arithmetic value of the result in register AR as follows:

Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

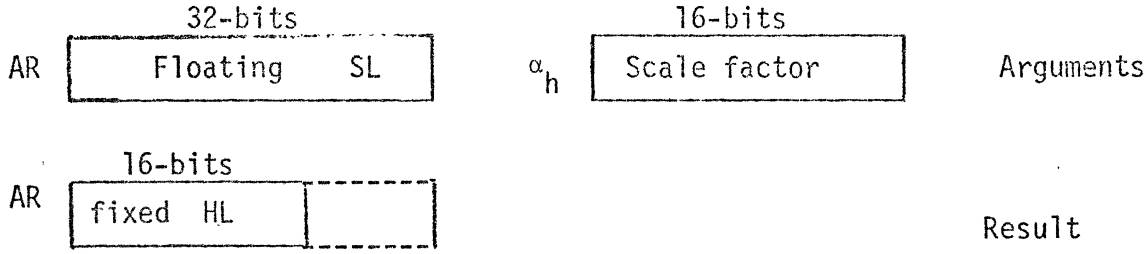
Program Interruption: Fixed point overflow.

CONVERT FLOATING POINT WORD
TO FIXED POINT HALFWORD (FLFH)

Operation Code	A1
Type Format	2
Operand Format	R, @ N, X
Type Addressing	αh, halfword

The floating point whole word operand to be converted is read from register AR. The scale factor used in the conversion process is read from halfword address α_h.

After conversion, the fixed point halfword signed integer result is stored into the left half of arithmetic register AR.



Result Code: The result code (RL, RG, RE) is set according to the arithmetic value of the result in the left half of register AR as follows:

Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

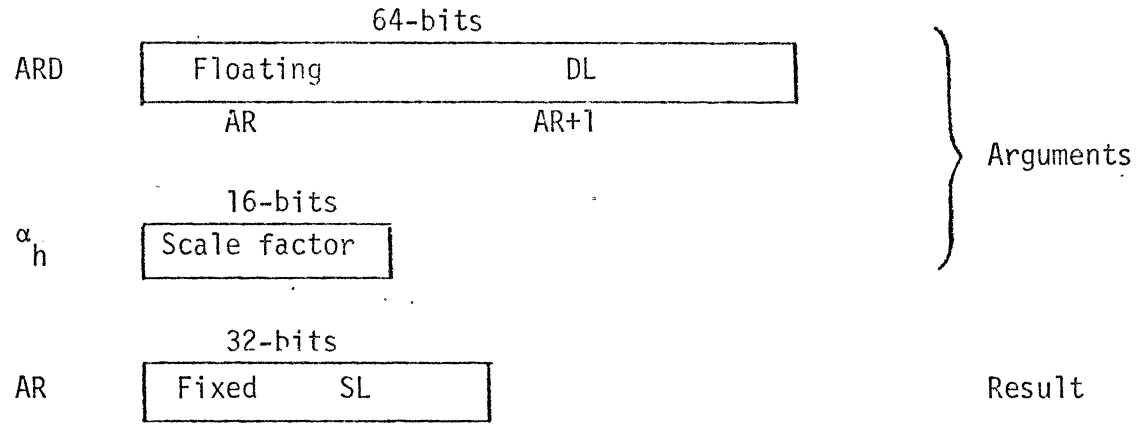
Program Interruption: Fixed point overflow.

CONVERT FLOATING POINT DOUBLEWORD TO FIXED POINT SINGLE WORD (FDFX)

Operation Code	A2
Type Format	2
Operand Format	R, @ N, X
Type Addressing	αh, halfword

The floating point doubleword operand to be converted is read from doubleword register ARD. The scale factor used in the conversion process is read from halfword address α_h.

After conversion, the fixed point whole word signed integer result is stored into arithmetic register AR.



Result Code: The result code (RL, RG, RE) is set according to the arithmetic value of the result in register AR as follows:

Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: Fixed point overflow. Specification error if R-field is odd.

FIXED TO FLOATING POINT CONVERSION INSTRUCTIONS

Fixed point whole or halfword numbers can be converted to whole or doubleword floating point numbers.

Scalar conversion instructions acquire the fixed point whole word signed integer operand from arithmetic register AR. Halfword fixed point signed integer operands are read from the left half of arithmetic register AR.

The scale factor or Q-point is supplied as one of the arguments for the conversion process and is obtained from halfword location α_h. Displacement indexing is applied in the standard manner when addressing halfwords. The scale factor is a 16-bit signed integer and is represented in 2's complement notation for negative numbers. The scale factor for these conversions establish the appropriate exponent for the floating point number. The unbiased hexadecimal exponent range is $-64 \leq$ HEX. EXP. $\leq +63$. Floating point exponent overflow and underflow will be detected if a scale factor results in a hexadecimal exponent outside the range following the conversion operation.

A floating point whole word result is stored into arithmetic register AR. A doubleword floating point result is stored into the doubleword register ARD. ARD is the even-odd register address pair AR and AR + 1.

The algorithm for converting fixed point to floating point number representation is as follows:

1) Determine the sign of the fixed point signed integer to be converted. If the sign is negative, take the 2's complement of the fixed point number. The sign information is saved. If the fixed point operand is a halfword, it is entered into the right half of the whole word register leading to the arithmetic unit (AU). The sign of this halfword operand is extended 16-bits into the most significant half of the register leading to the AU. The AU may now operate as though all fixed point operands are whole word (32-bit) signed integers.

2) Move the decimal point from its position to the right of the LSB for integer representation to the left of the MSB of the 32-bit register containing the operand from 1 above. The number is now in fractional representation. This operation is accomplished simply by adding 32 to the scale factor.

3) Perform a floating point normalize operation on the fixed point fraction by shifting the fraction left by a multiple of four bit positions such that the most significant four bits contain at least one "1". Four is subtracted from the scale factor for each multiple of four bit positions that the fraction is shifted.

4) The fraction is shifted left the number of bit positions specified by the two LSB's of the scale factor (0 = (00), 1 = (01), 2 = (10), 3 = (11), providing that this shift would not cause overflow. If an overflow condition exists, then the fraction is shifted right by an amount equal to zero for (00), one for (11), two for (10), and three for (01) and the scale factor is incremented by four.

5) The scale factor is shifted right two bit positions (equivalent to division by four) and then addition of the shifted scale factor and +64 = (1,000,000) is taken modulo 128. The result is placed in the biased hexadecimal exponent position of the floating point output.

6) The fraction is entered into the floating point fraction position of the output and the sign information saved in step 1 is placed in the MSB position. The floating point number is stored into register AR if a whole word and into doubleword register ARD if the result is specified to be a doubleword.

Figures 3 and 4 demonstrate the types of result to be obtained through use of a fixed to floating instruction. Halfword operands will be inserted into the sixteen LSB's with the sign bit extended into the sixteen MSB's. If the number to be converted is negative, a two's complement operation is performed in the AU before proceeding with alignment. This insures that the floating point magnitude will be the same if two fixed point numbers which are the two's complement of each other are converted by a fixed to floating instruction. Detection of exponent overflow or underflow will set the corresponding Arithmetic Exception Code.

FIXED POINT INTEGER	0010	1101	0001	1110	0110	0101	0111	0100

SCALE FACTOR (INPUT)	FLOATING POINT RESULT EXPONENT							
111100000	0100	0000	0010	1101	0001	1110	0110	0101
111100001	0100	0000	0101	1010	0011	1100	1100	1010
111100010	0100	0000	1011	0100	0111	1001	1001	0100
111100011	0100	0001	0001	0110	1000	1111	0011	0010
111100100	0100	0001	0010	1101	0001	1110	0110	0101

Figure 3

FIXED POINT INTEGER	1111	1111	1111	1001	0111	0100	1000	0101

SCALE FACTOR (INPUT)	FLOATING POINT RESULT EXPONENT							
111100000	1011	1101	0110	1000	1011	0111	1011	0000
111100001	1011	1101	1101	0001	0110	1111	0110	0000
111100010	1011	1110	0001	1010	0010	1101	1110	1100
111100011	1011	1110	0011	0100	0101	1011	1101	1000
111100100	1011	1110	0110	1000	1011	0111	1011	0000

Figure 4

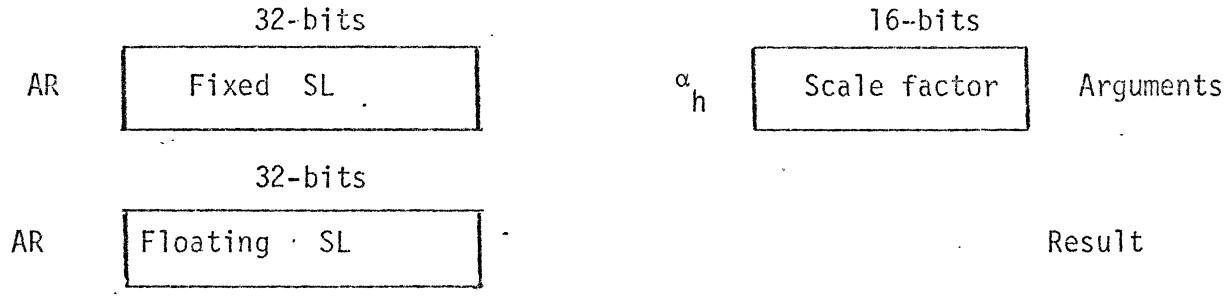
CONVERT FIXED POINT TO
FLOATING POINT WORD (FXFL)

Operation Code	A8
Type Format	2
Operand Format	R, @ N, X
Type Addressing	αh, halfword

The fixed point whole word operand to be converted is read from register AR.

The scale factor, used as an argument in the conversion, is read from halfword address α_h.

After conversion, the normalized floating point whole word result is stored into register AR.



Result Code: The result code (RL, RG, RE) is set according to the arithmetic value of the floating point result in register AR as follows:

Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: Floating point overflow.

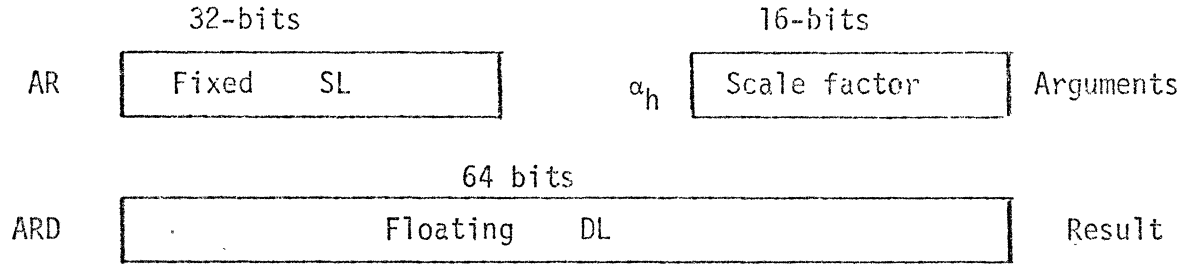
CONVERT FIXED POINT WORD
TO FLOATING POINT DOUBLEWORD (FXFD)

Operation Code	AA
Type Format	2
Operand Format	R, @ N, X
Type Addressing	αh, halfword

The fixed point whole word operand to be converted is read from register AR.

The scale factor, used as an argument in the conversion, is read from halfword address α_h.

After conversion, the normalized floating point doubleword result is stored into doubleword register ARD.



Result Code: The result code (RL, RG, RE) is set according to the arithmetic value of the floating point result in register ARD as follows:

Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: Floating point overflow.
Specification error if R-field is odd.

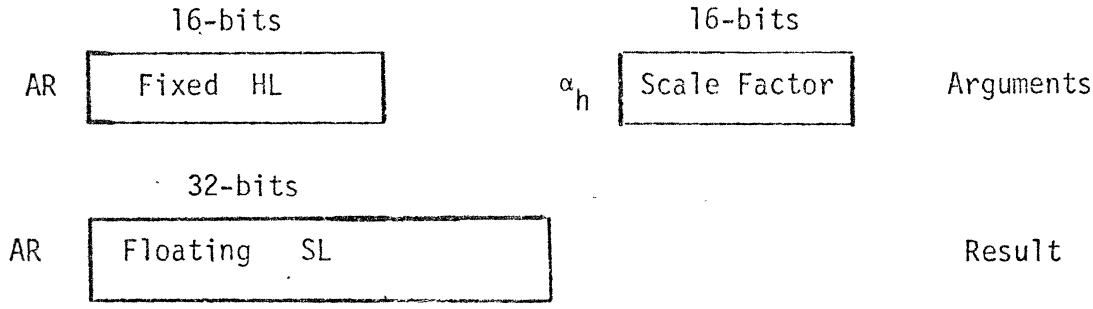
CONVERT FIXED POINT HALFWORD
TO FLOATING POINT WORD (FHFL)

Operation Code	A9
Type Format	2
Operand Format	R, @ N, X
Type Addressing	αh, halfword

The fixed point halfword operand to be converted is read from the left half of register AR.

The scale factor is read from halfword address α_h.

After conversion, the normalized floating point whole word result is stored into register AR.



Result Code: The result code (RL, RG, RE) is set according to the arithmetic value of the floating point result in register AR as follows:

Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: Floating point overflow.

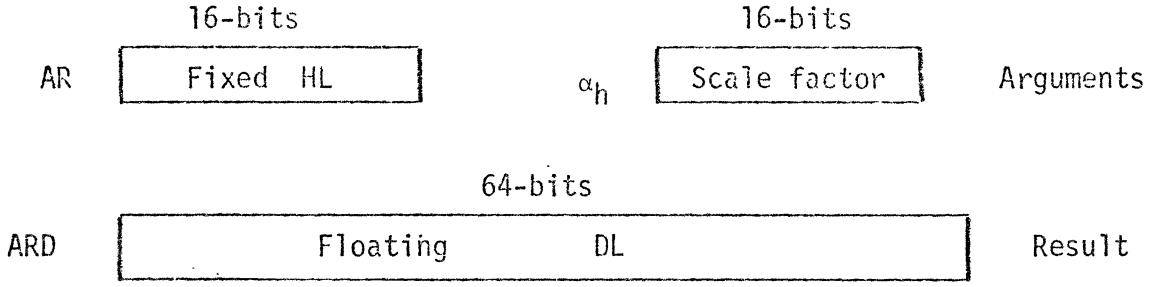
CONVERT FIXED POINT HALFWORD
TO FLOATING POINT DOUBLEWORD (FHFD)

Operation Code	AB
Type Format	2
Operand Format	R, @ N, X
Type Addressing	αh, halfword

The fixed point half length operand to be converted is read from register AR.

The scale factor is read from halfword location α_h.

After conversion, the normalized floating point double length result is stored into double length register ARD.



Result Code: The result code (RL, RG, RE) is set according to the arithmetic value of the floating point result in register ARD as follows:

Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: Floating point overflow.
Specification error if R-field is odd.

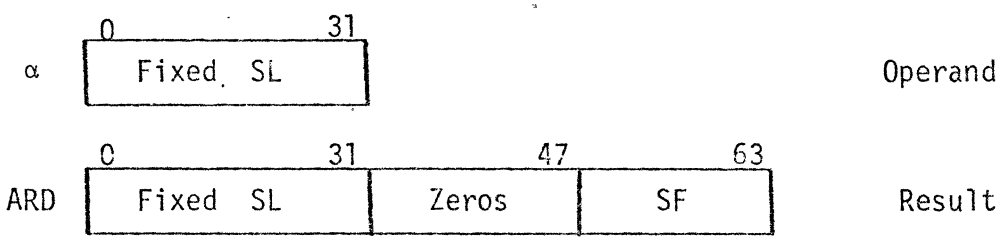
NORMALIZE FIXED POINT WORD (NFX)

Operation Code	AC
Type Format	2
Operand Format	R, @ N, X
Type Addressing	α, singleword

The fixed point whole word number to be normalized is read from location α.

After normalization, the fixed point whole word result is stored into the left half of doubleword register ARD. The scale factor, equal to the number of bit positions that the fixed point number was shifted, is stored into the right quarter of doubleword register ARD. Zeros are entered into bit positions 32 through 47 of (ARD).

The fixed point number has been normalized when the two most significant bit positions differ, (0, 1) or (1, 0). If the fixed point number was initially zero, it is considered normalized and the scale factor (or shift count) is zero. The shift count is stored as a negative number or zero for all normalizations.



Result Code: The result code (RL, RG, RE) is set according to the arithmetic value of the result in the left half of doubleword register ARD as follows:

Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

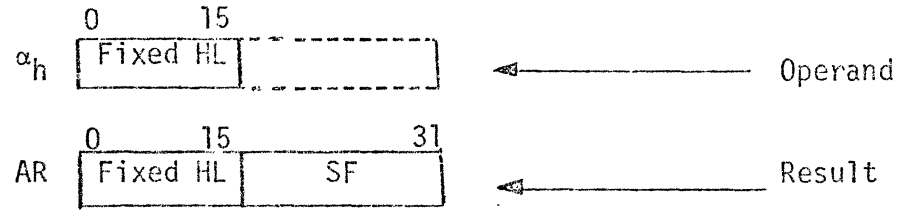
Program Interruption: Specification error if R-field is odd.

NORMALIZE FIXED POINT
HALFWORD (NFH)

Operation Code	AD
Type Format	2
Operand Format	R, @ N, X
Type Addressing	α_h, halfword

The fixed point half length number to be normalized is read from halfword location α_h.

After normalization, the fixed point half length result is stored into the left half of the single length register specified by the R-field. The scale factor is stored into the right half of register AR.



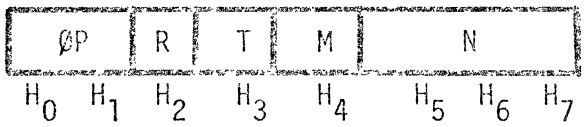
Result Code: The result code (RL, RG, RE) is set according to the arithmetic value of the result in the left half of register AR as follows:

Negative	(1, 0, 0)
Positive	(0, 1, 0)
Zero	(0, 0, 1)

Program Interruption: None.

INSTRUCTION FORMAT

A vector instruction appearing in the instruction fetch unit has the format shown below.



The instruction serves to convey: (1) the fact that this is a vector instruction and, (2) the location of an eight word vector parameter list in central memory. These eight words are loaded into the vector parameter registers assigned to locations 28 through 2F (hexadecimal) and are used for further definition of the vector operation. The vector parameter registers are loaded as part of the vector instruction.

- ØP-field

  The operation code field specified that a vector instruction is to be executed. The exact nature of the vector instruction is contained in the vector parameter file.

- R-field

  The R-field indicates whether information for the vector parameter file (VPF) is read from central memory as part of the vector initialization procedure or retained from a prior setting.

- R-field

  x x x 0 Load VPF from central memory
  x x x 1 Use current parameter file

  If R = 1, the current status of the vector parameter file is retained for the current instruction. A memory cycle to access a new parameter list is not executed under this condition.

- T-field

  The T-field is an address modifier tag that may be decomposed into an indirect addressing bit, I, and a 3-bit index register designator, X. The table below represents the address as a function of the N-field and the T-field.

T	Address type	Virtual Address of VPF
0	Direct address	N_b
1-7	Indexed address	$N_b + (T)$
8	Indirect address	(N_b)
9-F	Indexed indirect	$(N_b + (T-8))$

where $N_b = N + (M)$

Note that when an address refers to an octet of central memory (as for the case when loading the vector parameter file) the three least significant bits of the address are forced to zero after indexing and indirect addressing. This says that a memory octet cannot cross memory word (256-bit) boundaries.

The index registers designated by X are those assigned to register location 21 through 27 (hexadecimal).

- M & N-field

The M-field is a base register designator which selects one of 15 registers assigned to locations 1 through $F_{(H)}$. The N-field displacement is added to the base address when no indexing is used, also when indexing is used, provided $M \neq 0$. The N-field is added to the base address and the index register when indexing is used. Base addressing is not used when $M = 0$.

The VPF address generated for VECT is standard α addressing developed from the T, M, and N fields to specify an octet location of a vector parameter file to be loaded into the VPF registers prior to the execution of the vector instruction. The addressing convention of VECT is identical to that of scalar instructions. Addressing of vectors from the vector parameter file is different and is described on the next few pages.

PARAMETER FILE FORMAT

Register	H_0	H_1	H_2	H_3	H_4	H_5	H_6	H_7
28	ØPR		ALCT	SV	L			
29	-	XA	SAA					
2A	HS	XB	SAB					
2B	VI	XC	SAC					
2C	DAI				DBI			
2D	DCI				NI			
2E	DAØ				DBØ			
2F	DCØ				NØ			

Reg	Hex Character	Field	Description
28	H_0,H_1	ØPR	Operation code
28	H_2	ALCT	Arith. & Log. Comparison Tests
28	H_3	SV	Single-valued vector
28	H_4-H_7	L	Vector dimension
29	H_1	XA	Initial index A
2A	H_1	XB	Initial index B
2B	H_1	XC	Initial index C
29	H_2-H_7	SAA	Starting address A
2A	H_2-H_7	SAB	Starting address B
2B	H_2-H_7	SAC	Starting address C
29	H_0-H_7	(29)	Immediate operand A
2A	H_0-H_7	(2A)	Immediate operand B
2A	H_0	HS	Halfword starting addr.
2B	H_0	VI	Vector increment direction
2C	H_0-H_3	DAI	$\pm \Delta A_i$, inner loop
2C	H_4-H_7	DBI	$\pm \Delta B_i$, inner loop
2D	H_0-H_3	DCI	$\pm \Delta C_i$, inner loop
2D	H_4-H_7	NI	Inner loop count
2E	H_0-H_3	DAØ	$\pm \Delta A_0$, outer loop
2E	H_4-H_7	DBØ	$\pm \Delta B_0$, outer loop
2F	H_0-H_3	DCØ	$\pm \Delta C_0$, outer loop
2F	H_4-H_7	NØ	Outer loop count

ØPR, OPERATION CODE

ØPR-field - specifies the vector operation to be performed.

SV, INSTRUCTION VARIATIONS

SV-field - specifies the type of addressing for single-valued vectors.

Register 28 Bit Positions 12,13,14,15	Vector A	Vector B
X 0 X X	DV	DV
X 1 0 0	DSVV	DV
X 1 0 1	DV	DSVV
X 1 1 0	ISVV	DV
X 1 1 1	DV	ISVV

where

DV = Directly addressed vector

DSVV = Directly addressed single-valued vector

ISVV = Immediate single-valued vector

Vector A is said to be directly addressed when the starting address for that vector is determined by the contents of the 24-bit address field of register 29. (Bits 8 through 31, labeled SAA). Vector B is directly addressed by the contents of the 24-bit address field (labeled SAB) of register 2A. The XA and XB fields of registers 29 and 2A must be zero for non-indexed direct addressing of vectors A and B.

Indexed direct addressing is implied for the DV and DSVV cases when the XA and XB fields of registers 29 and 2A are non-zero. Any one of the seven index registers in locations 21 through 27 may be used to index the starting address of vector A or B. The index register selected is specified by the XA and XB fields similar to the selection used by the T-field for scalar instructions. In the case of vector addresses, the three LSB's of XA and XB select an index register which is added to the starting address of vectors A and B, respectively. The most significant bit of XA and XB is ignored (i.e., indirect vector starting addresses are not permitted).

Immediate operand A is the 32-bit contents of register 29, when SV = X110. Immediate operand B is the 32-bit contents of register 2A, when SV = X111.

SV-field description:

Bit 12 is used to specify the product length options for vector multiply and vector dot product operations. It also specifies the dividend length for vector divide instructions. See length option table following the vector divide instruction.

A "zero" in bit 13 indicates that the addresses of both A and B input vectors are automatically incremented by the self loop or by the inner and outer loop operations regardless of the value of the other three bits of SV.

A "one" in bit 13 and a "zero" in bit 14 (x10x) indicates the self loop does not increment the input variable specified by bit 15. Incrementing vector address B is disabled when bit 15 = 1. The starting addresses for vectors A and B is indicated by the contents of register 29 and 2A in the standard manner. Inner and outer loops incrementing occurs in the normal manner when bit 13 = 1 and bit 14 = 0 (x10x). This means that a new directly addressed single-valued vector can be used as a constant argument during the next self-loop following an inner or outer loop by specifying a non-zero delta increment for the input variable specified by bit 15 (designated DSVV in the table on the preceding page).

A "one" in bit 13 and a "one" in bit 14 (X11X) disables vector address incrementing in all loops (self loop, inner and outer loops) for the input variable specified by bit 15. For this case, the contents of vector parameter register 29 or 2A is interpreted as the value of the immediate operand. In addition, the word length of the immediate operand will depend on the word length of the vector instruction as specified by the operation code.

If the vector instruction is a fixed point half length operation, the immediate operand is the contents of bits 16 through 31 of register 29 or 2A (Register 29 if SV = x110 and register 2A if SV = x111).

If the vector instruction is a fixed or floating point single length operation, then bits 0 through 31 are interpreted as the immediate operand. If the vector instruction is a floating point double length operation, the bits 0 through 31 are interpreted as the left half of the double length immediate operand and bits 32 through 63 of the immediate operand are zero. Immediate operands for vector instructions may not be indexed by the contents of the index register specified by XA or XB.

ALCT, ARITHMETIC AND LOGICAL COMPARISON TEST

The ALCT field, located in hexadecimal character H_2 of vector parameter register 28, specified the test criteria which is applied to the results of the vector comparisons to determine which elements of vector A are stored in the output vector. See the instruction descriptions for Vector Arithmetic Comparison and Vector Logical Comparisons for further details on the function of the ALCT field.

L, VECTOR LENGTH

L-field specifies the length of the vector operation (self-loop dimension). The maximum length is $2^{16} - 1 = 65,535$ elementary operations.

XA, XB, AND XC-FIELDS

These fields specify the index register which may be used to modify the starting addresses of the A, B, and C vectors. The index registers specified by the 3 LSB's of the XA, XB, and XC fields are the same as those used by the scalar instructions (registers 21 through 27).

If the vector instruction is a fixed or floating point single length operation, then the contents of index registers XA, XB, and XC specify a singleword displacement which is added to the singleword starting address expressed by SAA, SAB, and SAC. Let the indexed starting addresses for vectors A, B, and C be denoted by IA, IB, and IC. Then, for indexed singleword vector operations:

$$IA = SAA + (XA)$$
$$IB = SAB + (XB)$$
$$IC = SAC + (XC)$$

If XA, XB, or XC are zero, then the starting addresses SAA, SAB, and SAC are not indexed.

HALF LENGTH ELEMENTS AND HS-FIELD

If the vector instruction is a fixed point half length operation, then the contents of index register XA, XB, and XC specifies a halfword memory address increment which is added to the singleword starting address expressed by SAA, SAB, and SAC. For the cases where it is undesirable to use up to three additional index registers to supply a halfword vector address displacement but where it is necessary to start a vector operation from an odd halfword memory address (the right half of a central memory singleword), the HS-field is provided in the most significant hex character (H_0) of register 2A. The least significant three bits (bits 1, 2, and 3) of hex character H_0 of register 2A provide a means of displacing the singleword starting addresses (SAA, SAB, and SAC) by one halfword address. If an index register is not used, then a "one" in bit position 1, 2, and 3 of the HS-field will specify a starting address in the right half of central memory singleword address SAA, SAB, and SAC, respectively, providing, of course, that a half length vector operation is ordered by the OPR-field.

If an index register is not used and a "zero" is in bit position 1, 2, and 3 of the HS-field, then the left half central memory singleword will be selected as the initial operand for vectors A, B, and C, respectively.

If an index register is used, then the even or oddness of the sum of (XA) + HSA, (XB) + HSB will determine whether the left or right half of a central memory word is selected for the initial halfword vector element.

An odd sum refers to the right half word, whereas an even sum refers to the left half word. HSA is in bit position 1 of register 2A (the second most significant bit position), HSB is in bit position 2 and HSC is in bit position 3. For halfword vector operations, the indexed halfword starting addresses of vectors A, B, and C are:

$$IA = SAA \times 2 + (XA) + HSA$$

$$IB = SAB \times 2 + (XB) + HSB$$

$$IC = SAC \times 2 + (XC) + HSC$$

DOUBLE LENGTH ELEMENTS

If the vector instruction is a floating point double length operation, then the contents of index registers XA, XB, and XC specify a doubleword displacement which is added to the singleword starting address expressed by SAA, SAB, and SAC. The least significant bit of SAA, SAB, and SAC is ignored when generating a doubleword vector starting address. Vector doubleword elements must be stored in even-odd central memory address pairs. For doubleword vector operations the indexed doubleword starting addresses of vectors A, B, and C are:

$$IA = \frac{SAA}{2} + (XA)$$

$$IB = \frac{SAB}{2} + (XB)$$

$$IC = \frac{SAC}{2} + (XC)$$

The most significant bit of the HS-Field of the vector parameter file previously was not used.

This bit is used to delete the indices from the output array for the eight Vector Compare (VC) instructions and the four Vector Peak Pick (VPP) instructions.

When the MSB of both the HS-field and the VI-field are "zero", the output array for these vector instructions is as described on page 171 of Section B3 of this Hardware Manual.

When the MSB of the HS-field is "one", only the item counts after each self loop completion are stored. All of the indices are deleted. Delta increments for the inner or outer loop are applied to the three vector addresses after each self loop completion. The delta C increment is applied to the address of the item count, and not to the location of the last index value as it would have been had the indices been stored.

VI, VECTOR SELF-LOOP INDEXING DIRECTION

The VI-field provides additional information for the vector index units. A "one" in bit position 1 (2,3) of register 2B specified that vector address A (B,C) is decremented by unity after each elementary operation of a vector instruction (useful for convolution, etc.). A "zero" causes the normal forward incrementing of vector addresses by unity.

The most significant bit of the VI-field of the vector parameter file is used to delete the item counts (except for the first item count) from the C output vector of Vector Compare (VC) and Vector Peak Pick (VPP) instructions. This applies to all four word size formats of the Vector Arithmetic Comparison and Vector Peak Pick instructions, and to the four Vector Logical Comparison instructions.

When the most significant bit of both the VI-field and the HS-field are "zero", the output array for these vector instructions is as described on page 171 of Section B3 of this Hardware Manual.

When the MSB of the VI-field is "one", the item count stored at the beginning of the output array is the total item count for the complete vector and is equal to the sum of all the item counts for each inner and outer loop that would have been stored had the MSB of the VI-field been"zero". All other item counts, except the one at the beginning of the output arry, are deleted. All that remains, following the beginning total item count, is a list of indices for each element where a comparison true is detected for Vector Compare instructions or where a peak or valley is found for Vector Peak Picking instructions. If inner or inner and outer loops are invoked, then the indices are not reset at the beginning of each new self loop, but continue to increment throughout the entire vector.

If the delta C increment for both inner and outer loops is unity, then a continuous output of indices is stored in consecutive halfword locations for VC and VPP instructions. In the case of VPP instructions, the normal peak isolation which occurs between two succesive self loops when the MSB of the VI-field is "zero" does not exist when this bit is "one". This allows one to set the self loop length to one (L=1) and find peak in a vector by sampling every nth data point, i.e. delta A for the inner loop is equal to n.

In addition to the deletion of indices for Vector Compare and Vector Peak Picking instructions, setting the MSB of the VI-field to "one" in a Vector Dot Product or Vector Search instruction reduces these instructions to one which produces a singular output regardless of the number of inner or outer loops specified. That is, a Vector Dot Product (VDP) produces one scalar output which has as its value the summation of all c_i elements which would have been produced from each self loop of the VDP with inner or inner

and outer loops had the most significant bit of the VI-field been "zero." The delta increments for the A and B input vector are applied at each turn of the loop, but the arithmetic unit does not receive the "end of self loop" signal and consequently does not reset its internal accumulator that is summing the individual products.

For the Vector Search instructions, a single index value results which represents the location of the element meeting the search criteria regardless of the number of inner or outer loops employed. For example, an entire matrix array could be searched for its largest arithmetic element, even in cases where address increments are required to move to the next row or column at inner or outer loop turning points. The index value of the result is a measure of the total number of elements tested up to the one meeting the search criteria.

The index value may overflow its maximum range if the product of the number of self loops times the number of inner loops times the number of outer loops is greater than or equal to 2^{16} for the Vector Search, Compare, and Peak Pick instructions when the MSB of the VI-field is "one." Also, for the Vector Compare and Peak Pick instructions the total item count will overflow if the total number of true comparisons exceeds $2^{16} - 1$.

Consider an example of the VI and HS-field MSB usage with a Vector Compare instruction. Given matrix A which is a 4 by 3 array of elements and B which is a row vector of length 3. Vector B is to be compared with the rows of A for arithmetic equals. The inner loop count is set to 4. Four cases are presented, one for each of the four settings of the VI and HS most significant bits. Item count values are enclosed by a square.

$$[A] = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 0 \\ 1 & 0 & 3 \\ 0 & 2 & 3 \end{bmatrix}$$

$$[B] = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

Vector Parameter File

OP	=	Vector Arithmetic Compare for Equal
ALCT	=	1
SV	=	0
L	=	3
DAI	=	1
DBI	=	-2
DCI	=	1
NI	=	4
DAO	=	0
DBO	=	0
DCO	=	0
NO	=	0

HS_{msb}	VI_{msb}	C Vector Output (Each element represents a halfword)
0	0	[3] 0 1 2 [2] 0 1 [2] 0 2 [2] 1 2
0	1	[9] 0 1 2 3 4 6 8 10 11
1	0	[3 2 2 2]
1	1	[9]

NI AND NØ, LOOP COUNT

NI-field specifes the inner loop count or the number of times that a given vector instruction is to be executed in an inner loop. Loop counting is done internally in the index unit, only the initial inner loop count is supplied by the NI-field. If NI is zero, the self loop routine is executed once, there is no inner loop, and the outer loop count (NØ) is not examined. If NI is one, the self loop routine is executed once, there is no inner loop address modifications, and the outer loop is executed NØ times. If NI is any value other than 0 or 1, the specified vector operation is executed NI times and then the outer loop is executed, this process is repeated until the outer loop counter equals zero.

NØ-field specifies the outer loop count or the number of times that the inner loop routine is to be executed. Here again the loop counting is accomplished by hardware in the index unit and only the initial outer loop count is supplied by the NØ-field. A branch to the outer loop is taken each time the inner loop counter (LPCI) in the index unit reaches zero. The LPCI is reloaded with the value in the NI-field and the outer loop counter (LPCØ) is decremented by one each time the outer loop is taken. The compound vector instruction is completed when the outer loop count in LPCØ reaches zero. If NØ is zero or one, the specified vector operation is executed NI times and then the operation is terminated. If NØ is any value other than zero or one, the specified vector operation is executed NI times and then the outer loop address modifications occur, the inner nested loop is repreated. This process continues until the outer loop counter equals zero. Refer to the flow chart for an illustration of the vector loop procedure.

DAI AND DBI-FIELDS

These fields specify the address increments for vectors A and B following each vector operation during an inner loop. The increments are not added to the addresses SAA and SAB contained in registers 29 and 2A, but instead are added to the address registers IA and IB contained with the MBU. The addition is accomplished using the addition hardware of the MBU. Increments may be positive or negative 2's complement 16-bit numbers.

$$\left.\begin{aligned} IA &\leftarrow (IA) + \Delta A_i \\ IB &\leftarrow (IB) + \Delta B_i \end{aligned}\right\} \text{inner loop}$$

For single length vector operations, the IA and IB addresses are initially equal to SAA + (XA) and SAB + (XB), respectively. In this case, DAI and DAØ are singleword address increments. Inner and outer loop increments are applied to the terminal address of each self loop. This terminal address is equal to the self loop initial address plus L-1.

Inner and outer loop increments for half length vector instructions represent halfword address increments. Inner and outer loop increments for double length vector instructions represent double length address increments. This is similar to the displacement indexing applied to scalar addressing, the increments are shifted right one bit for halfword operations and left one bit for doubleword operations before being added to the terminal self loop address in the index unit.

When the SV-field equals X100, the elements of Vector A remain constant during the self loop (i.e., L times). The constant value K is acquired from the contents of central memory location SAA + (XA) initially.

Then if the inner loop count NI is greater than one, the address IA = SAA + (XA) is incremented by DAI or possibly DAØ (if NØ > 1). The self loop is again exexuted L times with the new value of K. The number of different values of K which may be acquired using both inner and outer loop features is equal to NI times NØ.

When the SV-field equals X'01', the elements of vector B remain constant during the self loop. The constant value K is acquired from the contents of CM location SAB + (XB) initially. If inner and outer loops are specified, then this address is incremented by DBI or DBØ. The self loop is again executed L times with the new value of K. The number of different values of K which may be acquired using both inner and outer loop features is equal to NI time Nø.

Note that the addresses SAA + (XA) and SAB + (XB) represent singleword addresses. See the starting addresses listed under the description of the XA, XB, and XC field for halfword and doubleword starting address definitions.

When the SV-field equals X'10' or X'11', the elements of vector A or B (respectively) remain constant during the self loop, inner loop, and outer loop. The immediate operand from register 29 or 2A is used throughout the entire vector operation, including all inner and outer loop.

DAØ AND DBØ-FIELDS

These fields specify the address increments for vectors A and B following the vector operation for which the inner loop count has reached zero. The increments are added to the address registers IA and IB contained within the MBU. The same statements for half and double length vector operations apply here as they did for the DAI and DBI fields.

$$\left.\begin{aligned} IA &\rightarrow (IA) + \Delta A_0 \\ IB &\leftarrow (IB) + \Delta B_0 \end{aligned}\right\} \text{ outer loop (singleword incrementing shown)}$$

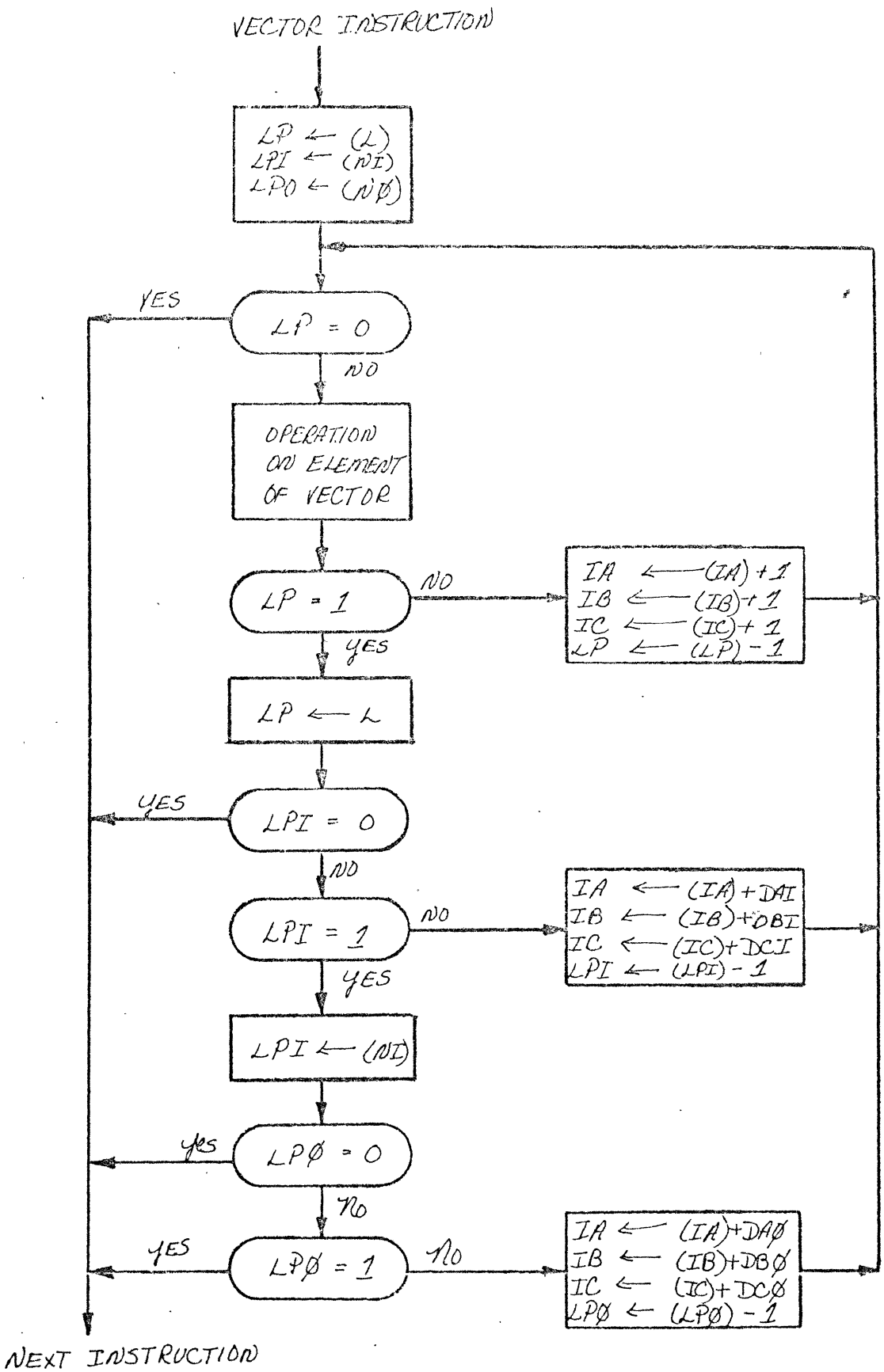
DCI AND DCØ-FIELDS

DCI-field specifies the address increment, ΔC_i, for the output vector address following each vector operation duing an inner loop.

When a vector dot product operation is specified by the ØPR-field, the C address in the index unit is not modified, since only a scalar result is stored. Therefore, if looping is desired the DCI field must indicate an increment for the C address.

DCO-field specifies the address increment, ΔC_0, for the output vector address after each inner loop routine has been completed. DCI and DCØ are shifted right or left one bit position prior to addition in the index unit for halfword or doubleword vector instructions.

An increment K will index a vector address by K halfwords, singleword, or doublewords depending on the word size of the vector operation which is specified. The flow chart below illustrates the procedure for a vector instruction with inner and outer loops.



Vector Inner and Outer
Loop Flow Chart

INSTRUCTION CHARACTERISTICS

The vector instructions are described with the understanding that four different data formats apply to each instruction (except for logical and shift instructions).

DATA FORMATS

a) Fixed point, single length, 32-bit word
b) Fixed point, half length, 16-bit word
c) Floating point, single length, 32-bit word
d) Floating point, double length, 64-bit word

Vector instructions have the following general characteristics:

VECTOR LENGTH

The dimension of the argument vectors $\vec{A}$ and $\vec{B}$ and the result vector $\vec{C}$ is specified by the L-field of the vector parameter registers (H_4-H_7) of register 28). If the L-field equals zero, the vector instruction becomes a no operation. If the L-field equals one, the vector instruction is equivalent to a scalar operation, although the inner and outer loop features may still be incorporated if desired. If the L-field is any value other than zero or one, the vector operation specified by the ØPR-field is executed as described in the vector instruction descriptions. The maximum length is $2^{16}-1$.

ADDRESSES

The elements of vectors $\vec{A}$ and $\vec{B}$ are read from consecutive memory locations. The result vector $\vec{C}$ is stored into consecutive locations. The addresses for the initial vector elements (a_1, b_1, and c_1) are determined from the vector address parameters contained in 29, 2A, and 2B. Vectors A, B, and C cannot address the register file. An address 2 2F references central memory.

SINGLE VALUE VECTORS

An instruction with an SV-field which specifies a single valued vector operation, has as its result a vector $\vec{C}$,
where $\vec{C} = (c_1, c_2, c_3 \ldots\ldots, c_n)$

with c_i = k operation b_i

for SV = X100 or X110

and $c_i = a_i$ operation k

for SV = X101 or X111

SV	k value
X100	Contents of location IA
X101	Contents of location IB
X110	Contents of register 29
X111	Contents of register 2A

The most significant bit of the SV-field is ignored except when it is used to specify the product length for multiply or dividend length for divide operations.

For SV = X100, K is the contents of location IA initially, where IA is equal to:

$IA = SAA + (XA)$ single length operations
$IA = SAA \times 2 + (XA) + HSA$ half length operations
$IA = \frac{SAA}{2} + (XA)$ double length operations

When the inner and outer loop feature is used, the subsequent single-valued operands for the self loops are acquired from central memory locations IA, where IA is modified by the loop increments.

IA ← (IA) + DAI If inner loop
IA ← (IA) + DAØ If outer loop

For SV = X101, K is the contents of location IB initially, where IB is defined similarly.

For SV = X110 or X111, K is the operand contained in register 29 or 2A, respectively. K remains unchanged during all self, inner, and outer loops. For halfword vector instructions, the value K is obtained from bits 16 through 31 of register 29 or 2A. For singleword vector instructions, K is obtained from bits 0 through 31 of register 29 or 2A. For doubleword vector instructions, the most significant half of K is obtained from bits 0 through 31 of register 29 or 2A and the least significant half (bits 32 through 63 of K) is zero.

INNER AND OUTER LOOPS

This instruction write-up only describes the self loop operation. The user can form multiple loops which change the starting address of each pass of the self loop. All vector instructions can use both the inner and outer loops as well as the self loop, except for vector order instructions. Some applications of the inner and outer loop feature are described in Section B4.

DOUBLE LENGTH

Vector double length operations require that the double length operands be stored into even-odd memory or register singleword address pairs.

ARITHMETIC EXCEPTION

When mask bits in the arithmetic exception (AE) register are off, the vector operation will run to normal completion. When masked on, the vector operation terminates when the arithmetic exception condition occurs, such that an "exchange intermediate" can be effected by the PPU.

ARITHMETIC EXCEPTIONS FOR SCALAR OR VECTOR OPERATIONS

	MASKED OFF			MASKED ON
	FLOATING POINT		FIXED POINT	FIXED OR FLOATING POINT
	UNDERFLOW	OVERFLOW	OVERFLOW	FLOATING UNDERFLOW FLOATING OVERFLOW FIXED OVERFLOW DIVIDE CHECK
Data	Set to 0	Set to $\pm \infty$	Modulo word size	Freeze CP and Exchange intermediate.
AE register	Set AE Cond	Set AE Cond	Set AE Cond	Handled by software

DEFINITIONS

+ ∞, floating point

Single length 7FFF FFFF

Double length 7FFF FFFF FFFF FFFF

- ∞, floating point

Single length FFFF FFFF

Double length FFFF FFFF FFFF FFFF

Zero, fixed or floating point

Half length 0000

Single length 0000 0000

Double length 0000 0000 0000 0000

When an arithmetic exception condition occurs, the result code will be set according to the table below.

Arithmetic Exception Condition	Result Code RL	RG	RE	AU Result
Fixed point overflow				
(positive overflow)	1	0	0	minus, modulo word size
(negative underflow)	0	1	0	positive, modulo word size
Floating point exponent overflow				
(positive fraction)	0	1	0	pos. ∞
(negative fraction)	1	0	0	neg. ∞
Floating point exponent underflow (pos. or neg. fraction)	0	0	1	zero
Divide check (fixed point)	Unpredictable			Unpredictable
Divide check (floating point)				
(positive dividend)	0	1	0	pos. ∞
(negative dividend)	1	0	0	neg. ∞

VECTOR HAZARD RULE

Consider the array of octets from which the $\vec{A}$ and $\vec{B}$ vectors are formed for input to the Arithmetic Unit (AU) and the array of octets into which the results are stored from the output of the AU. Define the "present octet address of input vectors $\vec{A}$ or $\vec{B}$" to be the octet addresses of the vector elements a_i and b_i which are presently being processed as inputs to a vector computation. The "present octet address of output vector $\vec{C}$" is defined to be the octet address of result c_i corresponding to the computation involving arguments a_i and b_i. The Vector Hazard Rule is stated as follows:

> A "Hazard Condition" occurs whenever the present octet addresses of input vector $\vec{A}$ or $\vec{B}$ or the next four octet addresses for each of vectors $\vec{A}$ or $\vec{B}$ is the same as the present result octet address or the eight past result octet address of output vector $\vec{C}$.

If the Hazard Rule is violated the "old" rather than the "new (updated) information is used as the operand. For example, a vector operation will use the "old" values for one of the operands if the element address of c_i is one greater than the element address of either a_i or b_i and all vectors are assigned a positive increment direction during the self loop.

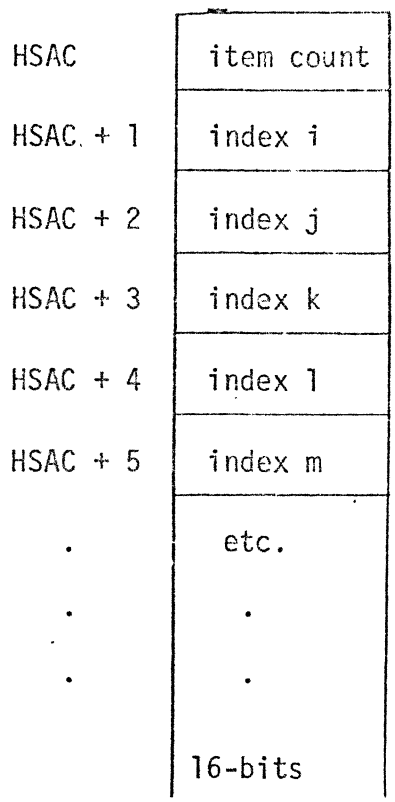
NORMALIZED INPUTS

Floating point inputs must be hexadecimally normalized prior to their use in the following vector instructions:

VAF	Vector Add Floating Poing Single Length
VAFD	Vector Add Floating Point Double Length
VAMF	Vector Magnitude Floating Point Single Length
VAMFD	Vector Magnitude Floating Point Double Length
VSF	Vector Subtract Floating Point Single Length
VSFD	Vector Subtract Floating Point Double Length
VSMF	Vector Subtract Magnitude Floating Point Single Length
VSMFD	Vector Subtract Magnitude Floating Point Double Length
VCF	Vector Arithmetic Compare Floating Point Single Length
VCFD	Vector Arithmetic Compare Floating Point Double Length

INDEX VALUE STORAGE

Format for storage of index values during a Vector Arithmetic Comparison, Vector Logical Comparison, or Vector Peak Picking instruction.



where HSAC= 2·SAC + (XC)+HSC ≡ Halfword starting address of vector $\vec{C}$.

and HSC ≡ Halfword starting address even-odd selection.
(0~even, 1~odd).

HALFWORD ADDRESS INCREMENTS FOR STORAGE OF INDEX VALUES

Increments for vector $\vec{C}$ (during inner and outer loops) are referenced from the last index stored as a result of the previous self loop operation. For example, if index m is the last index stored in a self loop (See diagram above) and the delta increments for the inner or outer loop are one (DCI = 1, DCØ = 1), then the entry point for the item count of the next self loop is stored into halfword location HSAC + 6. The unit of measure for DCI and DCO for these vectors is one halfword for a delta value of one.

ASSEMBLER MNEMONICS

The Assembler furnishes built-in procedures to aid in the building of Vector Parameter Files. These procedures generate data in the sequence of code from which the procedure calls are made. Each of the eight words which make up the Vector Parameter File for each Vector instruction may be generated by a separate procedure call. The user may wish to define an entire vector file by building a procedure which contains eight separate calls, but one will not be finished in the Assembler as a single procedure because of the volume of parameters which would be associated with the call.

Vector data generated by the above method may then be placed in the vector parameter registers by the use of instructions such as VECTL, which loads the file and executes the vector instruction; LF, which loads the vector file; or L which replaces individual vector parameters.

The first word (vector operation word) of the eight is built by calling the procedure whose name is one of the mnemonics defined on the following pages, depending upon the operation. In all cases, the format of the call resembles the scalar instruction format:

```
(label)      command   ALCT, L, SV
```

where ALCT is used only in certain test instructions and should be zero for the other instructions

L is the vector length

SV is the single valued vector addressing type

(label) is optional

Words 2, 3, and 4 are built by calling the procedure VCTRA whose format is:

```
(label)    VCTRA     SA, X, Q
```

where

SA is the starting address of a vector

X is the index used to find the starting address of the vector

Q is the value of the HS or VI field used in words 3 or 4 of a vector instruction

label is optional

If X or Q is left blank, a zero value is assumed.

Words 5-8 are built using the DATAH pseudo-directive.

VECTOR (VECTL)

Operation code	B0 R = 0
Type Format	3
Operand Format	@ N,X
Type Addressing	α, octet
Symbolic Notation (General)	$\vec{A}$ op. $\vec{B} \rightarrow \vec{C}$

A vector instruction appearing in the instruction fetch unit has the format common to most scalar instructions. The instruction serves to convey: (1) the fact that this is a vector instruction, and (2) the location of an eight word vector parameter list in central memory. These eight words are loaded into the vector parameter registers assigned to locations 28 through 2F (hexadecimal) and are used for further definition of the vector operation. The vector registers are loaded as part of the vector initialization procedure prior to the execution of the vector instruction.

The format of VECT appeared under INSTRUCTION FORMAT on page 151.

Result Code: Not useful after vector operation.

Program Interruption: Depends upon vector instruction being executed.

VECTOR (VECT)

Operation code	B0 R = 1
Type Format	3
Operand Format	
Type Addressing	
Symbolic Notation (General)	$\vec{A}$ op. $\vec{B} \rightarrow \vec{C}$

A vector instruction appearing in the instruction fetch unit has the format common to most scalar instructions. The instruction defines the vector operation.

The Vector registers are not loaded as part of this instruction. The current vector registers are used.

The format of VECT appeared under INSTRUCTION FORMAT on page 151.

Result Code: Not useful after vector operation.

Program Interruption: Depends upon vector instruction being executed.

METIC INSTRUCTIONS

ADD

A vector add instruction with argument vector $\vec{A}$ and $\vec{B}$,
where $\vec{A} = (a_1, a_2, a_3, \ldots, a_L)$
and $\vec{B} = (b_1, b_2, b_3, \ldots, b_L)$
has as its result a sum vector $\vec{C}$, with $c_i, = a_i + b_i$.

OP Code	MNEM Code		Instruction
40	VA	-	Vector add fixed point, single length
41	VAH	-	Vector add fixed point, half length
42	VAF	-	Vector add floating point, single length
43	VAFD	-	Vector add floating point, double length

ADD MAGNITUDE

A vector add magnitude instruction with argument vectors $\vec{A}$ and $\vec{B}$ generates a result vector $\vec{C}$,
where $\vec{C} = (c_1, c_2, c_3, \ldots, c_L)$
with $c_i = a_i + |b_i|$.

OP Code	MNEM Code		Instruction
44	VAM	-	Vector add magnitude fixed point, single length
45	VAMH	-	Vector add magnitude fixed point, half length
46	VAMF	-	Vector add magnitude floating point, single length
47	VAMFD	-	Vector add magnitude floating point, double length

SUBTRACT

A vector subtract instruction with argument vectors $\vec{A}$ and $\vec{B}$ generates a result vector $\vec{C}$,
where $\vec{C} = (c_1, c_2, c_3, \ldots, c_L)$
with $c_i = a_i - b_i$.

OP Code	MNEM Code		Instruction
48	VS	-	Vector subtract fixed point, single length
49	VSH	-	Vector subtract fixed point, half length
4A	VSF	-	Vector subtract floating point, single length
4B	VSFD	-	Vector subtract floating point, double length

SUBTRACT MAGNITUDE

A vector subtract magnitude instruction with argument vectors $\vec{A}$ and $\vec{B}$ generates a result vector $\vec{C}$,

where $\vec{C} = (c_1, c_2, c_3, \ldots, c_L)$

with $c_i = a_i - |b_i|$.

OP Code	MNEM Code		Instruction
4C	VSM	-	Vector subtract magnitude fixed point, single length
4D	VSMH	-	Vector subtract magnitude fixed point, half length
4E	VSMF	-	Vector subtract magnitude floating point, single length
4F	VSMFD	-	Vector subtract magnitude floating point,double length

MULTIPLY

A vector multiply instruction with argument vectors $\vec{A}$ and $\vec{B}$ generates a result vector $\vec{C}$.

where $\vec{C} = (c_1, c_2, c_3, \ldots, c_L)$

with $c_i = a_i \cdot b_i$.

See product length options table.

OP Code	MNEM Code		Instruction
6C	VM	-	Vector multiply fixed point, single length
6D	VMH	-	Vector multiply fixed point, half length
6E	VMF	-	Vector multiply floating point, single length
6F	VMFD	-	Vector multiply floating point, double length

DOT PRODUCT

The vector dot product instruction forms a sum of products of the type:

$$c_1 = \sum_{i=1}^{L} a_i \cdot b_i \qquad \text{(scalar result)}$$

where the a_i are elements of a row vector $\vec{A} = (a_1, a_2, a_3, \ldots a_L)$
and the b_i are elements of a column vector $\vec{B} = (b_1, b_2, b_3, \ldots b_L)$

The scalar result, c_1, is stored in central memory at the location specified by SAC + (XC). See product length options table.

OP Code	MNEM Code		Instruction
68	VDP	-	Vector dot product, fixed point, single length
69	VDPH	-	Vector dot product, fixed point, half length
6A	VDPF	-	Vector dot product, floating point, single length
6B	VDPFD	-	Vector dot product, floating point, double length

DIVIDE

A vector divide instruction with argument vectors $\vec{A}$ and $\vec{B}$, forms a result vector $\vec{C}$ such that each element c_i of the result equals a_i divided by b_i. See length options table.

$$c_i = a_i/b_i$$

The four cases for single-valued vectors are:

SV-field	Operation	Value K
x 1 0 0	k/b_i	k ≡ (SAA + (XA))
x 1 0 1	a_i/k	k ≡ (SAB + (XB))
x 1 1 0	k/b_i	k ≡ (29) } immediate
x 1 1 1	a_i/k	k ≡ (2A) } operand

OP Code	MNEM Code	Instructions
64	VD -	Vector divide fixed point, single length
65	VDH -	Vector divide fixed point, half length
66	VDF -	Vector divide floating point, single length
67	VDFD -	Vector divide floating point, double length

LENGTH OPTIONS

The dividend length options for vector divide and the product length options for vector multiply and vector dot product are specified by the MSB of the SV-field (single value field) as follows:

Argument Size / SV-field	Fixed Point: single length	Fixed Point: half length	Floating Point: single length	Floating Point: double length
0 x x x	64-bit	32-bit	32-bit	64-bit
1 x x x	32-bit	16-bit	32-bit	64-bit

NOTES: Vector dot products accumulate a 64-bit sum in the arithmetic unit. Whether the 16, 32, or 64 LSB's are read will depend on the type of VDP and the SV-field as shown above.

Fixed point signed integer products are formed from vector multiply and vector dot product instructions. Fixed point signed integer dividends and quotients are used and produced in vector divide instructions.

The product length for Vector Multiply, fixed point, single length, is 64-bits. When SV = 0 xxx, all 64-bits are stored. When SV = 1 xxx, the least significant 32-bits of the product are stored. Overflow cannot occure when SV = 0 xxx. Overflow is detected during the vector operation when SV = 1 xxx if the most significant portion of the product exceeds 32-bits.

Overflow is detected in the Arithmetic Unit for the above case if the 33 MSB's of the 64-bit product are not all "ones" or not all "zeros".

The product length for Vector Multiply, fixed point, half length is 32-bits. When SV = 0xxx, all 32 bits are stored and overflows cannot occur. When SV = 1xxx, the least significant 16-bits of the product are stored and overflows are detected during the vector operation if the significant portion of the product exceeds 16-bits.

Fixed point, single length, Vector Dot Product operations generate 64-bit products and accumlate a 64-bit sum in the Arithmetic Unit. When SV = 0xxx, the entire 64-bit sum is stored and overflow is detected during the vector operation if the sum exceeds the 64-bit accumlator word size.

When SV = 1xxx, the 32 least significant bits of the sum are stored. Overflow is detected during the vector operation if the significant portion of the 64-bit sum exceeds 32-bits.

Fixed point, half length, Vector Dot Product operations generate 32-bit products and accumulate a 32-bit sum in the Arithmetic Unit. When SV = 0xxx, a 32-bit sum is stored and overflow is detected during the vector operation if the sum exceeds 32-bits.

When SV = 1xxx, the 16 least significant bits of the 32-bit sum are stored. Overflow is detected during the vector operation if the significant portion of the 32-bit sum exceeds 16-bits.

During Vector Divide, fixed point, single length, when SV = 0xxx, the dividends are 64-bit signed integers. When SV = 1xxx, the dividends are 32-bit signed integers. The divisors are 32-bit signed integers. When the relative magnitude of dividend and divisor is such that the quotient cannot be expressed by a 32-bit signed integer, an overflow occurs and the central memory location corresponding to that output element is loaded with an unpredictable number.

During Vector Divide, fixed point, half length, when SV = 0xxx, the dividends are 32-bit signed integers. When SV = 1xxx, the dividends are 16-bit signed integers. The divisors are 16-bit bit signed integers. When the relative magnitude of dividend and divisor is such that the quotient cannot be expressed by a 16-bit signed integer, an overflow occurs and the central memory location corresponding to that output element is loaded with an unpredictable number.

In cases where a given length vector input argument is specified by the vector op code, but where the SV-field or other vector specifiers indicate a different word size result, the delta increments for the inner and outer loops will be adjusted automatically by hardware such that an increment of K results in an address advancement of K-words of whatever word size is appropriate. For example, a singlelength, fixed point, vector multiply may be specified with an SV-field of 0xxx which indicates that a double length product is to be generated. In this case, a delta C increment (DCI or DCØ) which is equal to the value K will advance the C vector address by K doublewords.

LOGICAL INSTRUCTIONS

A vector logical instruction with argument vectors $\vec{A}$ and $\vec{B}$, forms a result vector $\vec{C}$, where

$$c_i = a_i \text{ Boolean operation } b_i$$

The Boolean operations are defined for bits of the singleword or double word elements of a_i and b_i.

Boolean operation	Logical Equation
AND	$x \cdot y$
OR	$x + y$
Exclusive OR	$x \overline{y} + \overline{x} y$
Equivalence	$x y + \overline{x}\,\overline{y}$

where x = bit j of element a_i
and y = bit j of element b_i
for j range 0 through 31 if single length and 0-63 if double length.

OP Code	MNEM Code	Instruction
E0	VAND -	Vector logical AND, single length
E4	VOR -	Vector logical OR, single length
E8	VXOR -	Vector logical Exclusive OR, single length
EC	VEQC -	Vector logical Equivalence, single length
E1	VANDD -	Vector logical AND, double length
E5	VORD -	Vector logical OR, double length
E9	VXORD -	Vector Exclusive OR, double length
ED	VEQCD -	Vector Equivalence, double length

SHIFT INSTRUCTIONS

A vector shift instruction with argument vectors $\vec{A}$ and $\vec{B}$ result in vector $\vec{C}$, where

$$c_i = a_i \text{ shifted SC bit positions.}$$

The shift count, SC, is a 7-bit signed integer contained in bit positions 25 through 31 of the elements of vector B. Negative shift counts are represented in 2's complement form. A negative sign represents a right shift and a positive sign a left shift of SC positions.

OP Code	MNEM Code	Instruction
C0	VSA -	Vector arithmetic shift, fixed point, single length
C1	VSAH -	Vector arithmetic shift, fixed point, half length
C3	VSAD -	Vector arithmetic shift, fixed point, double word
C4	VSL -	Vector logical shift, single length
C5	VSLH -	Vector logical shift, half length
C7	VSLD -	Vector logical shift, double length
CC	VSC -	Vector circular shift, single length
CD	VSCH -	Vector circular shift, half length
CF	VSCD -	Vector circular shift, double length

PROGRAM INTERRUPTION: Fixed point overflow is detected, for arithmetic left shifts only, if the sign bit changes during the shift. The entire shift operation designated by the shift count is completed regardless of overflow conditions.

MERGE INSTRUCTIONS

A vector merge singleword instruction with arguments $\vec{A}$ and $\vec{B}$,

where $\vec{A} = (a_1, a_2, a_3, \ldots, a_L)$

and $\vec{B} = (b_1, b_2, b_3, \ldots, b_L)$

generates an output vector $\vec{C}$, where

$$\vec{C} = (a_1, b_1, a_2, b_2, a_3, b_3, \ldots, a_L, b_L)$$

or

$$\left.\begin{aligned} c_{2i-1} &= a_i \\ c_{2i} &= b_i \end{aligned}\right\} \text{ for } (i = 1, 2, 3, \ldots, L)$$

The elements a_i and b_i above represent single length, 32-bit words.

The L-field in the vector parameter file specifies the input vector length for vector merge instructions. The output vector will be twice the length of the input vector.

A vector merge halfword instruction generates an output vector $\vec{C}$ as in VMG, except that for this instruction the elements a_i and b_i represent half length, 16-bit words.

A vector merge doubleword instruction generates an output vector $\vec{C}$ as in VMG, except that for this instruction the elements a_i and b_i represent double length, 64-bit words.

OP Code	MNEM Code	Instruction
D8	VMG -	Vector merge single words
D9	VMGH -	Vector merge half words
DB	VMGD -	Vector merge double words

ORDER INSTRUCTIONS

A vector order instruction performs an arithmetic comparison of the elements of vectors $\vec{A}$ or $\vec{B}$, such that the smaller element, whether from $\vec{A}$ or $\vec{B}$, is the next element to be stored in forming the output vector $\vec{C}$.

Programing Notes:

1) The Vector Order instruction only applies to the vector self loop. There is no inner not outer loop feature for this instruction. If inner (NI) or outer (NØ) loop counts are specified in the vector parameter file, they are disregared by the hardware and only the self loop (vector operation of length L) will be executed.

2) Floating point vectors $\vec{A}$ and $\vec{B}$ must be normalized prior to use in a vector order instruction.

3) A boundary limit equal to the largest positive number must be placed in the data location following the last entry in the files to be ordered. These boundary values for the different data formats are:

Boundary limit				Data Format
7FFF	FFFF			Single length, fixed point
7FFF				Half length, fixed point
7FFF	FFFF			Single length, floating point
7FFF	FFFF	FFFF	FFFF	Double length, floating point

4) The output vector may not be written over either input vector.

Programming Example: VØD instruction

A = 2, 4, 5, 7, 1, 3, 4, 6, 8, 5, 6, 12, (7FFF)

B = 3, 6, 8, 9, 2, 3, (7FFF)

C = 2, 3, 4, 5, 6, 7, 1, 3, 4, 6, 8, 5, 6, 8, 9, 2, 3, 12, (7FFF)

where $L = l_A + l_{B+1} = 12 + 6 + 1 = 19$

The length specification, L, in the vector parameter file should be set to the sum of the lengths of the two vectors $\vec{A}$ & $\vec{B}$ plus 1 to include at least one boundary limit so that the result vector $\vec{C}$ can be used in a subsequent vector order instruction if desired.

$$L = \text{length } \vec{A} + \text{length } \vec{B} + 1$$

If, during the processing of a Vector Order instruction, two equal values are simultaneously presented to the arithmetic unit from vectors $\vec{A}$ and $\vec{B}$, the value presented from vector $\vec{A}$ is the element which is delivered to the arithmetic unit output; the value from vector $\vec{B}$ is retained at the arithmetic unit input for comparison with the next element of vector $\vec{A}$.

OP Code	MNEM Code		Instruction
D4	VO	-	Vector order singlewords, fixed point
D5	VOD	-	Vector order halfwords, fixed point
D6	VOF	-	Vector order singlewords, floating point
D7	VOFD	-	Vector order doublewords, floating point

ARITHMETIC

Element a_i of input vector $\vec{A}$ is arithmetically compared with element b_i of input vector $\vec{B}$. The result of this comparison will set one of the three comparison code bits and reset (zero) the other two depending on whether the comparison is: 1) a_i less than b_i, 2) a_i greater than b_i, 3) a_i equal to b_i. The comparison code bits CL, CG, and CE are then matched with the 3 LSB's of the ALCT field (hexadecimal character H_2 of register 28; bits denoted by r_1, r_2, and r_3). If the logical equation COND = $r_1 \cdot CL + r_2 \cdot CG + r_3 \cdot \overline{CE}$, is true for element a_i and b_i, then the index value, i, corresponding to the position of element a_i and b_i in the vector input stream, is stored as a halfword value in result vector $\vec{C}$. The first index value is stored into halfword location 2 • SAC +(XC)+HSC+1. Successive index values are stored into consecutive halfword locations. If COND is false, nothing is stored.

The next operand elements of the input vectors (a_{i+1} and b_{i+1}) are then acquired and the operation is repeated until the length (L) of the vector has been exhausted. Just before this vector instruction is terminated, a count of the number of items for which COND was true is stored into halfword memory location 2 • SAC + (XC) + HSC.

The ALCT-field (Arithmetic and Logical Comparison Test) of the vector parameter file is used to specify one of the following comparison options for a Vector Arithmetic Comparison instruction.

ALCT-field bits	r_1	r_2	r_3	Vector arithmetic Comparison options, a_i:b_i
X	0	0	0	do nothing
X	0	0	1	$a_i = b_i$
X	0	1	0	$a_i > b_i$
X	0	1	1	$a_i \geq b_i$
X	1	0	0	$a_i < b_i$
X	1	0	1	$a_i \leq b_i$
X	1	1	0	$a_i \neq b_i$
X	1	1	1	store index i for all i from 0 through L-1

When the 000 option is specified, the AU compares all a_i and b_i elements, but since COND never becomes true, no index values are ever stored. Although, before this instruction is terminated, a count of the number of items for which COND was true (for this case, item count equals zero) is stored into halfword memory location 2 • SAC + (XC) + HSC.

There are four types of Vector Arithmetic Comparison instructions, one for each of the four data formats.

OP Code	MNEM Code		Instruction
D0	VC	-	Vector arithmetic comparison, fixed point, single length
D1	VCH	-	Vector arithmetic comparison, fixed point, half length
D2	VCF	-	Vector arithmetic comparison, floating point, single length
D3	VCFD	-	Vector arithmetic comparison, floating point, double length

Programming Notes:

1) The address of a singleword vector element a_i is equal to SAA + (XA) + i, where SAA is the starting address of the vector $\vec{A}$, (XA) is the static index value during a given vector operation, and i is the dynamic index value. The value of i runs from 0 to L-1 during a vector operation. The maximum range of i is equal to the maximum range of the length specification (L) of a vector operation. L is limited to 16-bits corresponding to a maximum vector length of $2^{16} - 1$ or 65,535 elements.

2) The most significant bit of the ALCT-field in the vector parameter file is used to specify whether the vector comparison is to continue the full length of the vector operation or terminate after the first comparison true has been detected.

   If the MSB of the ALCT-field is "zero", then the Vector Arithmetic or Logical Comparison operation (whichever is specified) will continue until the length of the vector has been exhausted. However, if the MSB of the ALCT-field is "one", then the Vector Arithmetic or Logical Comparison operation will be terminated after the first comparison true condition has been detected. Just before the instruction is terminated the index value,i, corresponding to the position of elements a_i and b_i for which COND is true will be stored into halfword location 2 • SAC + (XC) + HSC + 1 and a count of the number of items for which COND was true will be stored into halfword memory location 2 • SAC + (XC) + HSC. In this case, the item count will be equal to one, if COND ever becomes true during the vector comparison operation. The item count will be equal to zero and no index value will be stored if COND never becomes true during the vector comparison operation.

For example, if inner and outer loops are used and the first comparison true is detected during the third inner loop, then two zero item counts are stored into locations 2·SAC+(XC)+HSC and 2·SAC+(XC)+HSC+1. The item count for the third self loop, which has a value of one, is stored into location 2·SAC+(XC)+HSC+ (loop number minus one) which equals 2·SAC+(XC)+HSC+2. The index value of the first element which compared is stored into location 2·SAC+(XC)+HSC+ (loop number) which equals 2·SAC+(XC)+HSC+3. The index value, i, stored here is referenced to the particular self loop being processed at the time that the first comparison true is detected. For example, if the first element of a new self loop is the first one which has a comparison true, then the index value, i, is equal to zero.

3) If it is desirable to use an odd halfword starting address for result vector $\vec{C}$ and a single or double length immediate operand is used (immediate single-valued vector), the immediate operand K must be stored in register 29. If K were stored in register 2A, the (HS) halfword starting address information would be covered by K. An SV-field equal to X'10' will specify an immediate single-valued operand K, where K is obtained from register 29. Another instruction which stores half length results, but which may use single or double length input arguments is the Vector Logical Comparison instruction.

4) Floating point vector $\vec{A}$ and $\vec{B}$ must be normalized prior to use in a vector arithmetic comparison instruction.

LOGICAL "AND" COMPARE INSTRUCTIONS

Element a_i of input vector $\vec{A}$ is logically "ANDed" with element b_i of input vector $\vec{B}$. One of the three comparison code bits (CL, CG, or CE) will be set depending upon the logical properties of c_i, where $c_i = a_i \wedge b_i$. For logical operations the conditions code is set as follows: (CL) c_i contains mixed "ones" and "zeros", (CG) all bit positions of c_i are "one", or (CE) all bit positions of c_i are "zero".

The comparison code bits CL, CG, and CE are then matched with the 3 LSB's of the ALCT field (bits that we shall label as r_1, r_2, and r_3). If the logical equation, $COND = r_1 \cdot CL + r_2 \cdot CG + r_3 \cdot CE$, is true, then the index value, i, of input vector element a_i and b_i is stored as a halfword value in result vector $\vec{C}$. The first index value is stored into halfword location $2 \cdot SAC + (XC) + HSC + 1$. Successive index values are stored into consecutive halfword locations. If COND is false, nothing is stored.

The next operand elements of the input vectors (a_{i+1} and b_{i+1}) are then acquired and the operation is repeated until the length (L) of the vector has been exhausted. Just before this vector operation is terminated, a count of the number of items for which COND was true is stored into halfword memory location $2 \cdot SAC + (XC) + HSC$.

The ALCT-field of the vector parameter file is used to specify one of the following comparison options for a Vector Logical Comparison instruction.

ALCT-field bits	r_1	r_2	r_3	Vector Logical Comparison options
X	0	0	0	Do nothing
X	0	0	1	All zeros
X	0	1	0	All ones
X	0	1	1	All ones or all zeros ≡ (not mixed)
X	1	0	0	Mixed ones and zeros
X	1	0	1	Not all ones ≡ (mixed or all zeros)
X	1	1	0	Not all zeros ≡ (mixed or all ones)
X	1	1	1	Store index i for all i from 0 through L-1

The comparisons in this table refer to the logical properties of $a_i \wedge b_i$ for a VCAND instruction and to $a_i \vee b_i$ for a VCOR instruction.

There are four types of Vector Logical Comparison instructions, two for each of two data lengths.

OP Code	MNEM Code		Instruction
E2	VCAND	-	Vector logical comparison using AND, single length
E3	VCANDD	-	Vector logical comparison using AND, double length
E6	VCOR	-	Vector logical comparison using OR, single length
E7	VCORD	-	Vector logical comparison using OR, double length

Vector Logical Comparisons using ØR functions are described identical to the logical AND comparison instructions, except that element a_i of input vector $\vec{A}$ is logically "ORed" with element b_i of input vector $\vec{B}$.

The vector terminating feature described for arithmetic compares are effective on all of the Vector Logical Comparison instructions listed above when the MSB of the ALCT-field in the vector parameter file is "one".

If it is desirable to use an odd halfword starting address for result vector $\vec{C}$ and a single or double length immediate operand is used (immediate single-valued vector), the immediate operand K must be stored in register 29. If K were stored in register 2A, the (HS) halfword starting address information would be covered by K. An SV-field equal to X'10' will specify an immediate single-valued operand K, where K is obtained from register 29.

SEARCH INSTRUCTIONS

There are sixteen search instructions. Four types of each of the following:

Search for largest arithmetic element
Search for largest magnitude
Search for smallest arithmetic element
Search for smallest magnitude

The four types of each of the above refer to word size data representation: (1) fixed point, single length; (2) fixed point, half length; (3) floating point, single length; and (4) floating point, double length.

The search instruction tests every element, a_i, of Vector $\vec{A}$ relative to all other elements of $\vec{A}$ and stores the index value, i, of the largest or smallest element (depending on the operation code) into the halfword memory location specified by address 2 . SAC + (XC) + HSC. The value of i is within the range 0 through L-1 and is the dynamic index value of $\vec{A}$ during a vector operation.

Programming Note: Floating point input vector $\vec{A}$ must be normalized prior to use in a vector search instruction.

Vector Search for Largest with 2 or more largest elements of equal value will store as its output the index of the first of such elements.

Similar logic applies to the vector search for largest magnitude, search for smallest, and search for smallest magnitude instructions.

Vector Search for Largest Magnitude will recognize the number 8000 0000 for fixed point single length of 8000 for fixed point half length instructions as having a larger arithmetic magnitude than 7FFF FFFF or 7FFF, respectively.

Vector Search for Smallest Magnitude will recognize the number 8000 0000 for fixed point single length or 8000 for fixed point half length instructions as having the largest magnitude and will therefore not output its index if any other element of the vector has a smaller magnitude.

OP Code	MNEM Code		Instructions
50	VL	-	Vector search for largest arithmetic element, fixed point, single length
51	VLH	-	Vector search for largest arithmetic element, fixed point, half length
52	VLF	-	Vector search for largest arithmetic element, floating point, single length
53	VLFD	-	Vector search for largest arithmetic element, floating point, double length
54	VLM	-	Vector search for largest magnitude, fixed point, single length
55	VLMH	-	Vector search for largest magnitude, fixed point, half length
56	VLMF	-	Vector search for largest magnitude, floating point, single length
57	VLMFD	-	Vector search for largest magnitude, floating point, double length
58	VSS	-	Vector search for smallest arithmetic element, fixed point, single length
59	VSSH	-	Vector search for smallest arithmetic element, fixed point, half length
5A	VSSF	-	Vector search for smallest arithmetic element, floating point, single length
5B	VSSFD	-	Vector search for smallest arithmetic element, floating point, double length
5C	VSSM	-	Vector search for smallest magnitude, fixed point, single length
5D	VSSMH	-	Vector search for smallest magnitude, fixed point, half length
5E	VSSMF	-	Vector search for smallest magnitude, floating point, single length
5F	VSSMFD	-	Vector search for smallest magnitude, floating point, double length

PEAK PICKING INSTRUCTIONS

ZERO CROSSING

The algorithm for the vector peak picking instruction is as follows:

$y_i = a_{i-1} - a_i$ for i = (1, 2, 3, ..., L-2)
$y_{i+1} = a_i - a_{i+1}$

If the sign of y_i is different than the sign of y_{i+1}, then store the index value, i.

If the sign of y_i is the same as the sign of y_{i+1}, then do not store the index value, i.

When the value of y_{i+1} is zero, y_{i+1} is considered to retain the sign of the last non-zero value in the history of y_i. This convention will select the trailing edge of a trace for which a series of $y_i = 0$ conditions exist, i.e., the "peak" value which is stored is at the trailing edge of a mesa. Points of inflection are not stored.

The formats for the storage of the index value, i., for the four types of vector peak picking instructions are identical to the formats for the vector test instructions.

The item count entered at halfword location 2 · SAC + (XC) + HSC of the output table is a count of the total number of peak and valley points stored as a result of the vector peak picking instruction.

The most significant bit of the ALCT-field in the vector parameter file is used to specify whether the Vector Peak Picking instruction is to continue the full length of the vector operation (as designated by the L-field) or terminate after the first peak or valley point has been detected.

If the MSB of the ALCT-field is "zero", then the Vector Peak Picking operation will continue until the length (L) of the input vector has been exhausted and all the peak and valley index points have been stored. However, if the MSB of the ALCT-field is "one", then the Vector Peak Picking operation will be terminated after the first peak or valley point has been detected. If a peak or valley point is detected and the MSB of ALCT is "one", then the index value, i, of that peak or valley point is stored into halfword location 2 · SAC + (XC) + HSC + 1 and a one, corresponding to the number of index values stored, is entered into halfword location 2 · SAC + (XC) + HSC. The item count will be equal to zero and no index values will be stored if the input vector elements are monotone increasing or decreasing.

Programming Note: A floating point input vector must be normalized prior to use in a vector peak picking instruction.

OP Code	MNEM Code		Instructions
DC	VPP	-	Vector peak, fixed point, single length
DD	VPPH	-	Vector peak, fixed point, half length
DE	VPPF	-	Vector peak, floating point, single length
DF	VPPFD	-	Vector peak, floating point, double length

Fixed point overflow is indicated for the VPP and VPPH instructions if a discontinuity exists between any two data points whose difference exceeds one half the range of the fixed point number representation.

Floating point overflow is indicated for the VPPF and VPPFD instructions if a discontinuity exists between two data points such that the difference results in a floating point overflow condition.

Floating point underflow is indicated for the VPPF and VPPFD instructions if the difference between two data points would cause an exponent underflow condition.

CONVERSION INSTRUCTIONS

FLOATING TO FIXED POINT

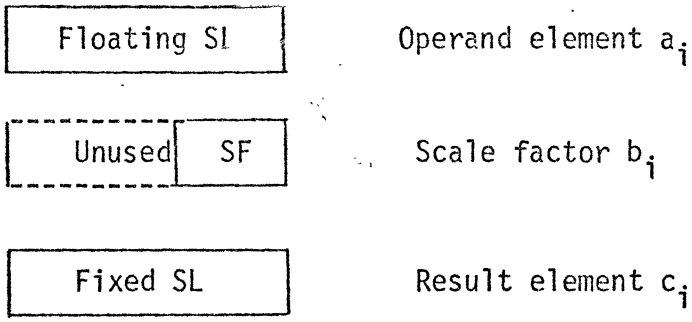
Vector conversion instructions acquire the operands to be converted from vector $\vec{A}$ and the scale factor from vector $\vec{B}$. Outputs are stored as result vector $\vec{C}$. When the same scale factor is applied to all conversions, an immediate or directly addressed single-valued $\vec{B}$ vector may be used. The algorithm for converting from floating to fixed point is the same as that previously described for scalar floating to fixed point conversions.

OP Code	MNEM Code		Instruction
AO	VFLFX	-	Vector convert floating point single length to fixed point single length

Vector $\vec{A}$ is the list of floating point single length elements to be converted. The elements are read from consecutive singleword memory locations beginning with starting address SAA + (XA).

Vector $\vec{B}$ is the list of 16-bit fixed point scale factors which have been pre-computed and which specify the placement of the fixed point signed integer result with respect to the decimal point to the right of the LSB. The scale factors are contained in the right half word of the singleword elements of Vector $\vec{B}$. Vector $\vec{B}$ may be specified as a single-valued vector in which case the same scale factor is applied to all converted elements.

The result vector $\vec{C}$ is a list of fixed point single length signed integer elements with scale factors according to the pre-determined values of vector $\vec{B}$.



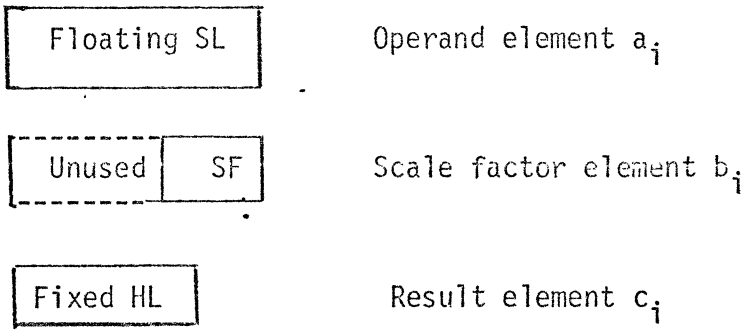
PROGRAM INTERRUPTION: Fixed point overflow.

OP Code	MNEM Code	Instruction
A1	VFLFH	Vector convert floating point single length to fixed point half length

Vector $\vec{A}$ is the list of floating point single length elements to be converted to fixed point half length representation.

Vector $\vec{B}$ is the list of scale factors which have been pre-computed and which specify the placement of the fixed point signed integer result. The scale factors are contained in the right half word of the singleword elements of Vector $\vec{B}$. Vector $\vec{B}$ may be specified as a single-valued vector in which case the same scale factor is applied to all converted elements.

The result vector $\vec{C}$ is a list of fixed point half length signed integer elements with scale factors according to the pre-determined values of vector $\vec{B}$. The elements of result vector $\vec{C}$ are stored in consecutive halfword locations.



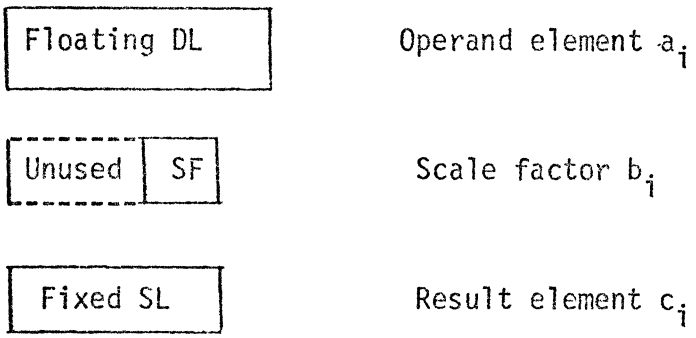
PROGRAM INTERRUPTION: Fixed Point Overflow

OP Code	MNEM Code	Instruction
A2	VFDFX	Vector convert floating point double length to fixed point single length

Vector $\vec{A}$ is the list of floating point double length elements to be converted to fixed point single length representation.

Vector $\vec{B}$ is the list of scale factors which have been pre-computed and which specify the placement of the fixed point signed integer result. The scale factors are contained in the right half word of the singleword elements of vector $\vec{B}$. Vector $\vec{B}$ may be specified as a single-valued vector in which case the same scale factor is applied to all converted elements.

The result vector $\vec{C}$ is a list of fixed point single length signed integer elements with scale factors according to the predetermined values of vector $\vec{B}$. The elements of result vector $\vec{C}$ are stored in consecutive singleword locations.



PROGRAM INTERRUPTION: Fixed point overflow

FIXED TO FLOATING POINT

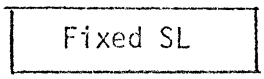
Fixed to floating point vector conversion instructions acquire the list of fixed point signed integer elements from vector $\vec{A}$. Vector $\vec{B}$ is the list of scale factors corresponding to the fixed point elements of vector $\vec{A}$. The scale factors are contained in the right half of the singleword elements of vector $\vec{B}$. The result vector $\vec{C}$ is a list of normalized floating point elements. The hexadecimal exponents of the floating point numbers are determined from the fixed point scale factors and the amount of shifting required to normalize the floating point fraction. The algorithm for fixed to floating point conversion is the same as described previously for scalar fixed to floating point conversions.

OP Code	MNEM Code	Instructions
A8	VFXFL	Vector convert fixed point single length to floating point single length

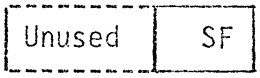
Vector $\vec{A}$ is the list of fixed point single length signed integer elements to be converted to floating point representation.

Vector $\vec{B}$ is the list of scale factors corresponding to the fixed point elements of vector $\vec{A}$. The scale factors are contained in the right half word of the singleword elements of vector $\vec{B}$. Vector $\vec{B}$ may be specified as a single-valued vector in which case the same scale factor is applied to each and every fixed point number of vector $\vec{A}$.

The result vector $\vec{C}$ is a list of normalized floating point single length elements. The hexadecimal exponents of the floating point numbers are determined from the fixed point scale factors and the amount of shifting required to normalize the floating point fraction.



Operand element a_i



Scale factor b_i

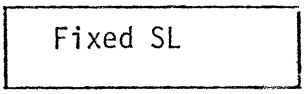
Result element c_i

PROGRAM INTERRUPTION: Floating point overflow

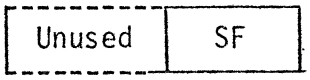
OP Code	MNEM Code	Instructions
AA	VFXFD	Vector convert fixed point single length to floating point double length

Vector $\vec{A}$ is the list of fixed point single length signed integer elements to be converted to floating point representation.

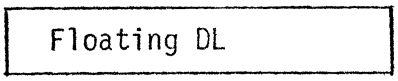
Vector $\vec{B}$ is a list of scale factors corresponding to the fixed point elements of vector $\vec{A}$. The scale factors are contained in the right half ward of the singleword elements of vector $\vec{B}$. Vector $\vec{B}$ may be specified as a single-valued vector in which case the same scale factor is applied to each and every fixed point number of vector $\vec{A}$. The result vector $\vec{C}$ is a list of normalized floating point double length elements. The hexadecimal exponents of the floating point numbers are determined from the fixed point scale factors and the amount of shifting required to normalize the floating point fraction.



Operand element a_i



Scale factor b_i



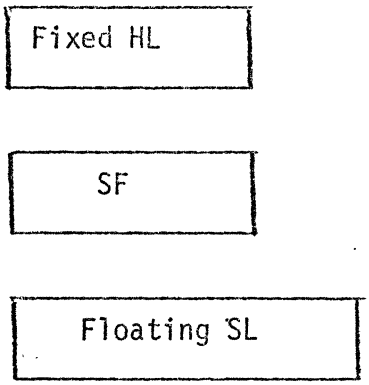
Result element c_i

PROGRAM INTERRUPTION: Floating point overflow

OP Code	MNEM Code	Instructions
A9	VFHFL	Vector convert fixed point half length to floating point single length

Vector $\vec{A}$ is the list of fixed point half length signed integer elements to be converted to floating point representation. Vector $\vec{B}$ is the list of scale factors corresponding to the fixed point elements of vector $\vec{A}$. The scale factors are contained in the halfword elements of vector $\vec{B}$. Vector B may be specified as a single-valued vector in which case the same scale factor is applied to each and every fixed point number of vector $\vec{A}$.

The result vector $\vec{C}$ is a list of normalized floating point single length elements. The hexadecimal exponents of the floating point numbers are determined from the fixed point scale factors and the amount of shifting required to normalize the floating point fraction.



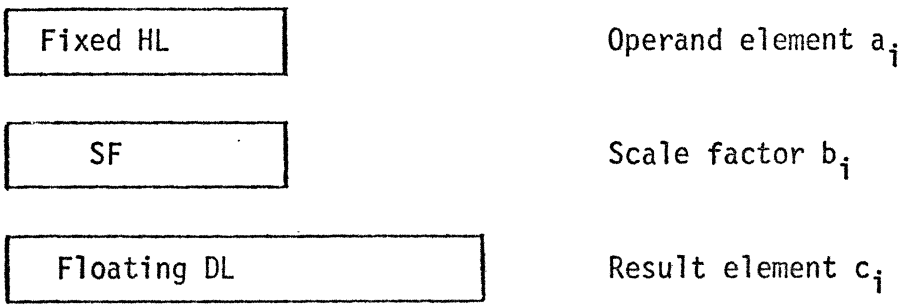
PROGRAM INTERRUPTION: Floating Point Overflow

OP Code	MNEM Code	Instructions
AB	VFHFD	Vector convert fixed point half length to floating point double length

Vector $\vec{A}$ is the list of fixed point half length signed integer elements to be converted to floating point representation.

Vector $\vec{B}$ is the list of scale factors corresponding to the fixed point elements of vector $\vec{A}$. The scale factors are contained in the halfword elements of vector $\vec{B}$. Vector $\vec{B}$ may be specified as a single-valued vector in which case the same scale factor is applied to each and every fixed point number of vector $\vec{A}$.

The result vector $\vec{C}$ is a list of normalized floating point double length elements. The hexadecimal exponents of the floating point numbers are determined from the fixed point scale factors and the amount of shifting required to normalize the floating point fraction.



PROGRAM INTERRUPTION: Floating Point Overflow

NORMALIZE INSTRUCTIONS

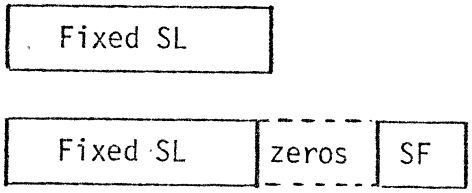
OP Code	MNEM	Instructions
AC	VNFX	Vector normalize fixed point single length

Vector $\vec{A}$ is the list of fixed point single length elements to be normalized.

Vector $\vec{B}$ is not used in this operation.

Result vector $\vec{C}$ is the list of normalized fixed point single length elements and scale factor. The scale factor is stored into the right quarter of doubleword element c_i and represents the number of bit positions that the fixed point fraction was shifted left until becoming normalized. The number of positions shifted is stored as a negative 2's complement number.

The left half of doubleword element c_i contains the normalized fixed point single length element corresponding to singleword element a_i of input vector $\vec{A}$.



PROGRAM INTERRUPTION: None

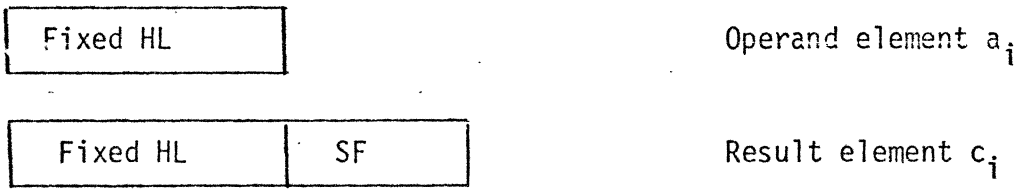
OP Code	MNEM Code	Instructions
AD	VNFH	Vector normalize fixed point half length

Vector $\vec{A}$ is the list of fixed point half length elements to be normalized.

Vector $\vec{B}$ is not used in this operation.

Result vector $\vec{C}$ is the list of normalized fixed point half length elements and scale factor. The scale factor is stored into the right half of singleword element c_i and represents the number of bit positions that the fixed point fraction was shifted left until becoming normalized. The number of positions shifted is stored as a negative 2's complement number.

The left half of singleword element c_i contains the normalized fixed point half length element corresponding to halfword element a_i of input vector $\vec{A}$.



PROGRAM INTERRUPTION: None

NOTE:

The SELECT and REPLACE instructions which follow have not been implemented for ASC serial numbers 1 and 2, but will be included in ASC serial number 3 and all subsequent machines. These instructions will be fully supported by the software and by the instruction level simulators

SELECT INSTRUCTION

A vector select instruction generates an output vector $\vec{C}$ composed of elements from vector $\vec{A}$. The elements selected from vector $\vec{A}$ are those for which the index location in vector $\vec{A}$ corresponds to the index value given by the elements of vector $\vec{B}$.

Programming Notes:

(1) Input vectors $\vec{A}$ and $\vec{B}$ are read from contiguous memory and the output is stored into contiguous memory for a given self loop.

(2A) The length specification of the self loop (L-field) for a vector select instruction is normally set equal to the number of elements of vector $\vec{A}$.

(2B) It is possible to shorten the vector operation and still obtain the same result vector $\vec{C}$ by setting the self loop length equal to one plus the value of the last index in vector $\vec{B}$.

(3A) If the vector length is specified according to 2A above, then an index boundary limit equal to the largest positive number ($7FFF_{hex}$) must be placed in the data location following the last index value of vector $\vec{B}$.

(3B) If the vector length is specified according to 2B above, then the index boundary limit is not necessary.

(4) Each index value given by vector $\vec{B}$ is a positive fixed point halfword. Vector $\vec{B}$ should be a contiguous list of monotone increasing halfwords.

(5) An index value of zero selects the first element of vector $\vec{A}$.

(6) If inner or outer loops are employed, then a dummy value should be placed at the end of each self loop vector $\vec{A}$ and the index of this dummy value should be placed at the end of each self loop index vector $\vec{B}$. Each successive index list must be in contiguous memory, i.e., DBI and DBØ must be equal to one. Vector $\vec{A}$ may use delta increments not equal to one for inner or outer loops if desired. However, the resultant vector $\vec{C}$ of selected elements should use delta increments, DCI and DCØ equal to one if the number of selected elements varies from self loop to self loop. Delta increments for vector $\vec{C}$ are added to the address of the last element selected for each self loop.

OP CODE	MNEMONIC CODE	INSTRUCTION
B4	VSEL	Select singlewords from vector $\vec{A}$
B5	VSELH	Select halfwords from vector $\vec{A}$
B7	VSELD	Select doublewords from vector $\vec{A}$

Example: A singleword select instruction using one self loop of length 8.

Singleword vector $\vec{A}$	Halfword index vector $\vec{B}$	Singleword selected vector $\vec{C}$
+16	2 , 3	-54
+72	5 , 6	-75
-54	7FFF , -	-64
-75		-15
+71		
-64		
-15		
+14		

REPLACE INSTRUCTION

A vector replace instruction accepts as inputs a contiguous list of replacement elements from vector $\vec{A}$ and a contiguous list of indices from vector $\vec{B}$. Elements from vector $\vec{A}$ replace previously existing elements in a central memory region defined as the $\vec{C}$ output array. Elements of the $\vec{C}$ output array that are replaced with elements of vector $\vec{A}$ are those elements for which the index location in the $\vec{C}$ output array corresponds to the index value given by the elements of vector $\vec{B}$.

Programming Notes:

(1) The length specification of the self loop (L-field) for a vector replace instruction should be set equal to the number of replacement elements in vector $\vec{A}$. This value is also equal to the number of indices of vector $\vec{B}$.

(2) Each index value given by vector $\vec{B}$ is a positive fixed point halfword. Vector $\vec{B}$ should be a contiguous list of monotone increasing halfwords.

(3) An index value of zero selects the first element of vector $\vec{A}$.

(4) If inner or outer loops are employed, then it becomes a requirement that each self loop be of the same length. In general, the length of the data replacement vectors throughout all of the inner and outer loops are not the same length. In order to obtain meaningful results using inner and outer loops, a dummy region of memory must be established at the end of the $\vec{C}$ data output array for each self loop. The size of the dummy region for each self loop $\vec{C}$ output array is equal to one plus the difference between the sizes of the maximum and minimum data replacement vectors as found by searching the data replacement lists throughout all inner and outer loops.

For the case of a self loop passing over the maximum data replacement vector, one dummy element is picked up one location past the end of the data replacement vector $\vec{A}$ and is placed in the final address available to the dummy output region of that self loop.

For the case of a self loop passing over the minimum data replacement vector, the first dummy replacement element after the last data replacement element is picked up and placed in the first location past the data output array, which is at the beginning of the dummy output region. The last dummy element is placed in the final address available to the dummy output region of that self loop.

This procedure establishes a constant number of replacement elements and indices for each self loop. The number of elements of the data output array is assumed to be constant for each self loop.

OP CODE	MNEMONIC	INSTRUCTION
B8	VREP	Replace singlewords in vector $\vec{C}$
B9	VREPH	Replace halfwords in vector $\vec{C}$
BB	VREPD	Replace doublewords in vector $\vec{C}$

Example: A singleword replace instruction using one self loop of length 4.

Singleword vector $\vec{A}$	Halfword index vector $\vec{B}$	Single vector $\vec{C}$ before replacement	Singleword vector $\vec{C}$ after replacement
-54	2 , 3	16	16
-72	5 , 6	72	72
-64		27	-54
-15		36	-72
		71	71
		32	-64
		8	-15
		14	14

CP Scalar Illegal Operation Codes

10	5E	79	A5
11	5F	7B	A6
		7E	A7
	61	7F	AE
26	63		AF
	69	9A	
53	6B	9B	B1
57	71	9E	B2
5A	73		B3
5B	76	A3	B4
5D	77	A4	
B5	C2	DA	F1
B6		DB	F3
B7	D0	DC	F5
B8	D1	DD	F7
B9	D2	DF	F9
BA	D3		FA
BB	D4	EA	FB
BC	D5	EB	FD
BD	D6	EE	FE
BE	D7	EF	FF
BF			

CP Vector Illegal Operation Codes

0X	60	C2
1X	61	C6
	62	C8
2X	63	C9
		CA
3X		
	A3	CB
7X	A4	CE
8X	A5	
	A6	DA
9X	A7	
BX	AE	EA
	AF	EB
FX		EE
		EF

where x represents any one of 16 possible codes (0,1, 2, 3,...C,D,E,F)

OP BITS 0-3

OP BITS 4-7	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NOP		STZ	LN	A	AI	A	AI	ISE	MCP	FLFX	VECT	SA		AND	ANDI
1			STZH	LNH	AH	AIH			ISNE	B---CC	FLFH		SAH		ANDD	
2		LAM	SPS	LNF	AF	LEA	A	AI	DSE	INT	FDFX				CAND	CANDI
3		LAC	STZD	LND	AFD				DSNE	PSH			SAD		CANDD	
4		L	ST	STN	AM	LI	D	DI	BCLE	MCW			SL		OR	ORI
5		LH	STH	STNH	AMH	LIH	DH	DIH	BCG	B---RC			SLH		ORD	
6		LLA		STNF	AMF	LEA	DF		BCLE	XEC			RVS		COR	CORI
7		LD	STD	STND	AMFD		DFD		BCG	PUL			SLD		CORD	
8		L	ST	LNM	S	SI	M	MI	IBZ	BLB	FXFL		C	CI	XOR	XORI
9		LL	STL	LNMH	SH	SIH			IBNZ	BLX	FHFL		CH	CIH	XORD	
A		XCH	STOH	LNMF	SF		M	MI	DBZ		FXFD		CF			
B		LF	STF	LNMD	SFD				DBNZ		FHFD		CFD			
C		L	ST	LM	SM	LI	M	MI	IBZ	BXEC	NFX		SC		EQC	EQCI
D		LR	STR	LMH	SMH		MH	MIH	IBNZ	B---AE	NFH		SCH		EQCD	
E		LO	STO	LMF	SMF		MF		DBZ				C	CI		
F		LFM	STFM	LMD	SMFD		MFD		DBNZ	MOD			SCD			

SCALAR *

*

OP BITS 0-3

OP BITS 4-7	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E
0					VA	VL					VFLFX		VSA	VC	VAND
1					VAH	VLH					VFLFH		VSAH	VCH	VANDD
2					VAF	VLF					VFDFX			VCF	VCAND
3					VAFD	VLFD							VSAD	VCFD	VCANDD
4					VAM	VLM	VD						VSL	VO	VOR
5					VAMH	VLMH	VDH						VSLH	VOD	VORD
6					VAMF	VLMF	VDF							VOF	VCOR
7					VAMFD	VLMFD	VDFD						VSLD	VOFD	VCORD
8					VS	VSS	VDP				VFXFL			VMG	VXOR
9					VSH	VSSH	VDPH				VFHFL			VMGH	VXORD
A					VSF	VSSF	VDPF				VFXFD				
B					VSFD	VSSD	VDPFD				VFHFD			VMG	
C					VSM	VSSM	VM				VNFX		VCS	VPP	VEQC
D					VSMH	VSSMH	VMH				VNFH		VCSH	VPPH	VEQCD
E					VSMF	VSSMF	VMF							VPPF	
F					VSMFD	VSSMFD	VMFD						VCSD	VPPFD	

VECTOR

SEQUENTIAL INDEX OF INSTRUCTIONS

MNEM DE	INSTRUCTION		OP CODE	PAGE NO.
BUO	FLOAT PT EXP UNDERFLOW OR OVERFLOW	R=3	9D	121
BX	FIXED PT OVERFLOW	R=4	9D	121
BXU	FIXED PT OVERFLOW OR FLOAT EXP UNDERFLOW	R=5	9D	121
BXO	FIXED PT OVERFLOW OR FLOAT EXP OVERFLOW	R=6	9D	121
BXUO	FIXED PT OVR OR FLOAT EXP PT OVR OR UNDERFLOW	R=7	9D	121
BD	DIVIDE CHECK	R=8	9D	121
BDU	DIVIDE CHECK OR FLOAT PT EXP UNDERFLOW	R=9	9D	121
BDO	DIVIDE CHECK OR FLOAT PT EXP OVERFLOW	R=A	9D	121
BDUO	DIVIDE CHECK OR FLOAT PT EXP UNDER OR OVERFLOW	R=B	9D	121
BDXU	DIVIDE CHK OR FIXED PT OVR OR FLOAT EXP UNDRFL	R=D	9D	122
BDX	DIVIDE CHECK OR FIXED PT OVERFLOW	R=C	9D	122
BDXO	DIVIDE CHK OR FIXED OVR OR FLOAT PT EXP OVRFLW	R=E	9D	122
BDXUO	DIVIDE CHK OR FIXED OVR OR FLT EXP OVR OR UNDR	R=F	9D	122
BXEC	BRANCH ON EXECUTE BRANCH CONDITION TRUE	R=1 OR ODD	9C	123
IBZ	INCREMENT TEST AND BRANCH ON ZERO ARITH REG		88	124
IBZ	INCREMENT TEST INDEX AND BRANCH ON ZERO		8C	124
IBNZ	INCREMENT TEST AND BRANCH ON NON-ZERO ARITH REG		89	125
IBNZ	INCREMENT TEST INDEX AND BRANCH ON NON-ZERO		8D	125
DBZ	DECREMENT TEST AND BRANCH ON ZERO ARITH REG		8A	126
DBZ	DECREMENT TEST INDEX AND BRANCH ON ZERO		8E	126
DBNZ	DECREMENT TEST AND BRANCH ON NON-ZERO ARITH REG		8B	127
DBNZ	DECREMENT TEST INDEX AND BRANCH ON NON-ZERO		8F	127
ISE	INCREMENT TEST AND SKIP ON EQUAL ARITH REG		80	128
ISNE	INCREMENT TEST AND SKIP ON NOT EQUAL ARITH REG		81	129
SE	DECREMENT TEST AND SKIP ON EQUAL ARITH REG		82	130
DSNE	DECREMENT TEST AND SKIP ON NOT EQUAL ARITH REG		83	131
BCLE	BRANCH ON ARITH REG LESS THAN OR EQUAL TO		84	132
BCLE	BRANCH ON INDEX LESS THAN OR EQUAL TO		86	132
BCG	BRANCH ON ARITH REG GREATER THAN		85	133
BCG	BRANCH ON INDEX GREATER THAN		87	133
PSH	PUSH WD ARITH REG		93	134
PUL	PULL WD ARITH REG		97	135
MOD	MODIFY ARITH REG		9F	135
BLB	BRANCH AND LOAD REG WITH PC		98	136
BLX	BRANCH AND LOAD INDEX REG OR VECTOR PARAM REG		99	137
LEA	LOAD EFFECTIVE ADDRESS INDEX REGISTER		56	138
LEA	LOAD EFFECTIVE ADDRESS INTO BASE REG		52	138
INT	INTERPRET ARITH REG		92	139
XEC	EXECUTE		96	139
MCP	MONITOR CALL AND PROCEED		90	140
MCW	MONITOR CALL AND WAIT		94	140
NOP	TAKE NEXT INSTRUCTION	R=0	91	115
FLFX	CONVERT FLOAT PT SNGLE LGTH TO FIXED PT SNGLE ARITH R		A0	145
FLFH	CONVERT FLOAT PT SNGLE LGTH TO FIXED PT HALF ARITH R		A1	146
FDFX	CONVERT FLOAT PT DBLE LGTH TO FIXED PT SINGLE LGTH		A2	147
FXFL	CONVERT FIXED PT SINGLE LGTH TO FLOAT PT SINGLE LGTH		A8	151
FXFD	CONVERT FIXED PT SINGLE LGTH TO FLOAT PT DBLE LGTH		AA	152
FHFL	CONVERT FIXED PT HALF LGTH TO FLOAT PT SINGLE LGTH		A9	153
FHFD	CONVERT FIXED PT HALF LGTH TO FLOAT PT DBLE LGTH		AB	154
FX	NORMALIZE FIXED POINT SINGLE LGTH ARITH REG		AC	155
NFH	NORMALIZE FIXED POINT HALF LGTH ARITH REG		AD	156
VECT	VECTOR	R=1	B0	173
VECTL	VECTOR AFTER LOADING VECTOR FILE	R=0	B0	173

ALPHABETICAL INDEX OF INSTRUCTIONS

ALPHABETICAL INDEX (CONTINUED)

ALPHABETICAL INDEX (CONTINUED)

MNEM CODE	INSTRUCTION	OP CODE	PAGE NO.
IBZ	INCREMENT TEST AND BRANCH ON ZERO ARITH REG	88	124
IBZ	INCREMENT TEST INDEX AND BRANCH ON ZERO	8C	124
IBNZ	INCREMENT TEST AND BRANCH ON NON-ZERO ARITH REG	89	125
IBNZ	INCREMENT TEST INDEX AND BRANCH ON NON-ZERO	8D	125
INT	INTERPRET ARITH REG	92	139
ISE	INCREMENT TEST AND SKIP ON EQUAL ARITH REG	80	128
ISNE	INCREMENT TEST AND SKIP ON NOT EQUAL ARITH REG	81	129
L	LOAD ARITH REG SINGLE LGTH WD	14	26
L	LOAD BASE REG SINGLE LGTH	18	26
L	LOAD INDEX REG OR VECTOR PARAM REG SINGLE LGTH	1C	26
LAC	LOAD ARITH CONDITION	13	49
LAM	LOAD ARITH MASK	12	48
LD	LOAD ARITH REG DBLE LGTH WD	17	33
LEA	LOAD EFFECTIVE ADDRESS INDEX REGISTER	56	138
LEA	LOAD EFFECTIVE ADDRESS INTO BASE REG	52	138
LF	LOAD BASE REG FILE, REG 1-7	1B	45
LF	LOAD BASE REG FILE, REG 8-F	1B	45
LF	LOAD ARITH REG FILE, REG 10-17	1B	45
LF	LOAD ARITH REG FILE, REG 18-1F	1B	45
LF	LOAD INDEX REG FILE, REG 20-27	1B	45
LF	LOAD VECTOR PARAM REG FILE, REG 28-2F	1B	45
LFM	LOAD ALL REG FILES	1F	46
LH	LOAD ARITH REG HALF LGTH WD	15	29
LI	LOAD IMMED INTO ARITH REG SINGLE LGTH	54	27
LI	LOAD IMMED INTO INDEX REG OR VECTOR PARAM REG SNGLE	5C	27
LIH	LOAD IMMED INTO ARITH REG HALF LGTH	55	30
LL	LOAD MEMORY RH WD INTO ARITH REG LH WD	19	32
LLA	LOAD LOCK AHEAD	16	50
LM	LOAD MAG FIXED POINT SINGLE LGTH ARITH REG	3C	34
LMD	LOAD MAG FLOAT POINT DBLE LGTH ARITH REG	3F	37
LMF	LOAD MAG FLOAT POINT SINGLE LGTH ARITH REG	3E	36
LMH	LOAD MAG FIXED POINT HALF LGTH ARITH REG	3D	35
LN	LOAD NEG FIXED POINT SINGLE LGTH (LD 2'S COMP)ARITH R	30	38
LND	LOAD NEG FLOAT POINT DBLE LGTH ARITH REG	33	40
LNF	LOAD NEG FLOAT POINT SINGLE LGTH ARITH REG	32	40
LNH	LOAD NEG FIXED POINT HALF LGTH ARITH REG	31	39
LNM	LOAD NEG MAG FIXED POINT SINGLE LGTH ARITH REG	38	41
LNMD	LOAD NEG MAG FLOAT POINT DBLE LGTH ARITH REG	3B	44
LNMF	LOAD NEG MAG FLOAT POINT SINGLE LGTH ARITH REG	3A	43
LNMH	LOAD NEG FIXED POINT HALF LGTH ARITH REG	39	42
LO	LOAD ARITH REG WITH 1'S COMP SINGLE LGTH	1E	51
LR	LOAD MEMORY RH WD INTO ARITH REG RH WD	1D	31
M	MULTIP BASE REG	68	79
M	MULTIP INDEX OR VECTOR PARAM REG	6A	79
M	MULTIP FIXED POINT SINGLE LGTH ARITH REG	6C	79
MCP	MONITOR CALL AND PROCEED	90	140
MCW	MONITOR CALL AND WAIT	94	140
MF	MULTIP FLOAT POINT SINGLE LGTH ARITH REG	6E	84
MFD	MULTIP FLOAT POINT DBLE LGTH ARITH REG	6F	84
MI	MULTIP IMMED TO BASE REG	78	81
MI	MULTIP IMMED TO INDEX OR VECTOR PARAM REG	7A	81
MI	MULTIP IMMED FIXED POINT SINGLE LGTH ARITH REG	7C	81
MH	MULTIP FIXED POINT HALF LGTH ARITH REG	6D	83
MIH	MULTIP IMMED FIXED POINT HALF LGTH ARITH REG	7D	83
MOD	MODIFY ARITH REG	9F	135

ALPHABETICAL INDEX (CONTINUED)

MNEM CODE	INSTRUCTION		OP CODE	PAGE NO.
NOP	TAKE NEXT INSTRUCTION	R=0	91	115
NFH	NORMALIZE FIXED POINT HALF LGTH ARITH REG		AD	156
NFX	NORMALIZE FIXED POINT SINGLE LGTH ARITH REG		AC	155
CR	CR ARITH REG		E4	90
CRD	CR ARITH REG DBLE LGTH		E5	93
CRI	IMMED CR ARITH REG		F4	90
PSH	PUSH WD ARITH REG		93	134
PUL	PULL WD ARITH REG		97	135
RVS	BIT REVERSAL SINGLE LGTH ARITH REG		C6	107
S	SUBTR FIXED POINT SINGLE LGTH ARITH REG		48	73
SA	ARITH SHIFT FIXED POINT SINGLE LGTH ARITH REG		C0	95
SAD	ARITH SHIFT FIXED POINT DBLE LGTH ARITH REG		C3	98
SAH	ARITH SHIFT FIXED POINT HALF LGTH ARITH REG		C1	97
SC	CIRCULAR SHIFT SINGLE LGTH ARITH REG		CC	103
SCD	CIRCULAR SHIFT DBLE LGTH ARITH REG		CF	106
SCH	CIRCULAR SHIFT HALF LGTH ARITH REG		CD	105
SF	SUBTR FLOAT POINT SINGLE LGTH ARITH REG		4A	75
SFD	SUBTR FLOAT PT DBLE LGTH ARITH REG		4B	75
SH	SUBTR FIXED POINT HALF LGTH ARITH REG		49	74
SI	SUBTR IMMED FIXED POINT SINGLE LGTH ARITH REG		58	73
SIH	SUBTR IMMED FIXED POINT HALF LGTH ARITH REG		59	74
SL	LOGICAL SHIFT SINGLE LGTH ARITH REG		C4	99
SLD	LOGICAL SHIFT DBLE LGTH ARITH REG		C7	102
SLH	LOGICAL SHIFT HALF LGTH ARITH REG		C5	101
SM	SUBTR MAG FIXED POINT SINGLE LGTH ARITH REG		4C	76
SMF	SUBTR MAG FLOAT POINT SINGLE LGTH ARITH REG		4E	78
SMFD	SUBTR MAG FLOAT POINT DBLE LGTH ARITH REG		4F	78
SMH	SUBTR MAG FIXED POINT HALF LGTH ARITH REG		4D	77
[illegible]	ONE'S COMPLEMENT SINGLEWORD		2F	61
SIOH	ONE'S COMPLEMENT HALFWORD		2A	62
SPS	STORE PROGRAM STATUS WORD		22	57
ST	STORE ARITH REG, SINGLE LGTH		24	52
ST	STORE BASE REG, SINGLE LGTH		28	52
ST	STORE INDEX REG OR VECTOR PARAM REG, SINGLE LGTH		2C	52
STD	STORE ARITH REG, DBLE LGTH		27	56
STF	STORE BASE REG FILE, REG 1-7		2B	63
STF	STORE BASE REG FILE, REG 8-F		2B	63
STF	STORE ARITH REG FILE, REG 10-17		2B	63
STF	STORE ARITH REG FILE, REG 18-1F		2B	63
STF	STORE INDEX REG FILE, REG 20-27		2B	63
STF	STORE VECTOR PARAM REG FILE, REG 28-2F		2B	63
STFM	STORE ALL REG FILES, REG 1-2F		2F	64
STH	STORE HALF LGTH, ARITH REG		25	53
STL	STORE REG LH INTO MEMORY RH, ARITH REG		29	55
STN	STORE NEG FIXED POINT SINGLEWORD		34	59
STND	STORE NEG FLOAT POINT DOUBLEWORD		37	61
STNF	STORE NEG FLOAT POINT SINGLEWORD		36	60
STNH	STORE NEG FIXED POINT HALFWORD		35	60
STR	STORE REG RH INTO MEMORY RH, ARITH REG		2D	54
STZ	STORE ZERO, SINGLE LGTH		20	57
STZD	STORE ZERO, DBLE LGTH		23	58
STZH	STORE ZERO, HALF LGTH		21	58
VECT	VECTOR	R=1	B0	173
[illegible]TL	VECTOR AFTER LOADING VECTOR FILE	R=0	B0	173
XCH	EXCHANGE ARITH REG		1A	47
XEC	EXECUTE		96	139
XOR	EXCLUSIVE OR ARITH REG		E8	91
XORD	EXCLUSIVE OR ARITH REG DBLE LGTH		F9	94
XORI	IMMED EXCLUSIVE OR ARITH REG		F8	91

OP CODE INDEX OF INSTRUCTIONS

OP CODE INDEX (CONTINUED)

OP CODE	MNEM CODE	INSTRUCTION		PAGE NO.
8B	DBNZ	DECREMENT TEST AND BRANCH ON NON-ZERO ARITH REG		127
8C	IBZ	INCREMENT TEST INDEX AND BRANCH ON ZERO		124
8D	IBNZ	INCREMENT TEST INDEX AND BRANCH ON NON-ZERO		125
8E	DBZ	DECREMENT TEST INDEX AND BRANCH ON ZERO		126
8F	DBNZ	DECREMENT TEST INDEX AND BRANCH ON NON-ZERO		127
90	MCP	MONITOR CALL AND PROCEED		140
91	BCC	BRANCH ON COMPARE CODE		115
91	NOP	TAKE NEXT INSTRUCTION	R=0	115
91	BE	(R) EQ (ALPHA)	R=1	115
91	BG	(R) GR (ALPHA)	R=2	115
91	BGE	(R) GR OR EQ (ALPHA)	R=3	115
91	BL	(R) LS (ALPHA)	R=4	115
91	BLE	(R) LS OR EQ (ALPHA)	R=5	115
91	BNE	(R) NOT EQ (ALPHA)	R=6	115
91	B	UNCONDITIONAL BRANCH	R=7	115
91	BCZ	ALL BITS ARE ZERO	R=1	118
91	BCO	ALL BITS ARE ONE	R=2	118
91	BCNM	NOT MIXED	R=3	118
91	BCM	MIXED ZEROS AND ONES	R=4	118
91	BCNO	NOT ALL ONES	R=5	118
91	BCNZ	NOT ALL ZEROS	R=6	118
92	INT	INTERPRET ARITH REG		139
93	PSH	PUSH WD ARITH REG		134
94	MCW	MONITOR CALL AND WAIT		140
95	BRC	BRANCH ON RESULT CODE		119
95	BZ	(R) EQ ZERO	R=1	119
95	BPL	(R) GR ZERO	R=2	119
95	BZP	(R) GR OR EQ ZERO	R=3	119
95	BMI	(R) LS ZERO	R=4	119
95	BZM	(R) LS OR EQ ZERO	R=5	119
95	BNZ	(R) NOT EQ ZERO	R=6	119
95	BLR	BRANCH ON LOGICAL RESULT		120
95	BRZ	ALL BITS ARE ZERO		120
95	BRO	ALL BITS ARE ONE		120 *
95	BRNM	NOT MIXED		120
95	BRM	MIXED ZEROS AND ONES		120
95	BRNO	NOT ALL ONES		120
95	BRNZ	NOT ALL ZEROS		120 *
96	XEC	EXECUTE		139
97	PUL	PULL WD ARITH REG		135
98	BLB	BRANCH AND LOAD REG WITH PC		136
99	BLX	BRANCH AND LOAD INDEX REG OR VECTOR PARAM REG		137
9C	BXEC	BRANCH ON EXECUTE BRANCH CONDITION TRUE	R=1 OR ODD	123
9C	BAE	BRANCH ON ARITHMETIC EXCEPTION		121
9C	BU	FLOAT PT EXP UNDERFLOW	R=1	121
9D	BO	FLOAT PT EXP OVERFLOW	R=2	121
9C	BUO	FLOAT PT EXP UNDERFLOW OR OVERFLOW	R=3	121
9C	BX	FIXED PT OVERFLOW	R=4	121
9D	BXU	FIXED PT OVERFLOW OR FLOAT EXP UNDERFLOW	R=5	121
9C	BXO	FIXED PT OVERFLOW OR FLOAT EXP OVERFLOW	R=6	121
9C	BXUO	FIXED PT OVR OR FLOAT PT OVR OR UNDERFLOW	R=7	121
9C	BC	DIVIDE CHECK	R=8	121
9D	BCU	DIVIDE CHECK OR FLOAT PT EXP UNDERFLOW	R=9	121
9C	BCO	DIVIDE CHECK OR FLOAT PT EXP OVERFLOW	R=A	121
9C	BCUO	DIVIDE CHECK OR FLOAT PT EXP UNDER OR OVERFLOW	R=B	121

VECTOR SEQUENTIAL INDEX OF INSTRUCTIONS

MNEM CODE	INSTRUCTION	OP CODE	PAGE NO.
VA	VECTOR ADD FIXED POINT SINGLE LGTH	40	174
VAH	VECTOR ADD FIXED POINT HALF LGTH	41	174
VAF	VECTOR ADD FLOAT POINT SINGLE LGTH	42	174
VAFD	VECTOR ADD FLOAT POINT DBLE LGTH	43	174
VAM	VECTOR ADD MAG FIXED POINT SINGLE LGTH	44	174
VAMH	VECTOR ADD MAG FIXED POINT HALF LGTH	45	174
VAMF	VECTOR ADD MAG FLOAT POINT SNGLE LGTH	46	174
VAMFD	VECTOR ADD MAG FLOAT POINT DBLE LGTH	47	174
VS	VECTOR SUBTR FIXED POINT SINGLE LGTH	48	174
VSH	VECTOR SUBTR FIXED POINT HALF LGTH	49	174
VSF	VECTOR SUBTR FLOAT POINT SINGLE LGTH	4A	174
VSFD	VECTOR SUBTR FLOAT POINT DBLE LGTH	4B	174
VSM	VECTOR SUBTR MAG FIXED POINT SINGLE LGTH	4C	175
VSMH	VECTOR SUBTR MAG FIXED POINT HALF LGTH	4D	175
VSMF	VECTOR SUBTR MAG FLOAT POINT SINGLE LGTH	4E	175
VSMFD	VECTOR SUBTR MAG FLOAT POINT DBLE LGTH	4F	175
VM	VECTOR MULT FIXED POINT SINGLE LGTH	6C	175
VMH	VECTOR MULT FIXED POINT HALF LGTH	6D	175
VMF	VECTOR MULT FLOAT POINT SINGLE LGTH	6E	175
VMFD	VECTOR MULT FLOAT POINT DBLE LGTH	6F	175
VDP	VECTOR DOT PRODUCT FIXED POINT SINGLE LGTH	68	175
VDPH	VECTOR DOT PRODUCT FIXED POINT HALF LGTH	69	175
VDPF	VECTOR DOT PRODUCT FLOAT POINT SINGLE LGTH	6A	175
VDPFD	VECTOR DOT PRODUCT FLOAT POINT DBLE LGTH	6B	175
VD	VECTOR DIVIDE FIXED POINT SINGLE LGTH	64	176
VDH	VECTOR DIVIDE FIXED POINT HALF LGTH	65	176
VDF	VECTOR DIVIDE FLOAT POINT SINGLE LGTH	66	176
VDFD	VECTOR DIVIDE FLOAT POINT DBLE LGTH	67	176
VAND	VECTOR LOGICAL AND SINGLE LGTH	E0	178
VOR	VECTOR LOGICAL OR SINGLE LGTH	E4	178
VXOR	VECTOR LOGICAL EXCLUSIVE OR SINGLE LGTH	E8	178
VEQC	VECTOR LOGICAL EQUIVALENCE SINGLE LGTH	EC	178
VANDD	VECTOR LOGICAL AND DBLE LGTH	E1	178
VORD	VECTOR LOGICAL OR DBLE LGTH	E5	178
VXORD	VECTOR EXCLUSIVE OR DBLE LGTH	E9	178
VEQCD	VECTOR EQUIVALENCE DBLE LGTH	ED	178
VSA	VECTOR ARITH SHIFT FIXED POINT SINGLE LGTH	C0	179
VSAH	VECTOR ARITH SHIFT FIXED POINT HALF LGTH	C1	179
VSAD	VECTOR ARITH SHIFT FIXED POINT DBLE LGTH	C3	179
VSL	VECTOR LOGICAL SHIFT SINGLE LGTH	C4	179
VSLH	VECTOR LOGICAL SHIFT HALF LGTH	C5	179
VSLD	VECTOR LOGICAL SHIFT DBLE LGTH	C7	179
VSC	VECTOR CIRCULAR SHIFT SINGLE LGTH	CC	179
VSCH	VECTOR CIRCULAR SHIFT HALF LGTH	CD	179
VSCD	VECTOR CIRCULAR SHIFT DBLE LGTH	CF	179
VMGH	VECTOR MERGE HALFWDS	D8	180
VMG	VECTOR MERGE SNGLE WDS	D9	180
VMGD	VECTOR MERGE DBLE WDS	DB	180
VO	VECTOR ORDER SINGLEWDS FIXED PT	D4	181
VOD	VECTOR ORDER HALF WDS FIXED PT	D5	181
VOF	VECTOR ORDER SINGLEWDS FLOAT PT	D6	181
VOFD	VECTOR ORDER DBLE WDS FLOAT PT	D7	181

VECTOR SEQUENTIAL INDEX (CONTINUED)

MNEM CODE	INSTRUCTION	OP CODE	PAGE NO.
VC	VECTOR ARITH COMP FIXED PT SINGLE LGTH	D0	183
VCH	VECTOR ARITH COMP FIXED PT HALF LGTH	D1	183
VCF	VECTOR ARITH COMP FLOAT PT SINGLE LGTH	D2	183
VCFD	VECTOR ARITH COMP FLOAT PT DBLE LGTH	D3	183
VCAND	VECTOR LOGICAL COMP USING AND SINGLE LGTH	E2	185
VCANDD	VECTOR LOGICAL COMP USING AND DBLE LGTH	E3	185
VCOR	VECTOR LOGICAL COMP USING OR SINGLE LGTH	E6	185
VCORD	VECTOR LOGICAL COMP USING OR DBLE LGTH	E7	185
VL	VECTOR SRCH FOR LGST ARITH ELEMENT FIXED PT SINGLE LGTH	50	187
VLH	VECTOR SRCH FOR LGST ARITH ELEMENT FIXED PT HALF LGTH	51	187
VLF	VECTOR SRCH FOR LGST ARITH ELEMENT FLOAT PT SINGLE LGTH	52	187
VLFD	VECTOR SRCH FOR LGST ARITH ELEMENT FLOAT PT DBLE LGTH	53	187
VLM	VECTOR SRCH FOR LGST MAG FIXED PT SINGLE LGTH	54	187
VLMH	VECTOR SRCH FOR LGST MAG FIXED PT HALF LGTH	55	187
VLMF	VECTOR SRCH FOR LGST MAG FLOAT PT SINGLE LENGTH	56	187
VLMFD	VECTOR SRCH FOR LGST MAG FLOAT PT DBLE LGTH	57	187
VSS	VECTOR SRCH FOR SMLST ARITH ELEMENT FIXED PT SNGLE LGTH	58	187
VSSH	VECTOR SRCH FOR SMLST ARITH ELEMENT FIXED PT HALF LGTH	59	187
VSSF	VECTOR SRCH FOR SMLST ARITH ELEMENT FLOAT PT SNGLE LGTH	5A	187
VSSFD	VECTOR SRCH FOR SMLST ARITH ELEMENT FLOAT PT DBLE LGTH	5B	187
VSSM	VECTOR SRCH FOR SMLST MAG FIXED PT SINGLE LGTH	5C	187
VSSMH	VECTOR SRCH FOR SMLST MAG FIXED PT HALF LGTH	5D	187
VSSMF	VECTOR SRCH FOR SMLST MAG FLOAT PT SINGLE LGTH	5E	187
VSSMFD	VECTOR SRCH FOR SMLST MAG FLOAT PT DBLE LGTH	5F	187
VPP	VECTOR PEAK FIXED PT SINGLE LGTH	DC	188
VPPH	VECTOR PEAK FIXED PT HALF LGTH	DD	188
VPPF	VECTOR PEAK FLOAT PT SINGLE LGTH	DE	188
VPPFD	VECTOR PEAK FLOAT PT DBLE LGTH	DF	188
VFLFX	VECTOR CNVRT FLOAT PT SNGLE LGTH TO FIXED PT SNGLE LGTH	A0	190
VFLFH	VECTOR CNVRT FLOAT PT SNGLE LGTH TO FIXED PT HALF LGTH	A1	191
VFDFX	VECTOR CNVRT FLOAT PT DBLE LGTH TO FIXED PT SNGLE LGTH	A2	191
VFXFL	VECTOR CNVRT FLOAT PT SNGLE LGTH TO FLOAT PT SNGLE LGTH	A8	192
VFXFD	VECTOR CNVRT FLOAT PT SNGLE LGTH TO FLOAT PT DBLE LGTH	AA	193
VFHFL	VECTOR CNVRT FIXED PT HALF LGTH TO FLOAT PT SNGLE LGTH	A9	193
VFHFD	VECTOR CNVRT FIXED PT HALF LGTH TO FLOAT PT DBLE LGTH	AB	194
VNFX	VECTOR NORMALIZE FIXED PT SINGLE LGTH	AC	195
VNFH	VECTOR NORMALIZE FIXED PT HALF LGTH	AD	195

VECTOR ALPHABETICAL INDEX OF INSTRUCTIONS

VECTOR OP CODE INDEX OF INSTRUCTIONS

VECTOR OP CODE INDEX (CONTINUED)

EXAMPLES OF VECTOR INSTRUCTION APPLICATION

TABLE OF CONTENTS

GENERAL

A set of examples will serve to illustrate the vector loop feature of the ASC. The example problems are: (1) matrix-vector multiplication, (2) matrix multiplication, (3) matrix transpose, (4) fixed filter, (5) fixed filter and decimate, and (6) interpolate.

MATRIX-VECTOR MULTIPLICATION

The data array for matrix [A] is stored consecutively by rows and consecutively within rows, i.e. the first element of one row is stored in the location following the last element of the previous row.

The data array for vector $\vec{B}$ is stored in consecutive locations.

Given: matrix [A] of dimension K by L

and vector $\vec{B}$ of dimension L

Find: $\vec{C} = [A] \quad \vec{B}$

where vector $\vec{C}$ is of dimension K and

element $c_i = \sum_{j=1}^{L} a_{ij} \cdot b_j \quad$ for $1 \leq i \leq K$

$$\begin{Bmatrix} c_1 \\ c_2 \\ \cdot \\ \cdot \\ \cdot \\ c_K \end{Bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1L} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2L} \\ \cdot & \cdot & \cdot & & \cdot \\ \cdot & \cdot & \cdot & & \cdot \\ \cdot & \cdot & \cdot & & \cdot \\ a_{K1} & a_{K2} & a_{K3} & \cdots & a_{KL} \end{bmatrix} \begin{Bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_L \end{Bmatrix}$$

Solution: Each element, c_i, of vector $\vec{C}$ is the result of a vector dot product (VDP) operation involving the i^{th} row of matrix [A] and the column vector $\vec{B}$.

A matrix-vector product may be programmed on the ASC by issuing a vector instruction with the following set of vector parameters:

ØPR ~ Vector dot product command

SV = 0 or 8 depending on whether a single or double length fixed point result is desired.

L = L Vector dimension.

XA = XB = XC = 0 No initial index for vectors A, B, & C.

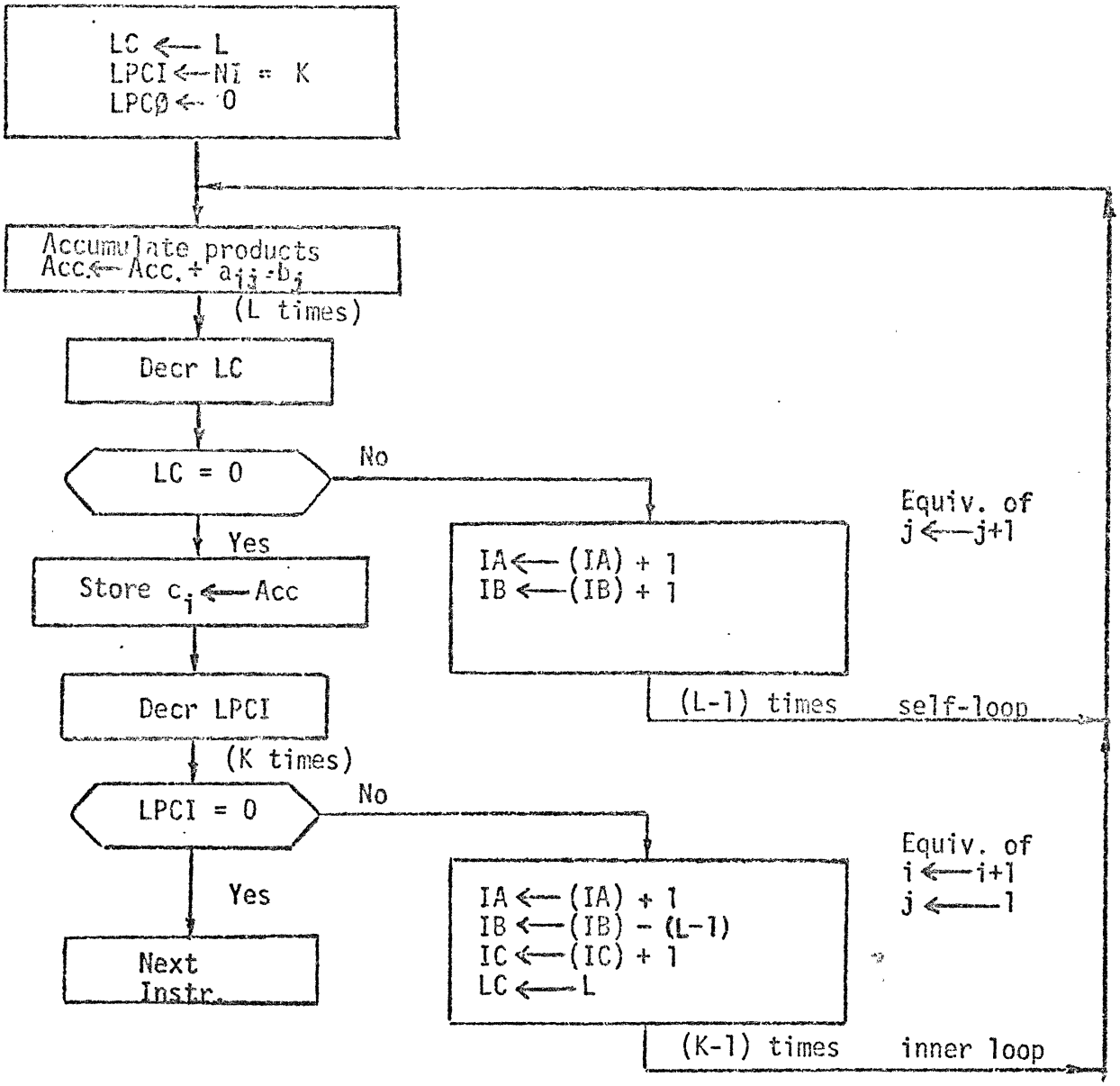


Figure 1. Flow chart for matrix-vector multiplication.

SAA ~ Starting address of matrix [A] (address of element a_{11})

SAB ~ Starting address of vector $\vec{B}$ (address of element b_1)

SAC ~ Starting address of result vector $\vec{C}$ (address of result element c_1)

VI = 0 A VI field equal to zero indicates positive incrementing of all vector addresses by unity during the self-loop.

DAI = 1 ΔA_i increment for inner loop

Advances the "A" address in the index unit to the address of the first element of the next row of matrix [A] from the address of the last element of the current row.

DBI = -(L-1) ΔB_i increment for inner loop

Returns the "B" address in the index unit to the starting address of vector $\vec{B}$ from the address of the last element of vector $\vec{B}$. DBI is equal to the number of backspaces required to re-establish the initial address of vector $\vec{B}$ for the next VDP operation involving the next row of matrix [A] and vector $\vec{B}$.

Note that the A & B addresses are incremented (by unity) L-1 times during the self-loop as shown in figure 1.

DCI = 1 ΔC_i for inner loop

Advances the storage address to the next location for the subsequent VDP operation.

Note that the C address is not incremented during the self-loop of a VDP operation, since a VDP generates a scalar result.

Also, the elements of vector $\vec{C}$ may be spaced any number of addresses apart up to $2^{15} - 1 = 32,767$ by inserting the value of the desired spacing interval into the DCI field.

NI = K Inner loop count

For this example, K is the number of rows of matrix [A], which also determines the number of elements of result vector $\vec{C}$. The operation is completed when all K rows of [A] have been processed.

MATRIX MULTIPLICATION, EXAMPLE A

The data array for matrix [A] is stored consecutively by rows, i.e. the first element of one row is stored in the location following the last element of the previous row.

The data array for matrix [B] is stored consecutively by columns, i.e. the first element of one column is stored in the location following the last element of the preceeding column (the column on its left).

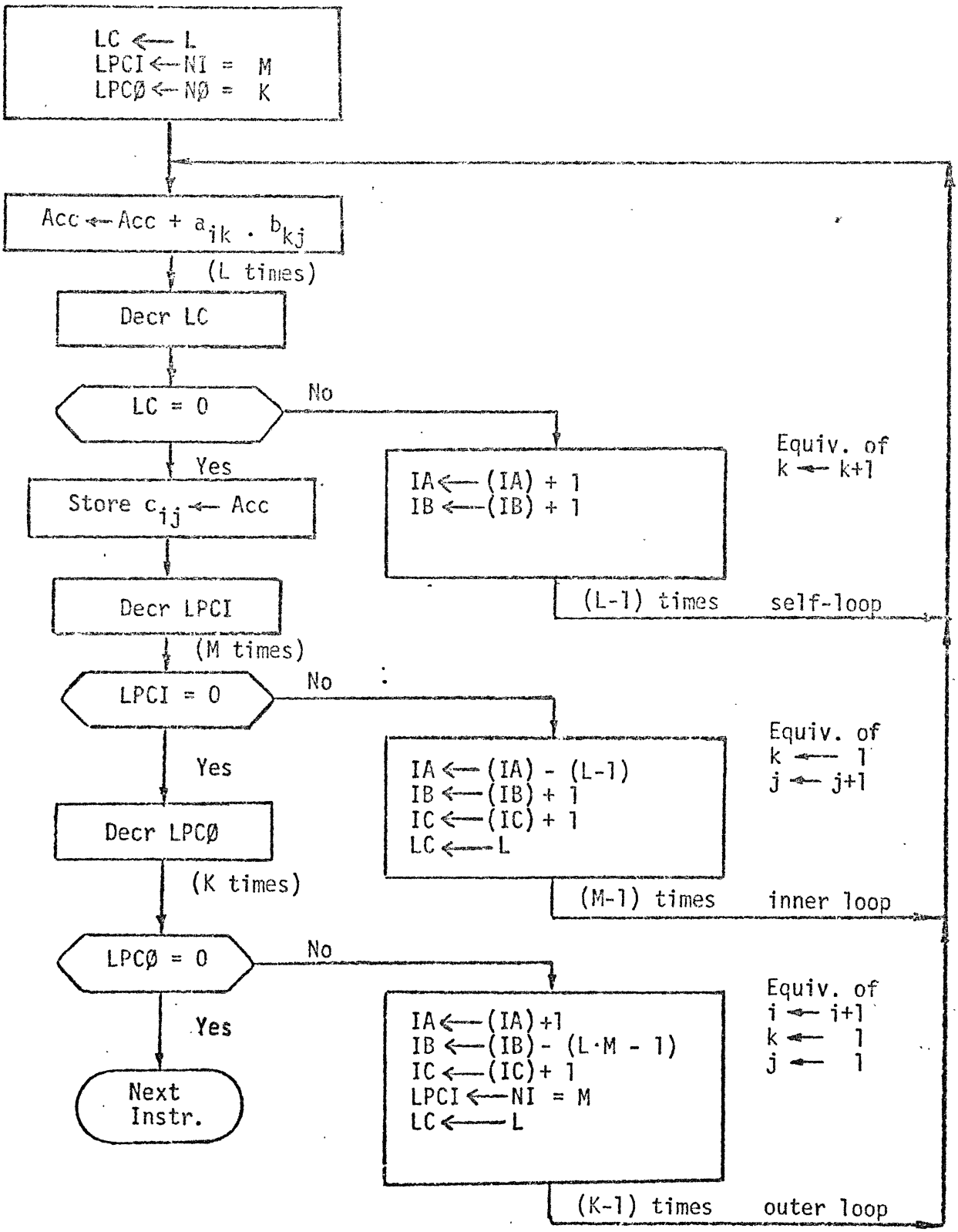


Figure 2. Flow chart for matrix multiplication of example A

Given: matrix [A] of dimension K by L

and matrix [B] of dimension L by M

Find: matrix [C] of dimension K by M,

where [C] = [A] [B]

such that element $c_{ij} = \sum_{k=1}^{L} a_{ik} \cdot b_{kj}$

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1L} \\ a_{21} & a_{22} & \cdots & a_{2L} \\ \vdots & \vdots & & \vdots \\ a_{K1} & a_{K2} & \cdots & a_{KL} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1M} \\ b_{21} & b_{22} & \cdots & b_{2M} \\ \vdots & \vdots & & \vdots \\ b_{L1} & b_{L2} & \cdots & b_{LM} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1M} \\ c_{21} & c_{22} & \cdots & c_{2M} \\ \vdots & \vdots & & \vdots \\ c_{K1} & c_{K2} & \cdots & c_{KM} \end{bmatrix}$$

Solution: Each element, c_{ij}, of matrix [C] is the result of a vector dot product operation involving the i^{th} row of matrix [A] and the j^{th} column of matrix [B].

Vector parameter set for matrix multiplication:

ØPR ~ Vector dot product command

SV = 0 or 8

L = L Vector dot product length

XA = XB = XC = 0 No initial index for vectors AB, & C.

SAA ~ Starting address of matrix [A] (address of element a_{11})

SAB ~ Starting address of matrix [B] (address of element b_{11})

SAC ~ Starting address of result matrix [C] (address of result element c_{11})

VI = 0 Positive incrementing of all vector addresses by unity during self-loop.

DAI = -(L-1) Number of backspaces to return to the first element of the current row of matrix [A] from the address of the last element of the current row.

DBI = 1 — Advance to the next column of matrix [B] from the address of the last element of the current column.

DCI = 1 — Advance the storage address to the next location for the subsequent VDP operation. Result matrix [C] is generated by rows.

NI = M — Inner loop count

For this example, M is the number of columns of matrix [B]. This method computes all products of the columns of [B] with the first row of [A], before advancing to the next row of [A].

DAØ = 1 — ΔA_0 increment for outer loop

Advance to the next row of [A] from the address of the last element of the previous row.

DBO = -(L·M-1) — ΔB_0 increment for outer loop

Return to the starting address of matrix [B] from the address of the last element, b_{LM}.

DCØ = 1 — ΔC_0 increment for outer loop

Advance the storage address to the next row of result matrix [C] from the address of the last element of the previous row.

NØ = K — Outer loop count

For this example, K is the number of rows of matrix [A]. The function of the outer loop is to advance to the next row of [A] before continuing with the processing of the inner loop. The operation is completed when all K rows of [A] have been processed. See Figure 2 for the flow chart describing matrix multiplication.

MATRIX MULTIPLICATION, EXAMPLE B

The result matrix [C] could just as easily have been generated a column at a time rather than a row at a time as previously described. This would entail modification of the increment and loop count information as follows:

DAI = 1 — Advance the address in index unit "A" to the address of the next row of matrix [A] from the address of the last element of the current row.

DBI = -(L-1) — Return the address in index unit "B" to the address of the first element of the current column of matrix [B] from the address of the last element of the current column.

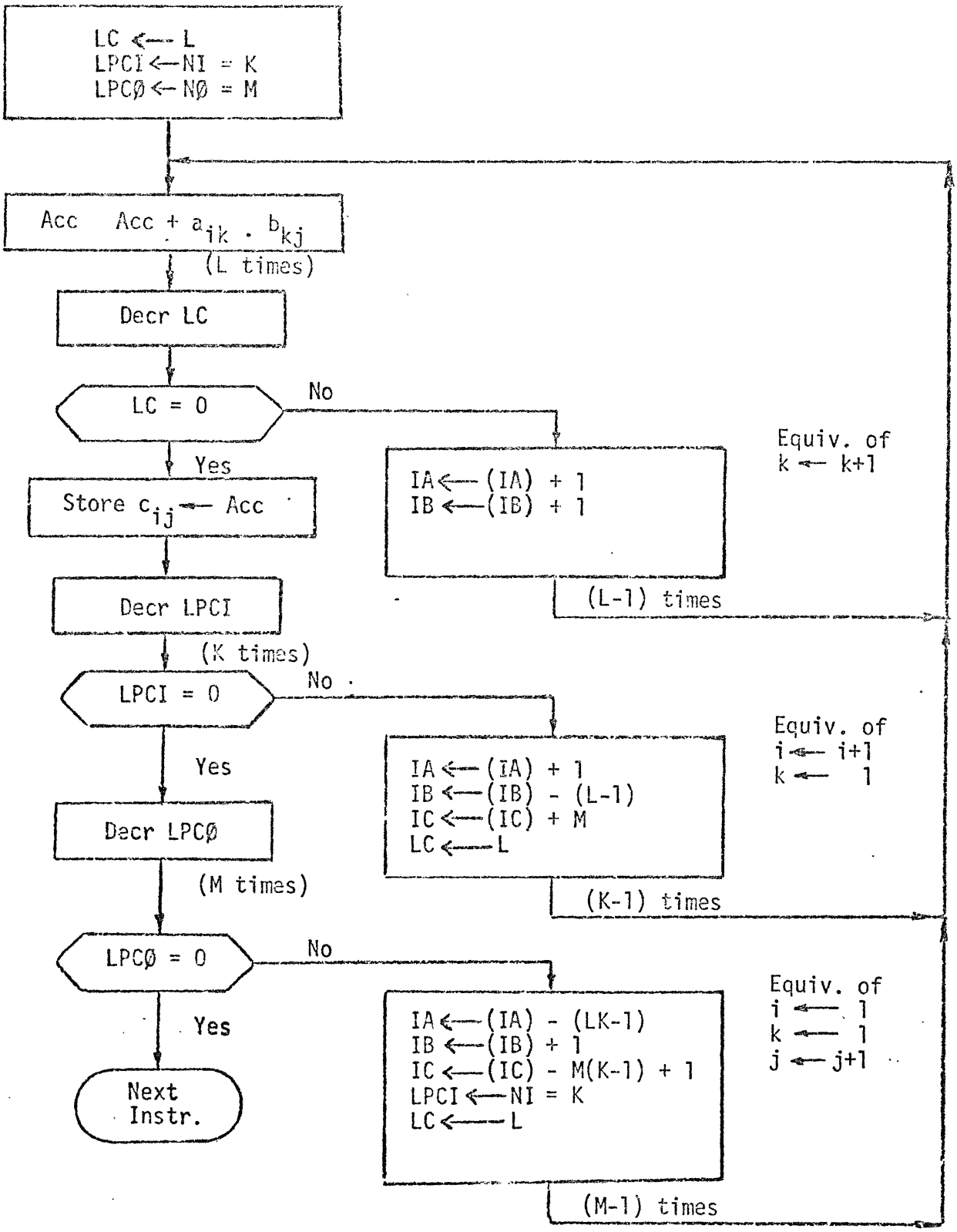


Figure 3. Flow chart for matrix multiplication in which the result is generated by columns instead of by rows as in example B

DCI = M — The DCI field specifies the number of spaces to advance the storage address. For this example, M-1 storage locations are skipped over. This has the effect of storing results by columns instead of by rows.

NI = K — Inner loop count

K is the number of rows of matrix [A]. The inner loop consists of multiplying a particular column of matrix [B] times all K rows of matrix [A], resulting in one column of [C].

DAØ =-(L:K-1) — Return to the a_{11} element of [A] from the address of the last element, a_{KL}.

DBØ = 1 — Advance to the top element of the next column of matrix [B] from the address of the bottom element of the current column.

DCØ = -M(K-1)+1 — Advance to the storage address of the first element in the next column of result matrix [C] from the last element of the current column.

NØ = M — Outer loop count

M is the number of columns of matrix [B]. The outer loop function in this example is to advance to the next column of [B] and to return to the start of [A] before continuing with the processing of the inner loop. The operation is completed when all M columns of [B] have been processed. See Figure 3 for the flow chart of this matrix multiplication procedure.

Now, suppose the result matrix [C] of a previous matrix multiplication is to be used as the [B] matrix in a subsequent matrix multiplication. Since the columns of [B] are multiplied by the rows of [A] using a vector dot product operation, the elements of the columns of [B] must be stored in consecutive memory locations for efficient processing of the column vectors. The data of matrix [B] is not ordered in this manner if it is the result of some previous matrix multiplication. The "transpose" of the data array (in memory) for matrix [B] is required before initiating any matrix multiplication involving [B]. Such a "transposition" can be accomplished separately (example 3) or in conjunction with the previous matrix multiplication. To perform "transposition" in conjunction with a matrix multiplication it is only necessary to modify the DCI and DCØ parameters in the vector parameter file of example 2b. The modification involves inserting the value of unity in place of M in the DCI field and inserting the value of unity in place of -M(K-1) + 1 in the DCØ field. This change has the effect of storing the results of the matrix multiplication by rows instead of by columns as was done in example 2b. The resulting matrix $[C]^T$ is then the transpose of [C].

MATRIX TRANSPOSE

Matrix transposition involves moving the element in position i, j of a data array into position j, i, where the first symbol designates row position and the second symbol designates column position. What effectively happens is that rows become columns and columns become rows. In terms of memory addresses, the column elements of a K by M dimension matrix, [A], are initially M locations apart, i.e. with M-1 storage locations between any two adjacent column elements. While the row elements of [A] are stored in consecutive memory locations and the first element of any row (except the first row) is stored in the location following the last element of the previous row.

After matrix transposition the original column elements will be stored in consecutive locations while the original row elements will be stored K locations apart. The resulting transposed matrix, $[A]^T$, is of dimension M by K.

Matrix transposition may be programmed on the ASC by issuing a vector instruction with the following vector parameters:

ØPR ~	VLØR, Vector logical ØR.
SV = 7	Vector $\vec{A}_i$ directly addressed.
	Immediate operand single-valued vector $\vec{B}$.
(2A) = 0	Immediate operand from the contents of register 2A is zero.
L = 1	Vector dimension equals one.
XA = XB = XC = 0	No initial index for vectors A, B, & C.
SAA	Starting address of matrix [A] (address of element a_{11})
SAB	Starting address B is not used when SV = 7.
SAC	Starting address of result matrix [C] (address of element c_{11})
VI = 0	Positive incrementing of vector addresses A, B, & C by unity during self-loop.
DAI = 1	Advance the address of [A] to the next element on the current row.
DBI = 0	Not used.
DCI = K	Advance the storage address by K so that the j^{th} element of the current row is stored into the (j-1)K location of the current column.

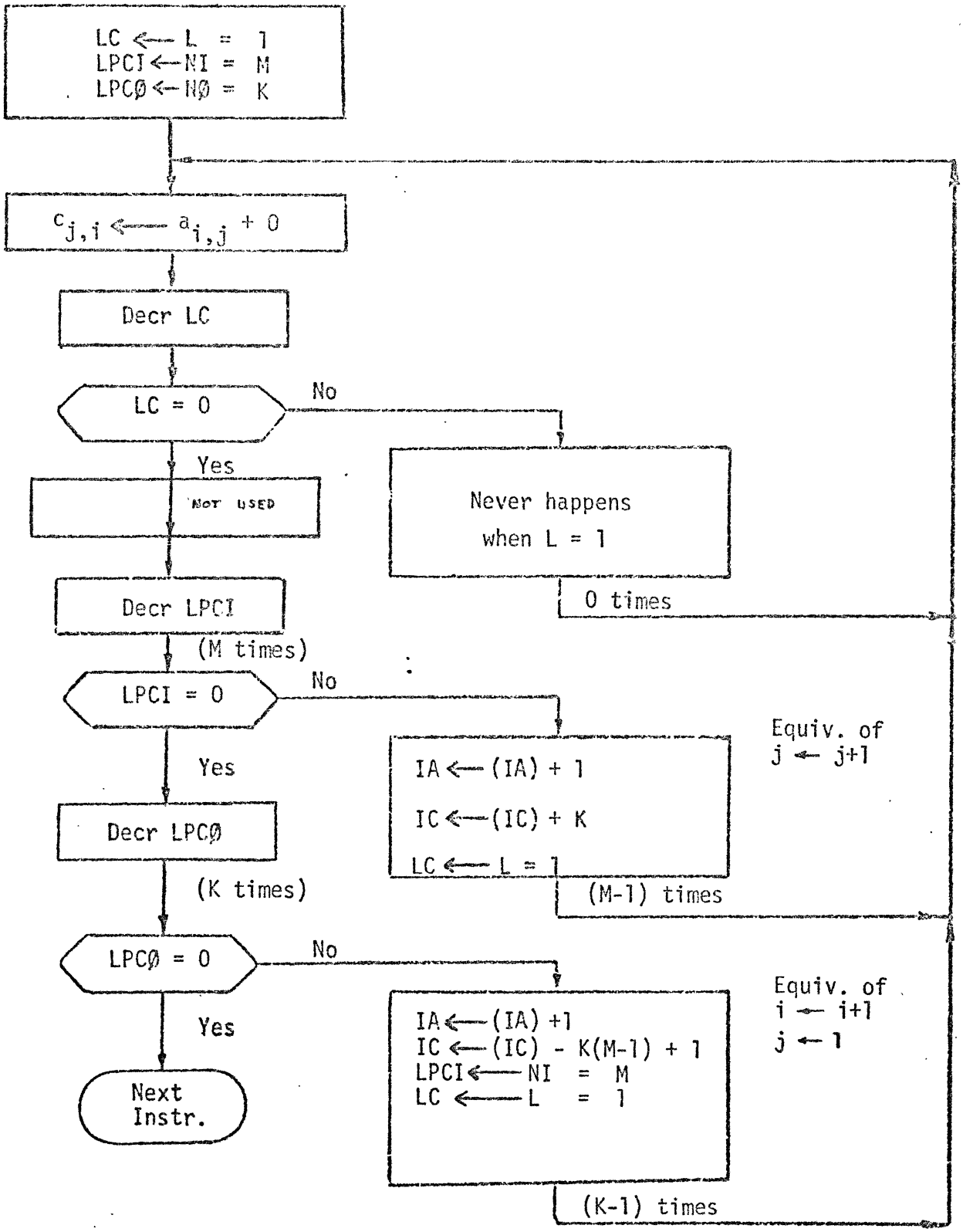


Figure 4. Flow chart for matrix transposition.

NI = M — M is the number of elements in each row of [A]. The inner loop is completed when the current row of matrix [A] has been transposed into a column of result matrix [C].

DAØ = 1 — Advance to the first element of the next row of matrix [A] from the address of the last element in the current row.

DBØ = 0 — Not used.

DCØ = -K(M-1)+1 — Advance to the storage address of the first element in the next column of result matrix [C] from the address of the last element in the current column.

NØ = K — K is the number of rows of matrix [A]. The outer loop is completed when all K rows of [A] have been converted into K columns of result matrix [C]. The flow chart for matrix transposition is shown in figure 4.

Note that a vector logical ØR (VLØR) instruction is used because it requires only one clock time in the arithmetic unit. An immediate operand of zero for the single-valued vector is used with the vector logical ØR instruction, so that the data is not affected by being moved to a new area of memory although the data array has been transposed.

Alternates for this instruction are a vector logical AND with an immediate operand of all "ones" (1 clock time) or a vector add with an immediate operand of zero (2 clock times) or a vector multiply with an immediate operand of one (3 clock times) etc.

FIXED FILTER

The fixed filter operation is described by the formula:

$$c_k = \sum_{i=0}^{L-1} a_{i+k} \cdot b_i \qquad \text{for } 0 \leq k \leq N-L$$

where the input trace consists of N data points, denoted by a_j $(0 \leq j \leq N-1)$ and the fixed filter is represented by vector $\vec{B}$ with components b_i, $(0 \leq i \leq L-1)$.

c_k represents an output point which results when filter $\vec{B}$ is applied to input trace $\vec{A}$ shifted by k.

This operation is programmed on the ASC by issuing a vector instruction with the following vector parameters:

ØPR ~ — Vector dot product

SV = 0 or 8 — Both vectors directly addressed

XA = XB = XC = 0 — No initial index for vectors A, B, & C

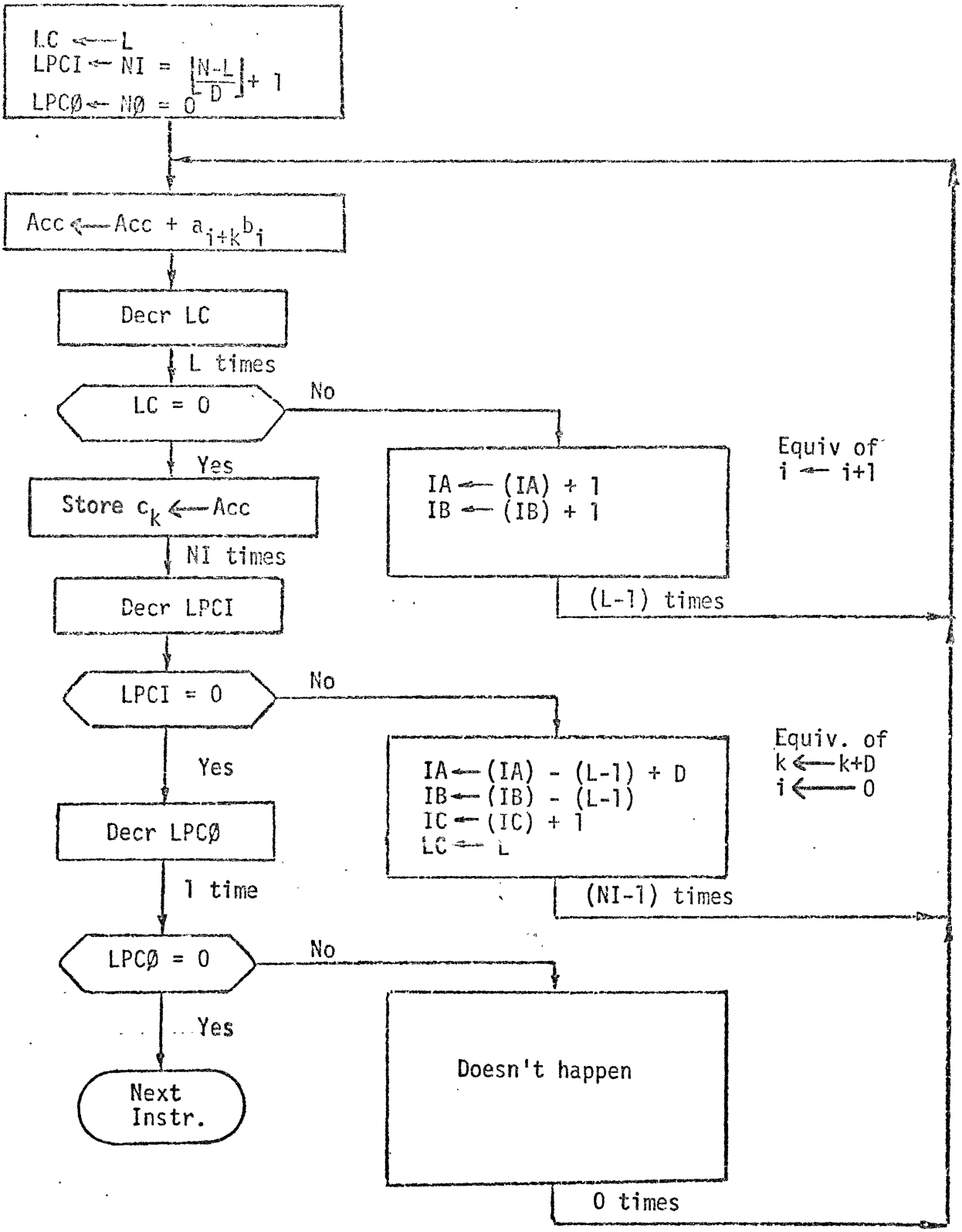


Figure 5. Flow chart for (a) fixed filter operation when D = 1 (b) fixed filter and decimate when D ≠ 1.

SAA ~ Starting address of input trace, $\vec{A}$. (address of data point a_0).

SAB ~ Starting address of fixed filter, $\vec{B}$ (address of filter point b_0).

SAC ~ Starting address of output trace, $\vec{C}$. (address of output point c_0).

VI = 0 Positive incrementing of vector addresses by unity during self-loop.

DAI = -(L-1)+1 Returns input trace to the current starting point (element a_k) plus one. Effectively moves filter along the input trace.

DBI =-(L-1) Returns filter address to starting point from last filter point.

DCI = 1 Advances output storage address by one for next VDP operation.

NI = N-L+1 Determines the number of output points. The fixed filter operation is complete when N-L+1 output points have been computed. See Figure 5 for the flow chart of a fixed filter operation.

FIXED FILTER AND DECIMATE

The fixed filter and decimate operation is described by the formula:

$$c_k = \sum_{i=0}^{L-1} a_{i+kD} \cdot b_i \qquad \text{for } 0 \le k \le \left\lfloor \frac{N-L}{D} \right\rfloor + 1$$

This operation requires modification of only two parameters from those listed in example 4. The DAI and NI parameters must be modified as follows:

DAI = -(L-1)+D Returns the input trace to the current starting point (element a_{kD}) plus D. Effectively moves the filter along the input trace at intervals spaced D apart. D-1 output points are "decimated" between each output point which is stored. These D-1 output points are <u>not</u> computed.

NI = $\left\lfloor \frac{N-L}{D} \right\rfloor$ +1 Determines the number of output points. The number $\left\lfloor \frac{N-L}{D} \right\rfloor$ is the nearest integral number below N-L/D, if this value is not an integer. The fixed filter and decimate operation is complete when $\lfloor N-L/D \rfloor$ +1 output points have been computed. See Figure 5 for the flow chart of this operation.

INTERPOLATION

Interpolation is described by the formula:

$$c_{mD+p} = \sum_{i=0}^{L-1} a_{i+m} \cdot b_{i+pL}$$

for $0 \leq m \leq N-L$

and $0 \leq p \leq D-1$

For this operation a set of filters must be stored in memory such that the first filter point of one filter is stored in the location following the last filter point of the previous filter. These filters may be viewed as column vectors of matrix [B]. There are D fixed filters all of length L, so the [B] matrix is of dimension L by D. In general, the 0-th filter in the first column has the values 1, 0, 0, ..., 0 stored in L consecutive locations. This filter will transfer the input trace to the output region of memory, storing each input point into locations spaced D addresses apart.

The D-1 locations between stored output points are reserved for insertion of the interpolated points. The other D-1 filters in matrix [B] are used for interpolation.

The interpolation operation is equivalent to the application of D fixed filters to the input trace, with storage arranged in such a manner that the D output traces formed are merged.

The vector parameter list for interpolation is as follows:

ØPR ~ Vector dot product command

SV = 0 or 8 Directly addressed vectors

XA = XB = XC = 0 No initial index for vectors A, B, & C.

SAA ~ Starting address of input trace (address of input point a_0)

SAB ~ Starting address of filter table (address of filter point b_{oo} of the 0-th filter)

SAC ~ Starting address of output trace (address of output point c_0)

VI = 0 Positive incrementing of vectors A, B, & C during self-loop.

DAI = -(L-1) Returns the address of the input trace to the current starting point (address of element a_m).

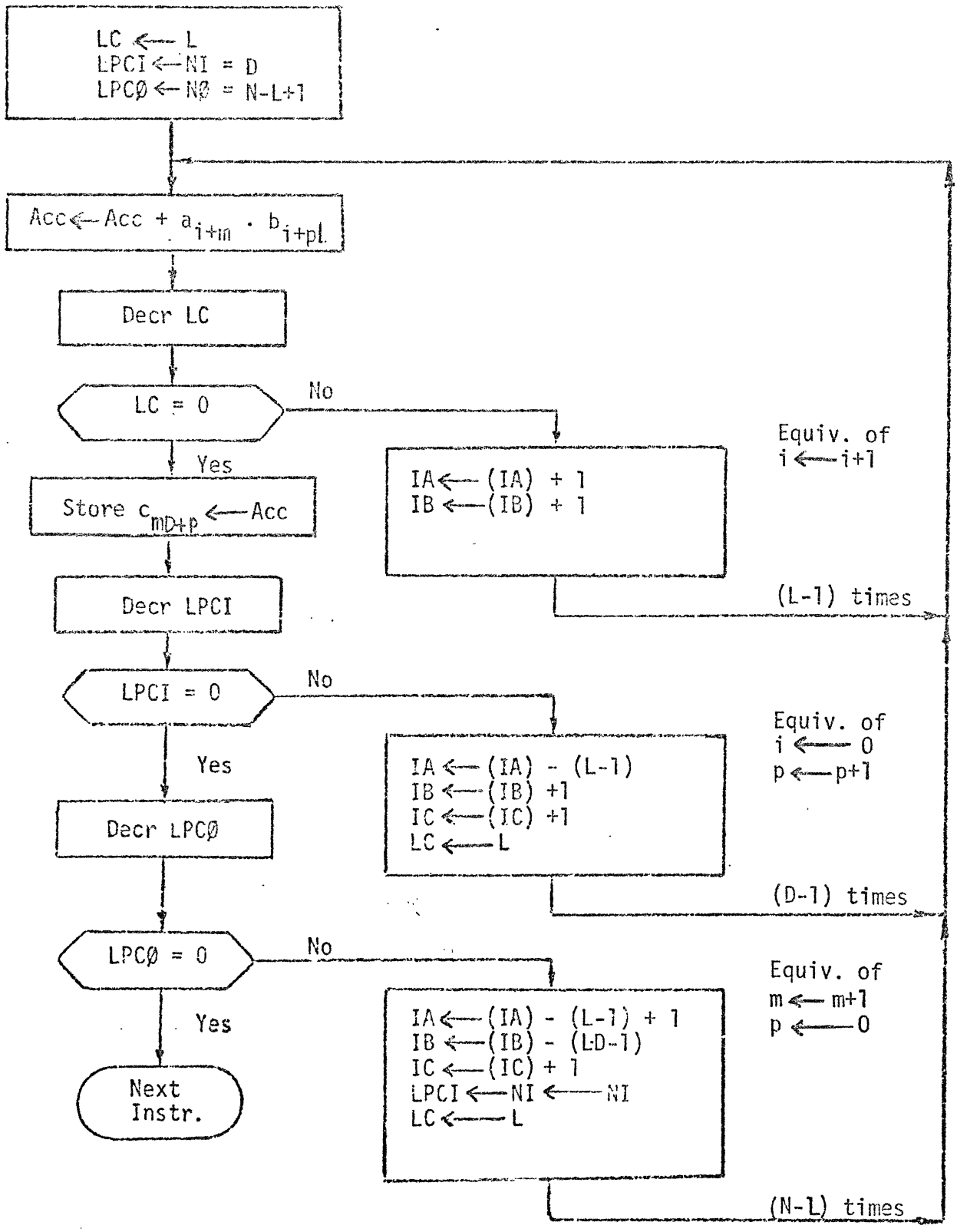


Figure 6. Flow chart for interpolation.

DBI = 1	Advances the address of the filter table to the first point of the next filter from the last point of the filter just applied.
DCI = 1	Advances the storage address of the output trace. Increments p in the representation of output term c_{mD+p}.
NI = D	D is the number of fixed filters. The inner loop is complete when D fixed filters have been applied to the current segment of the input trace.
DAØ = -(L-1)+1	Returns the address of the input trace to the current starting point plus one (address of element a_{m+1}).
DBØ = -(LD-1)	Returns the filter address to the start of the filter table from the address of the end filter point, $b_{L-1, D-1}$.
DCØ = 1	Advances the storage address of the output trace to the beginning of the next interpolation interval. Increments m by one in the representation of output term c_{mD+p}.
NØ = N-L+1	Specifies the outer loop count. For this example, D(N-L+1) output points are stored. The flow chart of Figure 6 shows that the output storage address is incremented D(N-L) + (D-1) times, or D(N-L+1) -1 times.

PERIPHERAL PROCESSOR DESCRIPTION

TABLE OF CONTENTS

FUNCTIONS OF THE PERIPHERAL PROCESSING UNIT

The peripheral processing unit (PPU) provides communication with I/O devices, functions as the system monitor, and fulfills those job requests which do not require a rich arithmetic instruction repertoire. The PPU is time shared at the bit time (85 n sec) level by up to eight programs, each of which is executed by one of eight virtual processors (VP_n).

One virtual processor is designated as the master controller. Other VP's are used for dedicated subservient control functions. The remainder of the VP's are scheduled as needed to perform I/O operations to schedule and perform other CP utilization tasks within the operating system.

ELEMENTS OF THE PPU

The PPU consists of eight virtual processors which time share a collection of other PPU elements. The shared elements include the arithmetic unit (AU), the read only memory (ROM), the file of communication registers (CR_n), and the single word buffer (SWB) which provides access to CM. These elements and their relationships are indicated in Fig. 1.

READ ONLY MEMORY

The ROM contains a pool of programs and is not accessed except by reference from the program counter. The pool includes a skeletal monitor program and at least one control program for each I/O device connected to the system. The ROM has an access time of 25 n sec and provides 32 bit instructions to the VP units. Total program space in ROM is expandable to 4K words. The memory is organized into 256 word modules so that portions of programs can be modified without complete refabrication of the memory.

The I/O device programs can include control functions for the storage media as well as data transfer functions. Thus, motion of mechanical devices can be controlled directly by the program rather than by highly special purpose hardware for each device type. Variations to a basic program are provided by parameters supplied by the monitor. Such parameters are carried in CM or in the accumulator registers of the VP executing the program.

VIRTUAL PROCESSORS

The eight VP's share the other PPU elements. To implement this sharing, time is divided into cycles with each cycle containing sixteen time slots. Each time slot is one bit period (85 n sec) in duration. These time slots are assigned to VP's on the basis of time slot availability and program requirements. A VP is operative whenever a time slot assigned to that VP

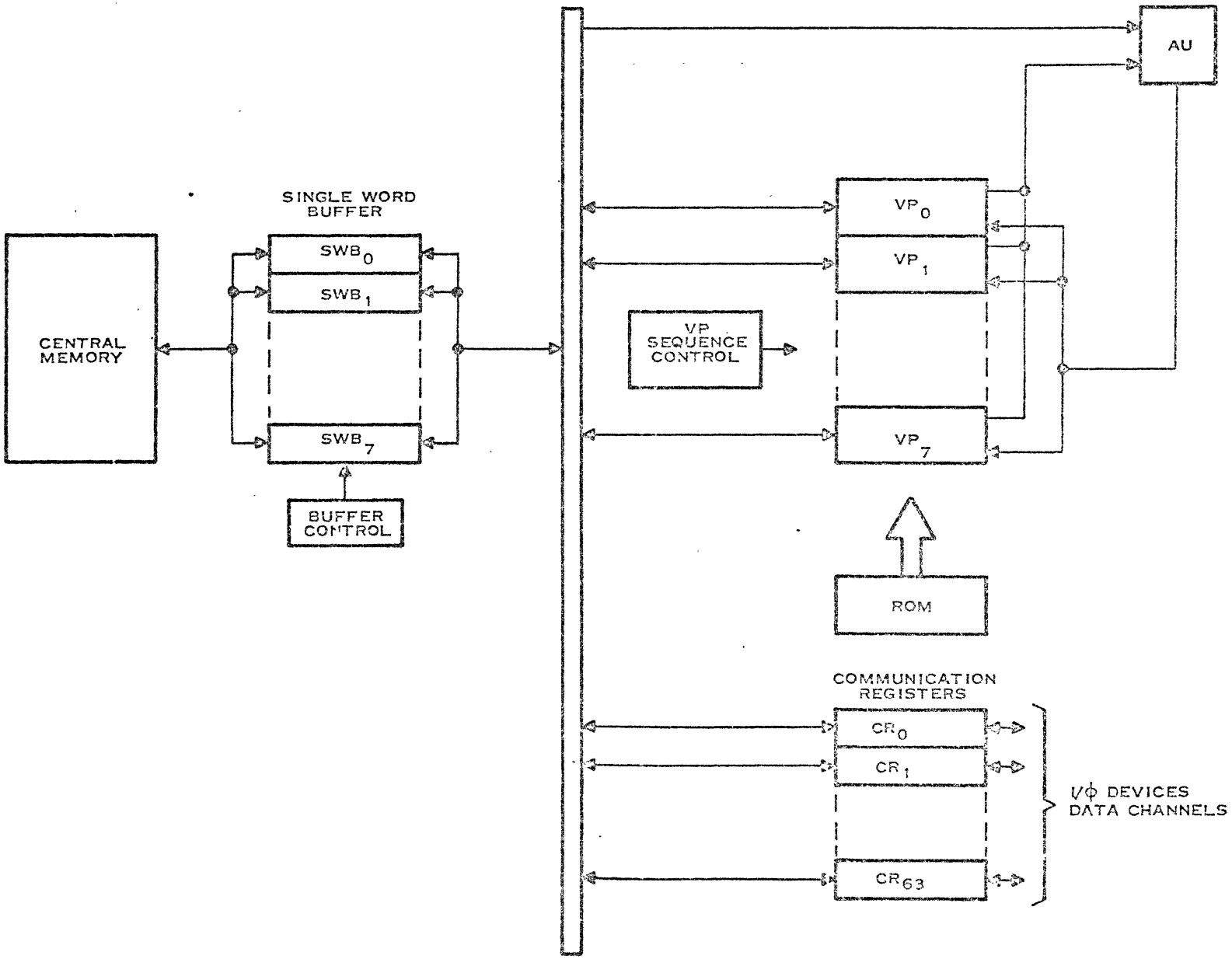


FIG. 1. Peripheral Processing Unit

occurs. Thus, if a VP has been assigned no time slots, it is not executing program. If a VP has been assigned one of the sixteen time slots, it is executing program 1/16 of the time. More than one time slot can be assigned to a single VP. The monitor VP turns the slave VP's on and off by manipulation of the time slot assignments.

The major components of each VP are a program counter (PC), a next instruction register (NIR), an instruction register (IR), and four accumulator registers (VPR_n) which are addressable to the byte level. Each VP also has a counter (BC) which keeps count of the number of bit periods which have been used in execution of the current instruction. When the time slot assigned to a particular VP occurs, the IR and the BC of that VP provide control of the PPU data paths and the AU. If the data manipulation thus effected completes the instruction, the IR is updated, and the BC is reset. However, if additional bit periods are required to complete the current instruction, the BC is advanced, and the IR remains unchanged.

The source of instructions may be either ROM or CM. The memory being addressed from the PC is controlled by the addressing mode which can be modified by the branch instructions or by clearing the system. Each VP is placed in the ROM mode and each PC points to location 0 when the system is cleared.

When the program sequence is obtained from central memory, it is acquired via the SWB. Since this is the same buffer used for data transfers to or from CM, and since CM access is slower than ROM access, execution time is more favorable when program is obtained from ROM.

All eight VP's are identical, but there exists a switch on the maintenance panel which selects one VP for certain operations. During normal system operation, the selector switch will always designate VP_0 but the manual selection is provided as a diagnostic aid. The switch selects the VP which will respond to the operator's switch manipulation during system initialization. The selected VP is also exempt from the scheme employed for protection of the contents of the CR file. In addition, the selected VP is provided with an automatic interrupt, whereas the interrupt for each of the other VP's must be programmed.

SINGLE WORD BUFFER

The SWB provides VP access to CM. The SWB consists of eight 32-bit data registers, eight 24-bit address registers, and controls. Each of the eight register sets has a fixed association with one of the eight VP's. Viewed by a single VP, the SWB appears to be a memory data register and a memory address register.

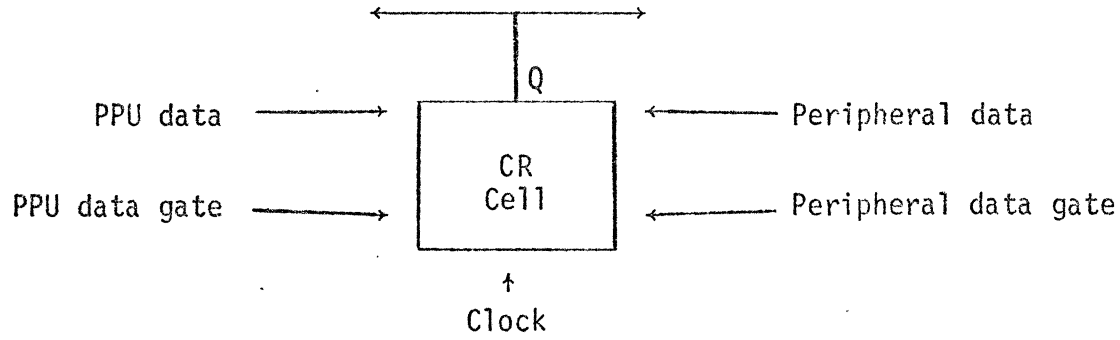
At any given time the SWB may contain up to eight memory requests, one for each VP. These requests are processed on a combinational priority - first in, first out basis. There are two priority levels, and if two or more requests of equal priority are unprocessed at any time, they are handled first in, first out.

When a request arrives at the SWB, it automatically has a priority assignment determined by the CM priority file maintained in one of the CR's. The file is arranged by VP number, and all requests from a particular VP receive the priority encoded in two bits of the priority file. The contents of the file are programmed by the monitor, and the priority code assignment for each VP is a function of the program to be executed by the VP.

COMMUNICATION REGISTERS

The PPU includes up to 64 CR's, each of which contains 32 cells. Each CR is addressable from the VP's and can also be read or written by the device to which it connects. The CR's provide the control and data links to all peripheral equipment including the system console. Some parameters which control system functioning are also stored in the CR's from where the control is exercised. An example of CR assignments is shown in Section C2. These assignments are unique to a particular ASC system.

Each CR cell has two sets of inputs as shown below. One set is connected into the PPU, and the other set is available for use by the peripheral device.



The contents of the first 20 CR's can be protected from modification by individual VP's. The protection is controlled by software via CR bits specifically for that purpose. The VP selected by the "VP SELECT" switch on the maintenance panel is insensitive to the CR protection scheme, and can always modify the contents of protected CR's.

PPU INTERRUPTS & TIME SLOT OVERRIDES

TIME SLOT OVERRIDES

The TIME SLOT OVERRIDE byte in the CR file provides one bit for each VP. The bit position corresponding with the VP selected by the VP SELECT switch on the Maintenance Panel is effective during certain maintenance operations. The remaining positions contain 1's to keep that VP inactive regardless of the contents of the TIME SLOT TABLE.

These override bits may be set or reset by software, however, under certain circumstances these bits will be set automatically by hardware, but they are never reset by hardware.

Automatic setting of override bit "i" occurs as a result of any of several events in VP_i. The events are:

a. CM parity error
b. CM protection violation
c. CM breakpoint
d. CR protection violation
e. illegal OP code

When one of these events is detected, the appropriate override bit is set (other than the selected VP) and subsequent time slots assigned to the VP are voided. The VP may not complete its current instruction when the override bit is set.

When automatic setting of an override bit occurs, the TIME SLOT OVERRIDE REASON byte in the CR file may also be automatically updated. This byte consists of a control bit, a three bit VP # field, and a four bit override reason field. If the control bit is "1", then updating of the TIME SLOT OVERRIDE REASON byte is inhibited. If the control bit is "0", then updating of the byte is permitted.

Updating consists of setting the control bit to "1", loading the VP # field with the number of the VP for which an override event has been detected, and loading the override reason field with a code indicating which event a. through e. has occurred. If an override event occurs while the control bit is set, the time slot override occurs, but the reason for the override is not recorded. The reason codes are as follows:

Code	Event
0	a
1	b
2	c
3	d
4	e
5-F	unused

AUTOMATIC INTERRUPT

The VP selected by the VP SELECT switch is provided with an automatic instruction level interrupt. When the interrupt signal occurs, the VP completes its current instruction, and then traps to ROM location 10_{16}. Any of several events can produce the interrupt signal. The events are:

A. A/C power failure
B. CM parity error in selected VP
C. CM protection violation in selected VP
D. CM breakpoint in selected VP
E. STOP button (Operator's Panel)
F. disc protection violation
G. illegal OP code in selected VP
H. CP interrupt

To provide effective, efficient reaction to these events, three bytes in the CR file are employed. These are the OUTSTANDING EVENT byte, the INTERRUPT MASK byte, and the PROCESSED INTERRUPTS byte. The relationship of these bytes is indicated in Figure 2 and is explained in the following paragraphs.

When events A. through H. are detected, they are recorded in the OUTSTANDING EVENT byte. This record of the event remains until the selected VP responds to an interrupt signal caused by the event or until the record is erased by software. When an event is thus recorded, the interrupt signal to the selected VP is generated providing the corresponding bit position in the INTERRUPT MASK byte is "1". When the VP responds to the interrupt signal, the three CR bytes are modified as follows:

1. A record of the event or events causing the interrupt is made in the PROCESSED INTERRUPTS byte. This record may be referenced by software during interrupt servicing and should be reset by software at the conclusion of the service.
2. The record in the OUTSTANDING EVENT byte of the event or events causing the interrupt is erased. Note that events recorded in this byte, but inhibited by the mask, are not erased.
3. The mask bit for the event causing the interrupt and the mask bits for all events of equal or lower priority are reset, thereby inhibiting interrupts caused by subsequent events whose priority is equal or lower. Event priority is as follows:

Event	Priority
A	high
B	middle
C-H	low

After these modifications have occurred, the software service routine commences. During this time, all events will be recorded as OUTSTANDING EVENTS, but will not generate interrupts unless their priority is high enough to overcome the automatic inhibiting which has occurred. At the conclusion of the interrupt service routine, the PROCESSED INTERRUPTS byte should be reset by software, and the INTERRUPT MASK byte should be restored by software to any desired mask. Note that when a mask bit is set by software that an interrupt will immediately occur if the corresponding event has been recorded in the OUTSTANDING EVENT byte but has previously been inhibited. If this is not desired, any of the event records in the OUTSTANDING EVENT byte may be erased by software before the modification of the INTERRUPT MASK byte.

PROGRAMMED INTERRUPTS

Each VP, including the selected VP, is provided with an instruction level interrupt which can be initiated only under software control via the INTERRUPT CONTROL byte in the CR file. VP reaction to the interrupt is similar to reaction to the automatic interrupt already described. When the interrupt signal occurs, the VP completes its current instruction and then traps to ROM location 11_{16}. Note that each VP employs the same trap location.

The three CR bytes employed for the previously described automatic interrupt are in no way related to the programmed interrupts. The programmed interrupts are controlled only by the INTERRUPT CONTROL byte in the CR file. The programmed interrupt structure is depicted in Figure 3.

When bit "i" in the INTERRUPT CONTROL byte has been set to "1" by software, an interrupt signal to VP_i is generated. When the VP responds to the interrupt, bit "i" is automatically reset.

Note that the VP will not respond to the interrupt under the following conditions:

1. The VP has no assigned time slot.
2. The VP's time slot assignment is being overriden by the TIME SLOT OVERRIDE byte.
3. The VP cannot complete its current instruction due to an endless loop of indirects.

MAINTENANCE EXCEPTIONS

During maintenance operations, the foregoing automatic hardware reactions may be modified. There are two switches on the Maintenance Panel which affect the reactions. The switches are the TEST MODE switch and the AUTO INTERRUPT OFF switch.

If the TEST MODE switch is in the "Normal" position, then the selected VP cannot experience a CR protection violation. However, if the switch is not in the "Normal" position, the selected VP becomes subject to CR protection under control of the CR PROTECT CONTROL byte in the CR file. If CR protection for the selected VP is in effect, and if a violation occurs, the hardware reaction is as follows:

1. The addressed CR is not modified, and
2. The event is recorded as event E in the OUTSTANDING INTERRUPT byte. Thus, under these circumstances, event E may represent either (a) CR protection violation in selected VP, or (b) STOP button (Operator's Panel).

The other Maintenance Panel switch which affects the hardware's reactions to the events under consideration is the AUTO INTERRUPT OFF switch. If this switch is in the "off" condition (not the normal operating condition), then the following applies:

1. Events A through H are recorded, as usual, in the OUTSTANDING EVENTS byte. However, no interrupt signal to the PPU results. The reaction is as if all interrupt mask bits were "0".
2. The time slot override bit for the selected VP is effective, and is set by hardware if event A through H occurs.
3. The override reason byte operates normally for non-selected VP's, but is not updated if the override is for the selected VP.

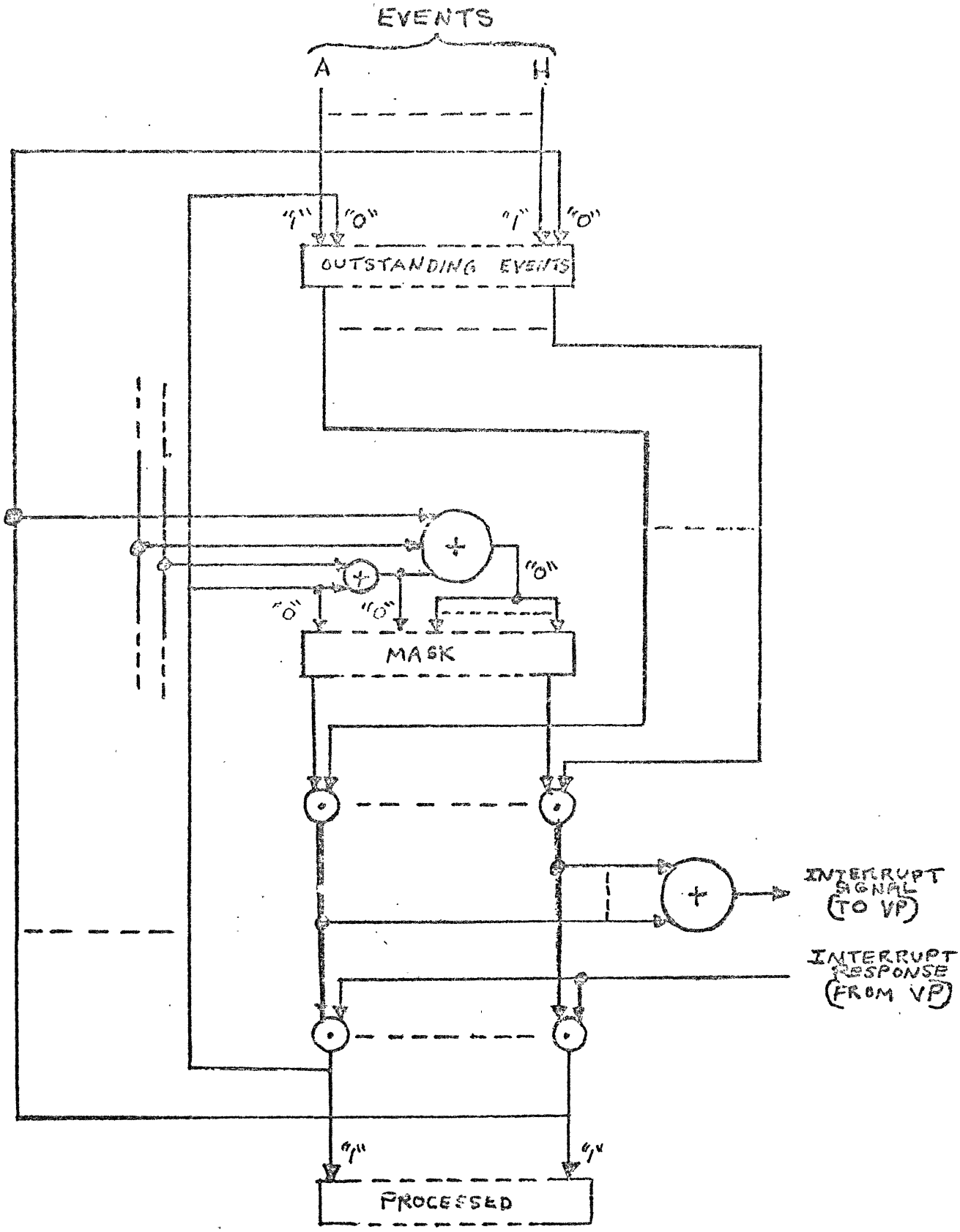


FIG. 2. AUTOMATIC INTERRUPTS

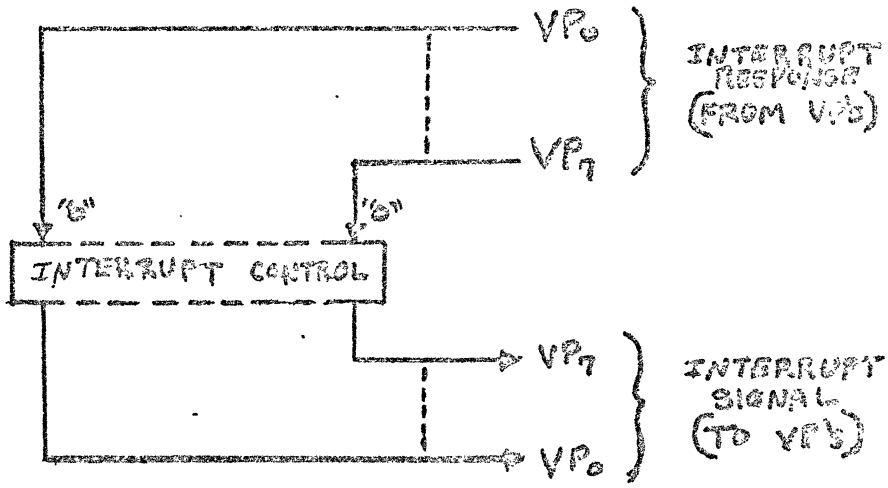


FIG. 3 PROGRAMMED INTERRUPTS

LOGIC CLOCK MODULE

The Logic Clock Module (LCM) is capable of performing maintenance functions and providing synchronization for normal operation. Specific clock pulse sources are three different internally (on the LCM Card) generated clock rates and provisions for using an oscillator external to the machine as a clock signal.

The three internally generated clock rates are normal (85 ns), marginal, (5% faster than normal), and slow (normal period plus 100 nanoseconds). The limits of the external signal source have not been determined. The internal source period is adjustable (for the normal and marginal rates) in steps of 1/2 over the range from 40 to 159 by using jumper wiring through the proper sequence of printed circuit delay lines.

The LCM has 9 input control lines. Seven of these input lines constitute a COMMAND to the logic clock as follows:

X_M	$X_8X_4X_2X_1$	X_AX_B
	Burst Count Code (BCD, 0-15 Pulses)	Clock Source Code
0 → Run continuously		0 0 → Normal
		0 1 → Marginal
1 → Burst		1 0 → Slow
		1 1 → External

The command is entered into the LCM registers by the SYSTEM STANDBY or LOAD signals. SYSTEM STANDBY initializes the LCM when it is true and causes COMMAND to be loaded and executed when it goes false. All further COMMANDS are loaded by LOAD. LOAD control signal enters COMMAND into the LCM registers when it goes true and execution begins with the second clock period. When COMMAND has been executed, a REPLY output signal is set true. When operating in BURST, the REPLY signal is not set until the completion of the required pulses.

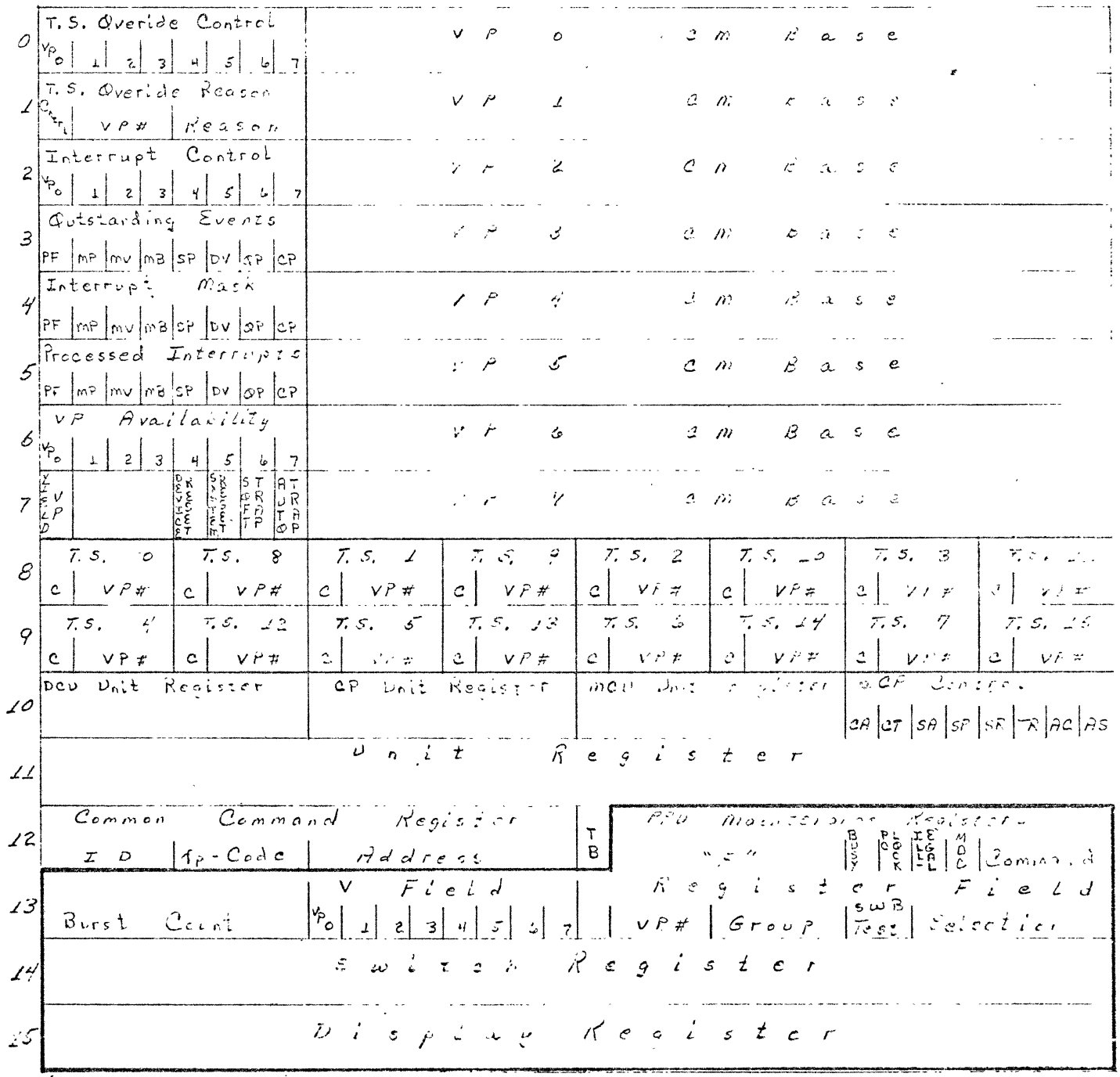
The logic clock can BURST or RUN continuously with any of the four clock sources, and it is stopped with a burst of zero pulses (COMMAND: 1 0000 X_AX_B). Since all synchronization on the LCM is accomplished using a locally shaped clock, control functions may be performed even if the system external to the LCM is not receiving a clock signal.

COMMUNICATION REGISTERS DESCRIPTION

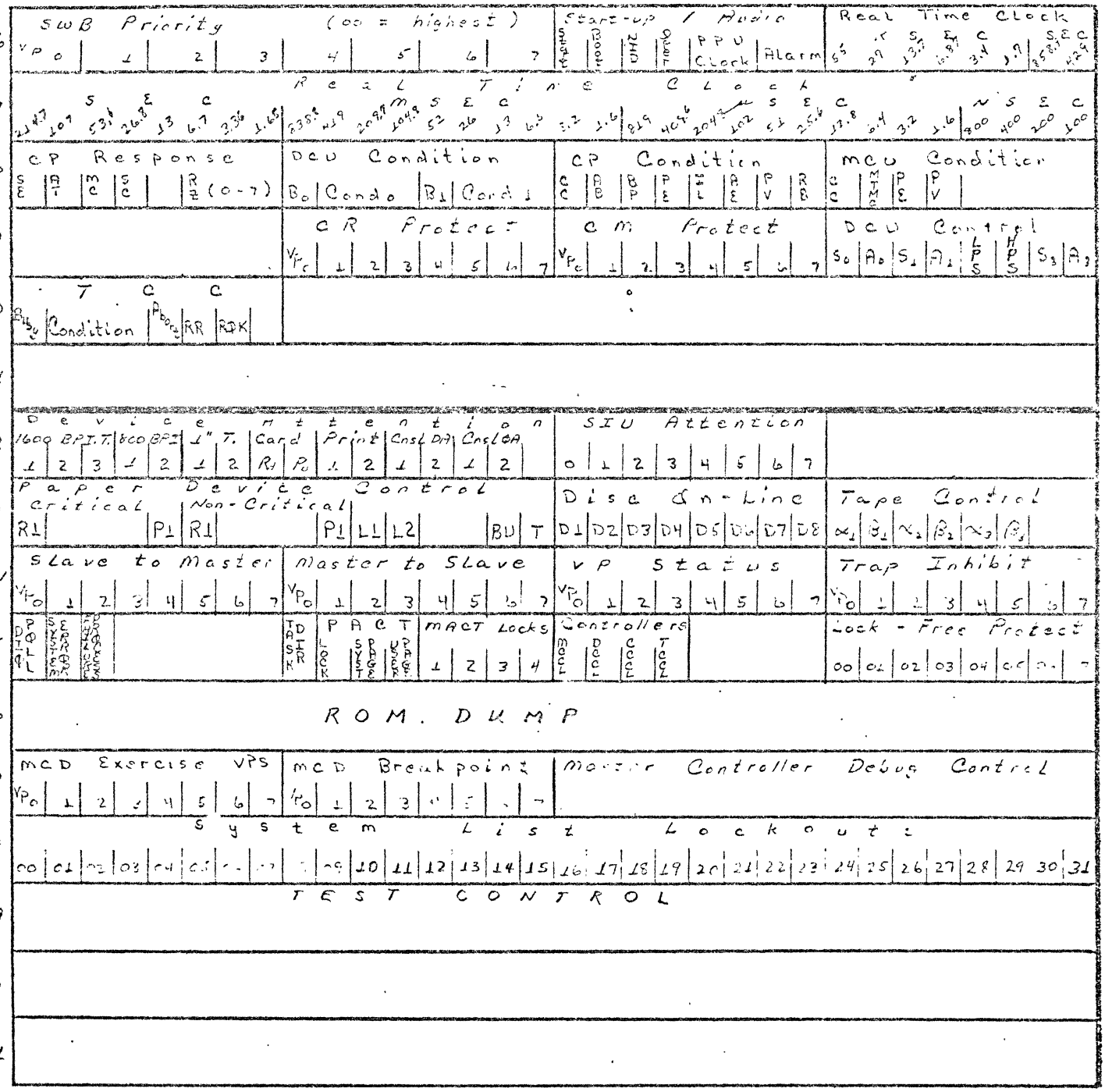
COMMUNICATION REGISTERS DESCRIPTION

TABLE OF CONTENTS

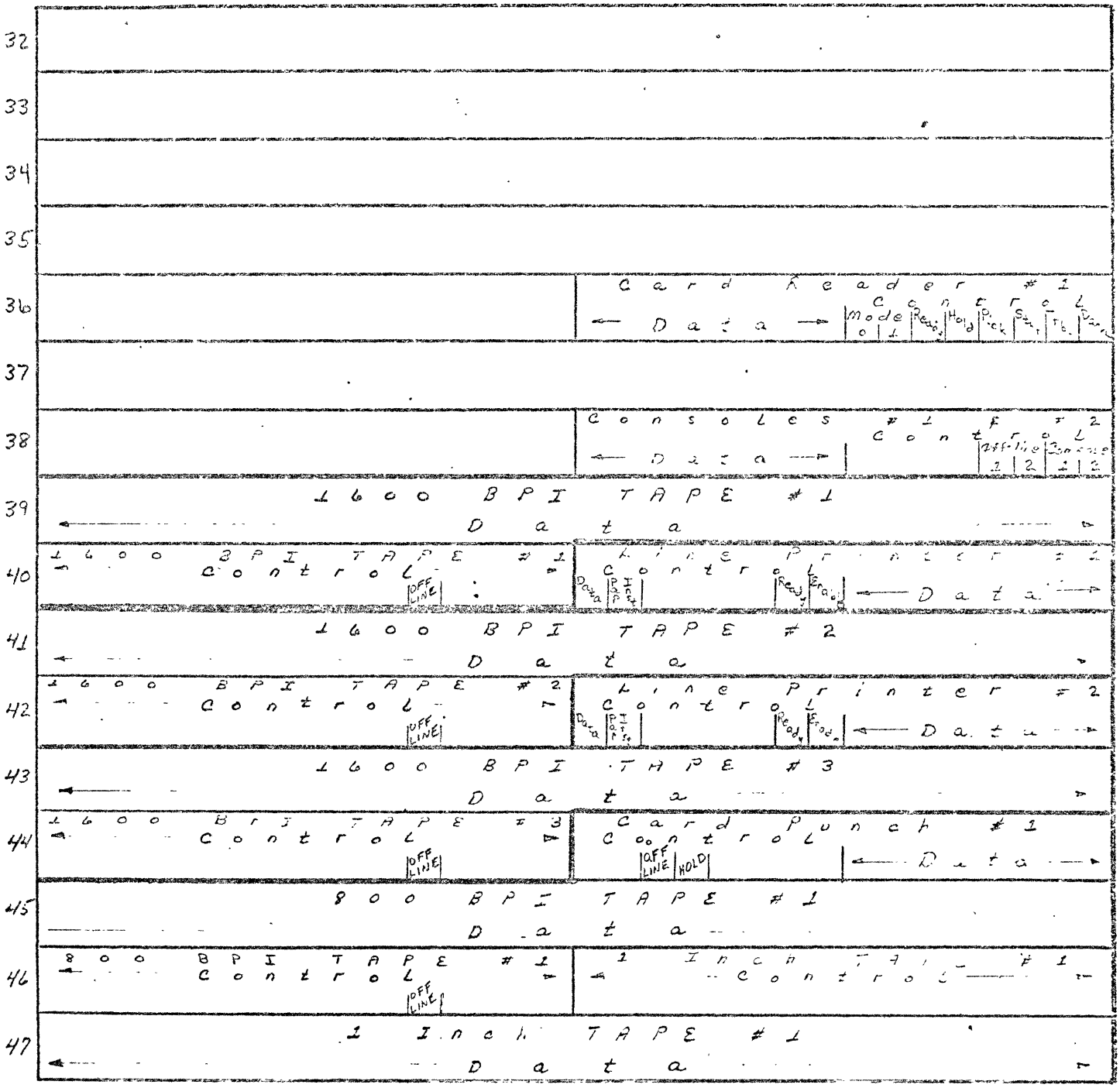
Communications Registers and Assignments for ASC System #1



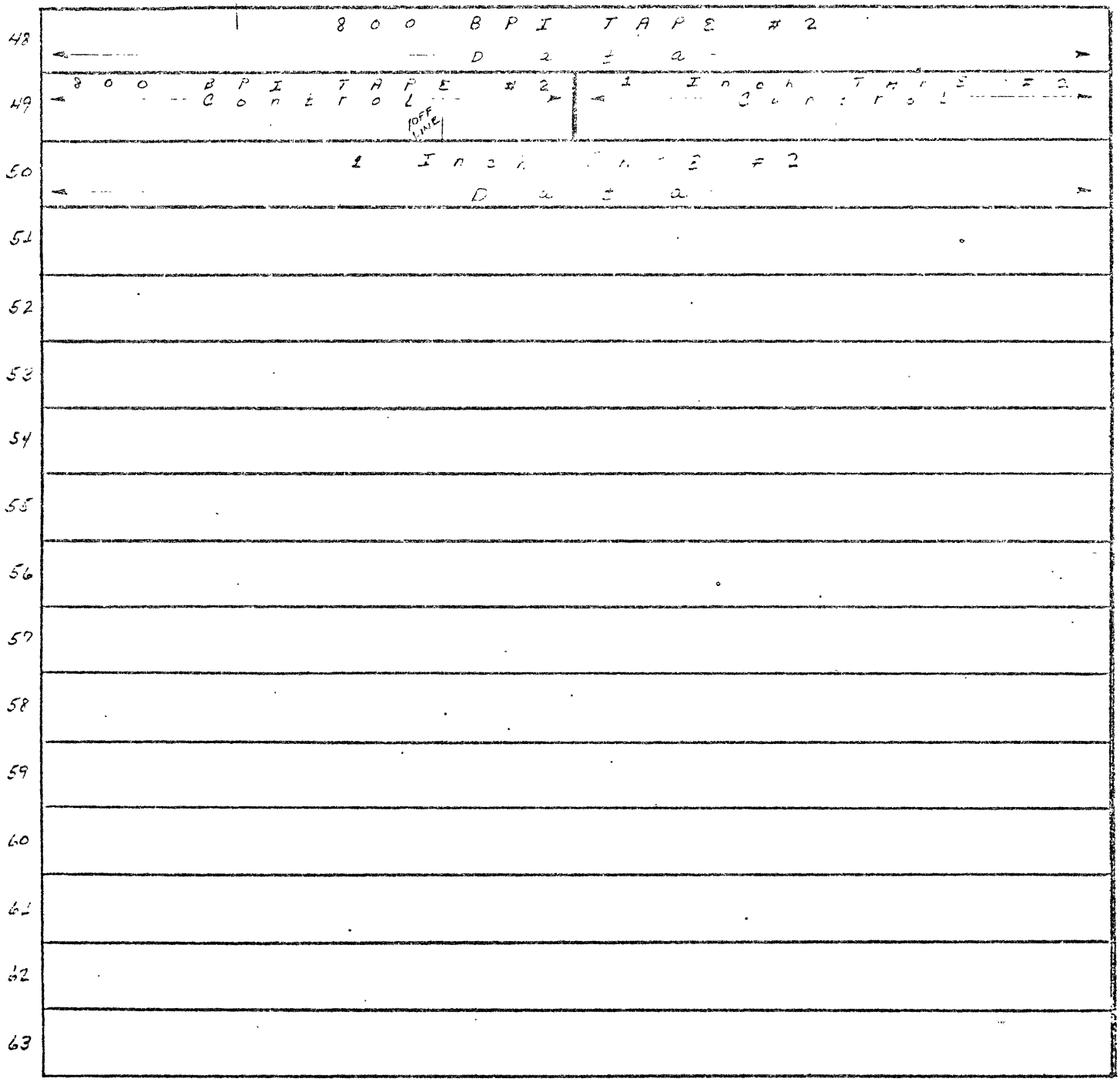
Communications Registers and Assignments for ASC System #1



Communications Registers and Assignments for ASC System #1



Communications Registers and Assignments for ASC System #1



OUTSTANDING EVENT

PF	MP	MV	MB	SP	DV	OP	CP

This byte records events which will cause an automatic interrupt. A bit is set by an event and is reset when the interrupt reaction occurs. The byte is never set by software, nor is it expected that the byte be read or tested by software. A complete description of this byte is available in Section C1.

Events associated with the bit positions are:

PF - a/c power failure

MP - CM parity error in selected VP

MV - CM protection violation in selected VP

MB - CM breakpoint in selected VP

SP - STOP button (Operator's Panel)

DV - Disc protection violation

OP - Illegal OP code in selected VP

CP - CP interrupt

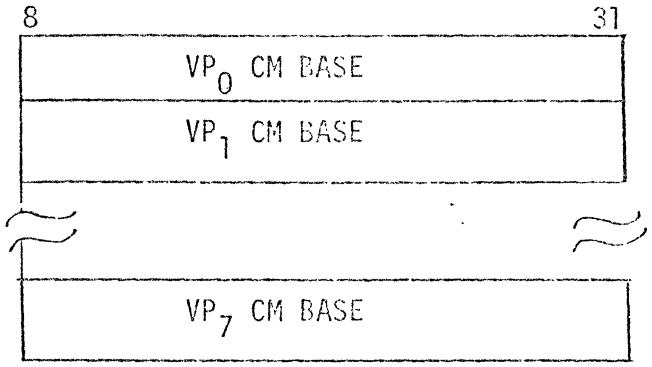
INTERRUPT MASK

This byte prevents events recorded in the Outstanding Event byte from causing the automatic interrupt. These bits are set by software to permit interrupts and may be reset by software to inhibit interrupts. Bit position interpretations correspond with those of the Outstanding Event byte. Hardware also resets bits as described in Section C1.

PROCESSED INTERRUPTS

This byte indicates which of several events has caused an automatic interrupt. These bits are set by hardware when the interrupt occurs, and must be reset by software. Bit position interpretations correspond with those of the Outstanding Event byte. A complete description of the Processed Interrupt byte is available in Section C1.

VP CM BASE



These registers are used by the VP's for development of CM addresses as indicated by the instruction descriptions in Section C4.

Software Responsibility:

1. Enter the appropriate CM base address for program to be run in the VP.

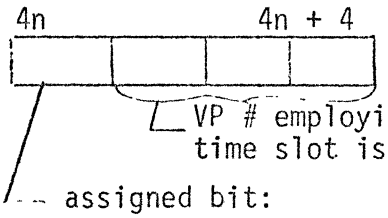
Hardware Responsibility:

1. Select one of the eight base registers as determined by the Time Slot Table and the time slot look ahead logic, and present the contents of the selected register to the PPU internal controls.

TIME SLOT TABLE

0							31
T.S. 0	T.S. 8	T.S. 1	T.S. 9	T.S. 2	T.S. 10	T.S. 3	T.S. 11
T.S. 4	T.S. 12	T.S. 5	T.S. 13	T.S. 6	T.S. 14	T.S. 7	T.S. 15

T.S. 1 Thru T.S. 15



0 this time slot not assigned.

1 this time slot assigned to the VP indicated.

Time slots occur in sequence from time slot 0 through time slot 15. Time slots 1 through 15 are assigned to the VP's designated in the table. If a VP is assigned a series of adjacent time slots, only the first and alternate slots of the series are effective. Time Slot 0 uses the TEST MODE switch to select the VP. If the TEST MODE switch is in the NORMAL position, the VP SELECT switch specifies the VP using T.S.0. If the TEST MODE switch is not NORMAL, the T.S. table specifies the VP. The Time Slot Table can be overridden when the Time Slot Override byte indicates that the table entry is invalid. The time slot use is also affected during maintenance operations by the V, BURST, and CONTROL fields of the PPU MAINTENANCE CONTROL registers, and by the TEST MODE switch on the Maintenance Panel.

Software Responsibility:

1. Modify the time slot table as required to control VP activity.

Hardware Responsibility:

1. Provide a time slot counter to designate the number of each time slot.

2. Scan the TIME SLOT TABLE to select the VP to be allotted each time slot.

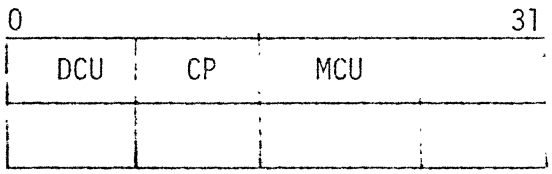
3. Alter the result obtained by the scan of the TIME SLOT TABLE if:

   a) TIME SLOT OVERRIDE is indicated.

   b) Adjacent time slots are assigned to a VP.

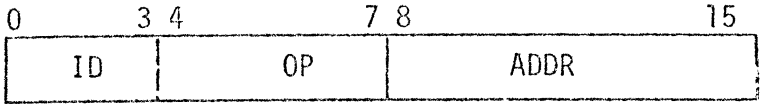
   c) Certain maintenance functions are in effect as determined by the Maintenance Logic which interprets the PPU MAINTENANCE CONTROL registers located in the CR file.

UNIT REGISTERS



These registers are used for direct communication between the PPU and other units within the system and are used in conjunction with the Common Command Register. Complete detailed assignments for these registers have not been determined, but the primary use of them will be for maintenance purposes.

COMMON COMMAND REGISTER



The COMMON COMMAND REGISTER (CCR) provides commands from the PPU to all other units to control maintenance facilities and to provide communication links which are infrequently used. The ID field designates which unit is to interpret the commands and perform the necessary actions. The OP field is an operation code. The ADDR field contains an address or an operand if required for the command.

A more detailed explanation of the intent of the CCR is in Section G.

Software Responsibility:

1. Insert commands into the CCR as required.

Hardware Responsibility:

1. Present the contents of the CCR to all units.

CCR TRANSFER BIT

This bit is used to control transfer of the command contained in the CCR. The CCR TRANSFER BIT is set by software to indicate to CCR recipients that a new command is available in the CCR. The unit addressed by the CCR then resets the CCR TRANSFER BIT when the command has been received or when it has been executed, depending on the type of command. If the type of command is such that the CCR TRANSFER BIT is reset on receipt of the command by the unit, then completion of the command will be indicated via the "Command Complete" bit in the CONDITION byte relating to the appropriate unit.

Note that a command can be issued to CCR recipient units by a single word transfer into the entire CR word containing the CCR and the CCR TRANSFER BIT, providing the remaining 15 bits of the word are written as "0's".

Software Responsibility:

1. Set the CCR TRANSFER BIT to notify CCR recipients that a new command is present.
2. Monitor the CCR TRANSFER BIT to determine when an issued command has been executed.

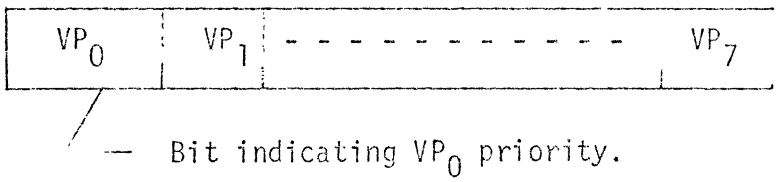
Hardware Responsibility:

1. Present the contents of the CCR TRANSFER BIT to all units.
2. Provide input lines for resetting of the bit.

PPU MAINTENANCE CONTROL

This portion of the CR file has been described in Section G.

SWB PRIORITY



code	Priority
0	highest
1	lowest

This halfword designates the priority of VP requests to CM. The bit associated with each VP may be modified by VP programs at any time. When more than one CM request is present in the SWB, the requests are serviced according to the priority assignment in the SWB PRIORITY halfword. If two requests of equal priority are present, then the oldest request is serviced first.

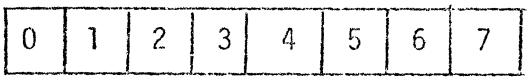
Software Responsibility:

1. Load SWB PRIORITY as required for optimum use of CM by the PPU.

Hardware Responsibility:

1. Hardware selects the appropriate bit for presentation to the internal SWB controls. Selection is based in the contents of the T.S. table and the time slot look ahead logic.

START-UP AND AUDIO



bit 0 - Start

bit 1 - Bootstrap

bit 2 - Native Input Device Code

bit 3 - Operating

bit 4 -
bit 5 - > PPU Clock Control

bit 6 -
bit 7 - > Audible Alarm

Start-up (bits 0 through 3) is initiated by the use of the START button or the LOAD button located on the Operator's Panel. Depression of the LOAD button must be simultaneous with positioning of the spring loaded system initialization switch in order to be effective. Start-up causes the following sequence:

1. All cells are cleared via the asynchronous reset lines. (The system can be shut down and stopped via the console or by depression of the STOP button on the Operator's Panel with no intervening power loss to clear the cells.)

2. a) The Start bit is set.

   b) The Bootstrap bit is set if the start-up was initiated by the LOAD button, but is not set if the start-up was initiated by the START button.

   c) The Native Input Device code is set according to the position of the system initialization switch on the Operator's Panel. "0" denotes Card Reader, "1" denotes disc.

   d) The Operating bit is set.

The Audible Alarm bits control an audible alarm. If bit 6 is "1" then the alarm is in the fixed frequency mode, and bit 7 is the on/off switch. A "1" in bit 7 denotes "on". If bit 6 is "0" then the alarm is in the generated frequency mode, in which case bit 7 is the audio source. In this mode, any desired tone can be created by programming bit 7 to change at the desired frequency.

Software Responsibility:

1. Interprets and resets bits 0 and 1.
2. Interprets bit 2.
3. Resets bit 3 at the conclusion of shut down.
4. Sets and resets bits 6 and 7 as desired for audio control.

Hardware Responsibility:

1. Sets bit 0 when the START button or the LOAD button is depressed. The buttons are on the Operator's Panel.
2. Sets bit 1 when the LOAD button is depressed.
3. Sets and/or resets bit 2 when the LOAD button is depressed so as to reflect the position of the system initialization switch on the Operator's Panel. A "0" in bit 2 indicates that the system initialization switch is in the Card Reader position, and a "1" indicates the disc position.
4. Sets bit 3 when the START button or the LOAD button is depressed. Employs bit 3 for operation of the OPERATING light on the Operator's Panel.
5. Continually react to bits 6 and 7 as described above.

The PPU Clock Control bits control the frequency of the PPU logic clock when the CLOCK RATE switch on the Maintenance Panel is in the "Normal" position. For other positions of the CLOCK RATE switch, the two Clock Control bits are ineffective. The code is:

bit 4	bit 5	
0	0	Nominal Rate
0	1	Slow (Nominal Period + 100 ns)
1	0	Fast (Nominal Frequency + 5%)
1	1	Nominal Rate

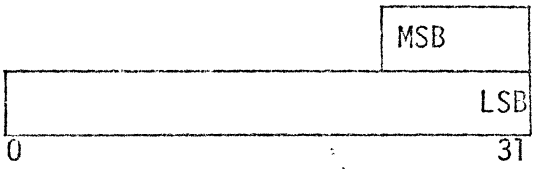
Software Responsibility:

1. Set and Reset bits for desired frequency code.

Hardware Responsibility:

1. Control PPU logic clock frequency as described above.

CLOCK



These five bytes provide a binary count of lapsed time in the system. This counter is driven by an independent 10 mhz oscillator; however, since these CR bytes are updated in synchronism with the PPU logic clock (nominal 85 nanosecond period), the least significant two or three bits of the count will not be accurate.

If the contents of these bytes are altered by software, the count is indeterminant until the next hardware update occurs. The update nullifies the attempted programmed modification.

Software Responsibility:

1. Read the CLOCK as required.
2. Never write into the CLOCK.

Hardware REsponsibility:

1. Update the CLOCK in synchronism with the PPU logic clock.
2. Provide a 10 mhz counter as a source for updating of CLOCK.

AVAILABILITY

VP_0	VP_1	- - -	VP_7

This byte provides a running account of which VP's are available for assignments. The start-up procedure software sets these bits to 1 to show all VP's available.

Software Responsibility:

1. Set bits to show VP availability.
2. Reset bits to show VP activity.

Hardware Responsibility:

1. None.

DCU CONDITION

The MSB of this byte is the Command Complete bit. This bit is set by the DCU at the conclusion of certain types of maintenance commands issued to the DCU via the Common Command Register. The bit is subsequently reset by software.

The remaining bits are used by the DCU to notify the PPU that certain unusual events have occurred. Detailed assignments for the bits have not been determined, but it is anticipated that they will include CM parity error, CM breakpoint, and CM protection violation. It is possible that some of the condition bits may indicate that additional data relative to the event may be obtained thru the use of the CCR.

CP CONDITION

The MSB of this byte is the Command Complete bit. This bit is set by the CP at the conclusion of certain types of maintenance commands issued to the CP via the Common Command Register. The bit is subsequently reset by software.

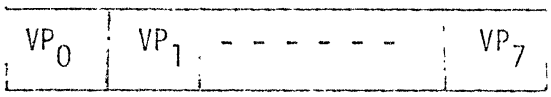
The remainder of this byte is used by the CP to notify the PPU that unusual events have occured. These unusual events include CM parity error, CM breakpoint, and CM protection violation. Refer to this Section, page 23, for a detailed description of the CP condition byte.

MCU CONDITION

The MSB of this byte is the Command Complete bit. This bit is set by the MCU at the conclusion of certain types of maintenance commands issued to the MCU via the Common Command Register. The bit is subsequently reset by software.

The remainder of this byte is used by the MCU to notify the PPU that unusual events have occurred. Detailed assignments for the bits have not been determined, but it is anticipated that they will include CM parity error, CM breakpoint, and CM protection violation. It is possible that some of these condition bits may indicate that additional data relative to the event may be obtained through use of the CCR.

CR PROTECT CONTROL



These bits can place the VP under CR Protection to prohibit writing into any CR up to CR number 13_{16}. When a CR protection violation occurs, the CR cells involved are not changed and the time slot override is set. The bit position indicated by the VP SELECT switch on the Maintenance Panel is ineffective when the TEST MODE switch on the Maintenance Panel is in the "Normal" position.

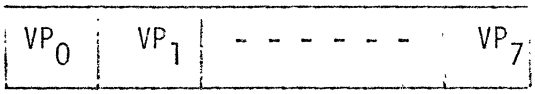
Software Responsibility:

1. Set bits to subject VP's to CR protection on all subsequent write references.
2. Reset bits to inhibit the CR protection check on all subsequent write references.

Hardware Responsibility:

1. When a bit is set, it sends a protect enable signal into the internal PPU controls at the appropriate time as indicated by the T.S. Table and the time slot lookahead logic. A "1" in the bit position indicated by the VP SELECT switch is ignored.

CM PROTECT CONTROL



These bits can place the VP under CM protection. All VP's employ the same three boundary pair parameters that are set up through the use of the Common Command Register (CCR). All CM requests, issued by a VP, are classified in one of three classes. If protection is in effect, the address requested is checked against the appropriate boundary pair. The three classes of requests are write, instruction fetch, and operand fetch. If a write request violates the protection, the memory cell is not modified. All violations result in a hardware reaction identical to that for breakpoint, i.e., either time slot override or automatic VP interrupt, depending on comparison of the VP number in violation and the VP SELECT switch.

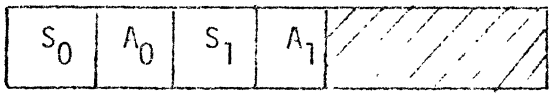
Software Responsibility:

1. Set bits to subject VP's to CM protection on all subsequent CM references.
2. Reset bits to inhibit the CM protection check on all subsequent CM references by VP's.

Hardware Responsibility:

1. When a bit is set, it sends a CM protect enable signal into the internal SWB controls at the appropriate time as indicated by the Time Slot Table and the time slot lookahead logic.

DCU CONTROL (for ASC system # 1)



These bits control the disc activity via the two Data Channels, DC_0 and DC_1. The start bit (S_n) is set to "1" by the software to indicate to the channel that execution of a chain of commands will start. This bit is reset by the channel when the chain has been completed and the channel is again idle. The Abort bit (A_n) is set by software to cause the channel to abort the current link in a chain of commands. The bit is reset by the channel, indicating completion of the link deletion.

SLAVE TO MASTER

These bits are flags for the software. Bits are set by slave VP's when service from the master VP is required and are reset by the master VP.

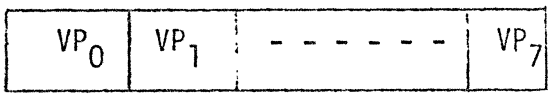
Software Responsibility:

1. Set and reset bits as required for passing messages from slave VP's to master VP.

Hardware Responsibility:

1. None.

MASTER TO SLAVE



These bits are flags for the software. Bits are set by the master VP to indicate required action by the slave VP. The bits are reset by the slaves.

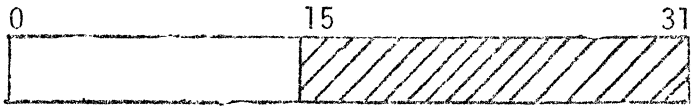
Software Responsibility:

1. Set and reset bits as required for passing messages from the master VP to the slave VP's.

Hardware Responsibility:

1. None.

DEVICE ATTENTION (for ASC System # 1)



These bits indicate that an I/O device status has changed. There is one bit for each magnetic tape, paper device, and console. When software has taken appropriate action, software resets the bits. The bits do not indicate status but rather status change. Bit positions are assigned as follows:

b_0 - 1600 bpi tape #1

b_1 - 1600 bpi tape #2

b_2 - 1600 bpi tape #3

b_3 - 800 bpi tape #1

b_4 - 800 bpi tape #2

b_5 - 1" tape #1

b_6 - 1" tape #2

b_7 - card reader #1

b_8 - card punch #1

b_9 - printer #1

b_{10} - printer #2

b_{11} - console #1

b_{12} - console #2

b_{13} - console operator #1

b_{14} - console operator #2

Software Responsibility:

1. Monitor these bits to determine that status changes have occurred.
2. Reset bits.

Hardware Responsibility:

1. Set bits to indicate device status changes.

MONITOR Q NON-EMPTY

This byte is employed by software in conjunction with the Monitor Q Control byte in order to implement efficient and orderly access to queques in CM.

Software Responsibility:

1. Set and reset bits as required by activity in queques.

Hardware Responsibility:

1. None.

MONITOR Q CONTROL

This byte is employed by software in conjunction with the Monitor Q Non-Empty byte in order to implement efficient and orderly access to queques in CM.

Software Responsibility:

1. Set and reset bits as required by activity in queques.

Hardware Responsibility:

1. None.

TCP Q NON-EMPTY

This byte is employed by software in conjunction with the TCP Q Control byte in the CR file in order to implement efficient and orderly access to queques in CM.

Software Responsibility:

1. Set and reset bits as required by activity in queques.

Hardware Responsibility:

1. None.

TCP Q CONTROL

This byte is employed by software in conjunction with the TCP Q Non-Empty byte in order to implement efficient and orderly access to queques in CM.

Software Responsibility:

1. Set and reset bits as required by activity in queques.

Hardware Responsibility:

1. None.

PAPER CONTROL (for ASC system #1)

These bits are used by several paper device handler programs to share a VP. The paper device driver program operates at a functional level between the PCU programs and the paper device handlers. The driver temporarily relinquishes control of the VP to device handler programs as required. The driver polls the Paper Control bits to determine which of several paper devices requires use of the shared VP. The driver and the Paper Control bits allow for expansion to ten paper devices as shown below:

Card Readers.....(R) bits 0 (critical) b_0 to b_3 expansion direction
5 (non-critical) b_5 to b_8 expansion direction
Card Punchers.....(P) bits 4 (critical) b_4 to b_1 expansion direction
9 (non-critical) b_9 to b_6 expansion direction
Line Printers.....(L) bits 10,11 (critical) b_{10} to b_{14} expansion direction

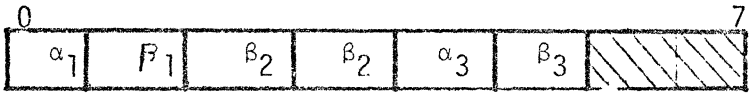
Paper Driver Terminate(T) bits 15

These bits are controlled entirely by the paper driver, handlers, and the associated hardware except when the Programming System wants to indicate a device or to terminate the driver. The initiating a device, the Programming System will set the appropriate device CALL (K) bit, which will be reset when the driver detects it and the driver is initiated. If the TERMINATE (T) bit is set by the Programming System, the driver will reset the bit and terminate the driver.

Software Responsibility:
1. Set call bits to initiate devices.
2. Set the terminate bit to terminate the device.
3. Monitor all bits to determine that status changes have occurred.
4. Reset bits when recognized and responded to.

Hardware Responsibility:
1. Set Critical Service bits in response to device activity.

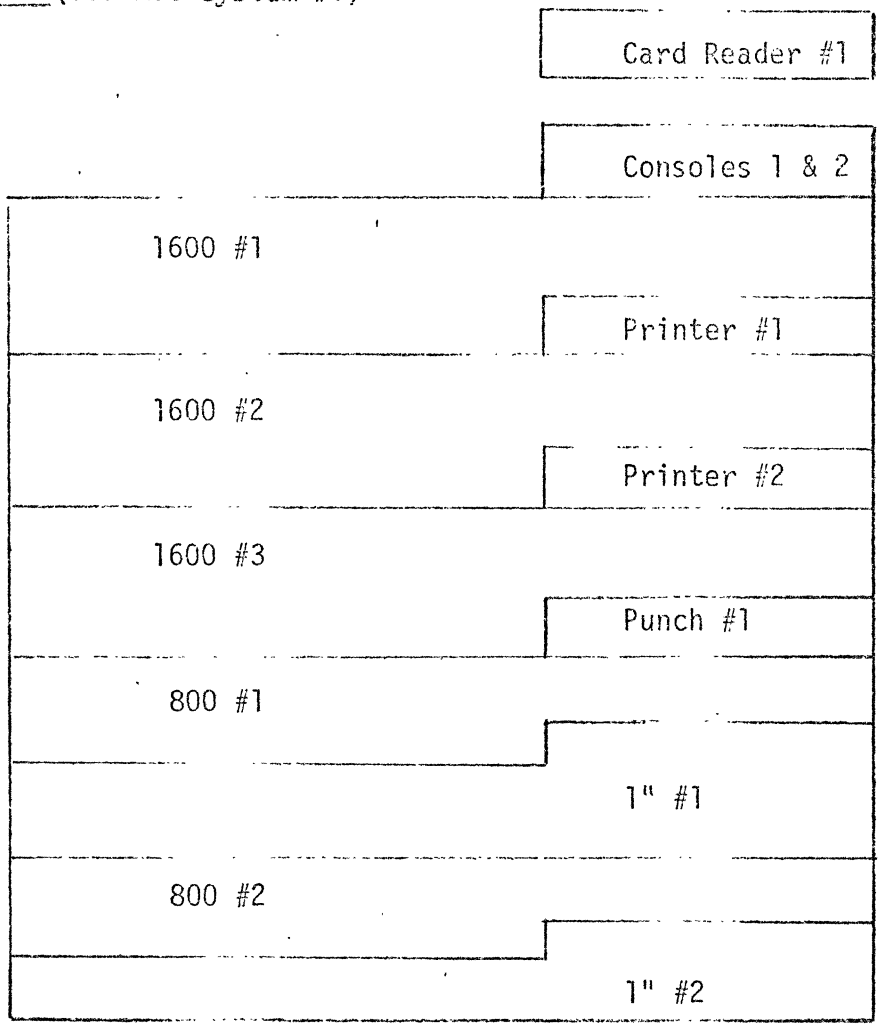
TAPE CONTROL (for ASC system # 1)



These bits are employed for the sharing of a VP by two tape handlers. There are three programs, called tape drivers, each of which operates at a functional level between the PCU programs and the tape handlers. Each driver can control two handler/device combinations. The Tape Control bits are polled by the driver programs, and the bits indicate that the associated transport or handler requires use of the shared VP. Driver #1 polls α_1 and β_1, Driver #2 polls α_2 and β_2, etc.

These bits are controlled entirely by the tape handlers except for one situation. When a PCU wants to initiate a tape handler, it does so by setting one of the Tape Control bits. The driver initiates a handler and upon task completion, the handler will reset the bit.

I/O DEVICES (for ASC System #1)



These CR's are used for communication between the device handler programs and the devices. These bits are controlled and interpreted by the handlers and devices only.

CP-PP COMMUNICATION

The following pages define the direct communication link between the Central Processor and the Peripheral Processor.

Software has the option of monitoring the CP requests for attention via polling loops, an interrupt structure, or a combination of the two. The options, once selected, are "hard wired" and not under program control.

Interrupts may be selected for each or any of the following conditions:

(1) System Errors (Parity or Breakpoint)

(2) MCP Instructions

(3) MCW Instructions

(4) Error Conditions (Protect Violation, Illegal Operation, or Arithmetic Exception)

(5) Reason Codes ("Master Controller" busy conditions)

CP CONTROL AND CONDITION BYTES

CR	BYTE	
		UNIT REGISTER
OA	1	UR (0-7)
		CONTROL BYTE
OA	3	CA | CT | SA | SP | SR | TR | AC | AS
		RESPONSE BYTE
12	0	SE | AT | MC | SC | X | RZ (0-2)
		CONDITION BYTE
12	2	CC | AB | BP | PE | IL | AE | PV | RB

UNIT REGISTER: Loaded with contents of CP hard core status via 44nn CCR commands.

CONTROL BYTE: Set and reset by Master Controller for control of CP Automatic Switching. Bits CA, CT, SA, and SP are internal to Master Controller while bits SR, TR, AC, and AS are monitored by CP hardware.

RESPONSE BYTE: Set by CP hardware to inform the Master Controller of a service requirement. Reset by Master Controller after service performed.

CONDITION BYTE: Set by CP hardware to inform the Master Controller of the CP's condition. All bits but RB are reset by the Master Controller.

CODE INTERPRETATION

CA:	CP Available	Set by Master Controller when no CP step is primed. Indicates the need to pool for activity on the CP execution queue. Reset when a CP step is primed.
CT:	CP Test	Set by Master Controller to indicate CP control is being relinquished to MCD. When set, the Master Controller will not respond to any other activity in the CP Response or Condition bytes.
SA:	Step Active	Set by Master Controller when a CP step is initiated. Reset when a step terminates and no step is primed for execution.
SR:	System Reset	Set by Master Controller to initiate a CP reset. Must be reset before any other CP action is taken.
TR:	Terminate Request	Set by the Master Controller to terminate outstanding CCR or Automatic Switches in the CP Reset by the MC.
AC:	Allow Call	Set by the Master Controller to permit Automatic MCP and MCW calls. Reset to inhibit these calls. Should be reset anytime a CCR command is used that invalidates the next job step status defined by pointers 16, 17, and 28.
AS:	Allow Switch	Set by the Master Controller to permit automatic MCW and Error context switching. Reset to inhibit these switches.
SE:	System Error	Set by CP to indicate a Parity Error or Breakpoint Match during normal CP operation. Reset by Master Controller.
AT:	Attention	Set by the CP to indicate an abnormal termination (as defined by the Condition Byte) of an automatic call or switch (as defined by the MC and SC bits). Reset by the Master Controller.
MC:	Message Complete	Set by the CP to indicate the completion of an MCP or MCW. Reset by the Master Controller after operation on the message.

SC: Switch Complete — Set by the CP to indicate the completion of an MCW or Error switch. Reset by the Master Controller after priming the next switch.

RZ(0-2): Reason Codes — Set by the CP to inform the Master Controller of the following context switch conditions:

CODE	INTERPRETATION
000	NOOP
001	MCP inhibited by MC or SC bits being set.
010	MCW inhibited by MC or SC bits being set.
011	Error switch inhibited by MC or SC bits being set.
100	MCW inhibited by AC = 0.
101	MCP inhibited by AC = 0.
110	MCW inhibited by AS = 0.
111	Error switch inhibited by AS = 0.

The Master Controller resets these bits and sets the CP Rnn Bit via a CCR command after preparing for the indicated condition.

CC: Command Complete — Set by the CP to indicate the completion of the last requested CP CCR command. Reset by the Master Controller.

AB: Abnormal — Set by the CP to indicate the last requested CP CCR command terminated abnormality. Reset by the Master Controller.

BP: Breakpoint — Set by the CP to inform the Master Controller of a CM Parity Error. Reset by the Master Controller.

PE: Parity Error — Set by the CP to inform the Master Controller of a CM Parity Error. Reset by the Master Controller.

IL:	Illegal Operation	Set by the CP to inform the Master Controller that an illegal operation code forced or attempted to force an Error context switch. Reset by the Master Controller.
AE:	Arithmetic Exception	Set by the CP to indicate that an arithmetic exception forced or attempted to force an Error context switch. Reset by the Master Controller.
PV:	Protect Violation	Set by the CP to indicate that a CM protect violation forced or attempted to force an Error context switch. Reset by the Master Controller.
RB:	Run Bit	Continuously gated CR bit reflecting the state of the CP's internal run bit.

PERIPHERAL PROCESSOR TIMING ANALYSIS

TABLE OF CONTENTS

EXECUTION TIMES

There are several factors which affect instruction execution times in the PPU. The effect of some of these variables is not amenable to quantitative analysis. However, if some assumptions are made, tables of execution times can be formulated. Then the effect of the assumptions can be considered qualitatively.

Table 1 gives time units for executing all PPU instructions. These tables are based on the following assumptions.

1. Where CM access is required, no memory interference occurs.

2. The VP has two diametrically opposed time slots. (Under this assumption each time unit in the Table is equal to 680 ns).

It will be noted that the table presents two values for skip or branch instructions. The first figure is for "fall through", and the second figure is for taking the skip or branch.

Effects of Assumptions

Assumption 1, is not unusual for execution time tables. Because there are multiple users of the CM, it is possible that a VP may not be able to acquire memory access immediately. The number of time units lost when this occurs and the frequency of occurrence are functions of the total system use of the CM.

Each VP communicates with CM via the SWB, and present estimates are that this requires 300 ns when no interference occurs. Since all VP units within a PPU share the same memory bus, a given VP may have to "wait its turn." Other VP units may be using the bus for program acquisition and/or data transfer.

Deviations from assumption 2. affect the execution times in two ways. First, and most obvious, when a different number of time slots are allocated to a VP the value of each time unit is altered. For example, if only one time slot is used, then each time unit is equal to 1360 ns. If four equally spaced time slots (0, 4, 8, 12) are used, then each time slot is equal to 340 ns. Note that if the time slots are not symetrically spaced, then the time unit values vary. For example, if time slots 0, 2, 8, 10 are assigned to a VP, then the time units are alternately 170 ns and 510 ns. These, of course, average out to the 340 ns figure used for four equally spaced slots, but the overall picture is changed due to the second way in which slot assignments affect execution times.

Time slot assignment can affect execution time due to the time unit becoming less than the 300 ns required for CP access. When the time unit is reduced to 170 ns by the assignment of alternate time slots, the number of time units given by the tables must be increased by one for every reference to CM.

In addition to these general considerations there is a specific class of instructions for which instruction times interact with memory access in another way. Instructions which store data into CM are considered completed when the data is delivered to the SWB even though the SWB may not have actually accessed CM yet. If successive instructions require CM access, they will be delayed until the SWB has completed the storage requested by the previous store instruction. Thus, in a memory interference environment, it is desirable to avoid following this class of instructions with any instruction which references CM.

Table 1. Execution Times

Instruction		Time Units from ROM			Time Units from CM		
		T = 0-7	T = 8-F	each additional level of indirect	T = 0-7	T = 8-F	each additional level of indirect
STORES							
ST	14	1	3	2	2	4	2
	1C	1	3	2	2	4	2
	90	1	3	2	1	4	2
	98	1	3	2	1	4	2
	10	1	3	2	2	4	2
	18	1	3	2	2	4	2
	94	1	3	2	1	4	2
	9C	1	3	2	1	4	2
STA	1E	1	3	2	2	4	2
	16	1	3	2	2	4	2
STH	97	1	-	-	1	-	-
	99	1	-	-	1	-	-
	95	1	-	-	1	-	-
	9D	1	-	-	1	-	-
STB	93	1	-	-	1	-	-
	9B	1	-	-	1	-	-
	97	1	-	-	1	-	-
	9F	1	-	-	1	-	-
STL	15	2	4	2	3	5	2
	1D	2	4	2	3	5	2
	11	2	4	2	3	5	2
	19	2	4	2	3	5	2
STR	17	2	4	2	3	5	2
	1F	2	4	2	3	5	2
	13	2	4	2	3	5	2
	1B	2	4	2	3	5	2
STF	1A	4	6	2	5	7	2
	3A	4	6	2	5	7	2
LOAD							
LD	04	2	4	2	2	4	2
	0C	2	4	2	2	4	2
	80	1	4	2	1	4	2
	88	1	4	2	1	4	2
	38	2	4	2	2	4	2
	08	2	4	2	2	4	2
	84	1	4	2	1	4	2
	8C	1	4	2	1	4	2
LDA	06	2	4	2	2	4	2
	0E	2	4	2	2	4	2
LDH	81	1	-	-	1	-	-
	89	1	-	-	1	-	-
	85	1	-	-	1	-	-
	8D	1	-	-	1	-	-
LDB	83	1	-	-	1	-	-
	8B	1	-	-	1	-	-

Table 1. Cont'd.

Instruction		Time Units from ROM			Time Units from CM		
		T = 0-7	T = 8-F	each additional level of indirect	T = 0-7	T = 8-F	each additional level of indirect
LDB	87	1	-	-	1	-	-
	8F	1	-	-	1	-	-
LDL	05	2	4	2	2	4	2
	0D	2	4	2	2	4	2
	39	2	4	2	2	4	2
	09	2	4	2	2	4	2
LDR	07	2	4	2	2	4	2
	0F	2	4	2	2	4	2
	3B	2	4	2	2	4	2
	0B	2	4	2	2	4	2
LDF	0A	5	7	2	5	7	2
	2A	5	7	2	5	7	2
ADD							
AD	50	2	4	2	2	4	2
	D0	1	4	2	1	4	2
ADH	D1	1	-	-	1	-	-
ADB	D3	1	-	-	1	-	-
ADL	51	2	4	2	2	4	2
R	53	2	4	2	2	4	2
SUBTRACT							
SU	54	2	4	2	2	4	2
	D4	1	4	2	1	4	2
SUH	D5	1	-	-	1	-	-
SUB	D7	1	-	-	1	-	-
SUL	55	2	4	2	2	4	2
SUR	57	2	4	2	2	4	2
LOGICAL--OR							
OR	44	2	4	2	2	4	2
	C4	1	4	2	1	4	2
	E4	1	4	2	1	4	2
ORH	C5	1	-	-	1	-	-
	E5	1	-	-	1	-	-
ORB	C7	1	-	-	1	-	-
	E7	1	-	-	1	-	-
ORL	45	2	4	2	2	4	2
ORR	47	2	4	2	2	4	2
LOGICAL--AND							
	40	2	4	2	2	4	2
	C0	1	4	2	1	4	2
	E0	1	4	2	1	4	2
ANH	C1	1	-	-	1	-	-
	E1	1	-	-	1	-	-

Table 1. Cont'd.

Instruction		Time Units from ROM			Time Units from CM		
		T = 0-7	T = 8-F	each additional level of indirect	T = 0-7	T = 8-F	each additional level of indirect
ANB	C3	1	-	-	1	-	-
	E3	1	-	-	1	-	
ANL	41	2	4	2	2	4	2
ANR	43	2	4	2	2	4	2
LOGICAL--EXCLUSIVE OR							
EX	4C	2	4	2	2	4	2
	CC	1	4	2	1	4	2
	EC	1	4	2	1	4	2
EXH	CD	1	-	-	1	-	-
	ED	1	-	-	1	-	-
EXB	CF	1	-	-	1	-	-
	EF	1	-	-	1	-	-
EXL	4D	2	4	2	2	4	2
EXR	4F	2	4	2	2	4	2
LOGICAL--EQUIVALENCE							
EQ	48	2	4	2	2	4	2
	C8	1	4	2	1	4	2
	E8	1	4	2	1	4	2
EQH	C9	1	-	-	1	-	-
	E9	1	-	-	1	-	-
EQB	CB	1	-	-	1	-	-
	EB	1	-	-	1	-	-
EQL	49	2	4	2	2	4	2
EQR	4B	2	4	2	2	4	2
COMPARE							
CE	30	2/3	4/5	2/2	2/3	4/5	2/2
	D8	1/2	4/5	2/2	1/2	4/5	2/2
	F8	1/2	4/5	2/2	1/2	4/5	2/2
CEH	D9	1/2	-	-	1/2	-	-
	F9	1/2	-	-	1/2	-	-
CEB	DB	1/2	-	-	1/2	-	-
	FB	1/2	-	-	1/2	-	-
CEL	31	2/3	4/5	2/2	2/3	4/5	2/2
CER	33	2/3	4/5	2/2	2/3	4/5	2/2
CN	34	2/3	4/5	2/2	2/3	4/5	2/2
	DC	1/2	4/5	2/2	2/3	4/5	2/2
	FC	1/2	4/5	2/2	1/2	4/5	2/2
CNH	DD	1/2	-	-	1/2	-	-
	FD	1/2	-	-	1/2	-	-
CNB	DF	1/2	-	-	1/2	-	-
	FF	1/2	-	-	1/2	-	-
CNL	35	2/3	4/5	2/2	2/3	4/5	2/2
CNR	37	2/3	4/5	2/2	2/3	4/5	2/2

Table 1. Cont'd

Instruction			Time Units from ROM			Time Units from CM		
			T = 0-7	T = 8-F	each additional level of indirect	T = 0-7	T = 8-F	each additional level of indirect
STACK								
PUSH		58	3/10	5/12	2/2	3/10	5/12	2/2
PULL		59	3/9	5/11	2/2	3/9	5/11	2/2
MOD		5B	3/9	5/11	2/2	3/9	5/11	2/2
MISC								
LDEA		5D	1	3	2	1	3	2
ANAZ	(4)	5F	3+	5+	2	3+	5+	2
POLL		F5	1/2	-	-	1/2	-	-
EXEC	(3)	5C	2+	4+	2	2+	4+	2
LDMB		F4	1	4	2	1	4	2
NOP		00	1	-	-	1	-	-
IMMEDIATES								
LDI		72	1	-	-	1	-	-
		62	1	-	-	1	-	-
LDHI		76	1	-	-	1	-	-
		66	1	-	-	1	-	-
LDBI		7E	1	-	-	1	-	-
		6E	1	-	-	1	-	-
RMI		65	1	-	-	1	-	-
ORBI		67	1	-	-	1	-	-
ANHI		61	1	-	-	1	-	-
ANBI		63	1	-	-	1	-	-
EXHI		6D	1	-	-	1	-	-
EXBI		6F	1	-	-	1	-	-
EQHI		69	1	-	-	1	-	-
EQBI		6B	1	-	-	1	-	-
ADI		70	1	-	-	1	-	-
ADHI		71	1	-	-	1	-	-
ADBI		73	1	-	-	1	-	-
SUI		74	1	-	-	1	-	-
SUHI		75	1	-	-	1	-	-
SUBI		77	1	-	-	1	-	-
CEI		78	1/2	-	-	1/2	-	-
CEHI		79	1/2	-	-	1/2	-	-
CEBI		7B	1/2	-	-	1/2	-	-
CNI		7C	1/2	-	-	1/2	-	-
CNHI		7D	1/2	-	-	1/2	-	-
CNBI		7F	1/2	-	-	1/2	-	-
SET/RESET CR BITS								
SL		FA	1	-	-	1	-	-
SR		FE	1	-	-	1	-	-
RL		F2	1	-	-	1	-	-
RR		F6	1	-	-	1	-	-

Table 1. Cont'd

Instruction		Time Units from ROM			Time Units from CM		
		T = 0-7	T = 8-F	each additional level of indirect	T = 0-7	T = 8-F	each additional level of indirect
TEST CR UNDER MASK AND SKIP							
TOL	CA	1/2	-	-	1/2	-	-
TOR	CE	1/2	-	-	1/2	-	-
TZL	C2	1/2	-	-	1/2	-	-
TZR	C6	1/2	-	-	1/2	-	-
TAOL	EA	1/2	-	-	1/2	-	-
TAOR	EE	1/2	-	-	1/2	-	-
TAZL	E2	1/2	-	-	1/2	-	-
TAZR	E6	1/2	-	-	1/2	-	-
SHIFT							
SHL (1)	64	0+	-	-	0+	-	-
SHA (1)	60	0+	-	-	0+	-	-
SHC (2)	6C	0+	-	-	0+	-	-
TEST & SET							
TSZL	D2	1/2	-	-	1/2	-	-
TSOL	DA	1/2	-	-	1/2	-	-
TRZL	92	1/2	-	-	1/2	-	-
TROL	9A	1/2	-	-	1/2	-	-
TSZR	D6	1/2	-	-	1/2	-	-
TSOR	DE	1/2	-	-	1/2	-	-
TRZR	96	1/2	-	-	1/2	-	-
TROR	9E	1/2	-	-	1/2	-	-
ARITHMETIC TEST (CONDITIONAL BRANCHES)							
TZ	B0	1/3	1/5	0/2	1/3	1/5	0/2
	A0	1/3	1/5	0/2	1/3	1/5	0/2
TZH	B1	1/3	1/5	0/2	1/3	1/5	0/2
	A1	1/3	1/5	0/2	1/3	1/5	0/2
TZB	B3	1/3	1/5	0/2	1/3	1/5	0/2
	A3	1/3	1/5	0/2	1/3	1/5	0/2
TN	B4	1/3	1/5	0/2	1/3	1/5	0/2
	A4	1/3	1/5	0/2	1/3	1/5	0/2
TNH	B5	1/3	1/5	0/2	1/3	1/5	0/2
	A5	1/3	1/5	0/2	1/3	1/5	0/2
TNB	B7	1/3	1/5	0/2	1/3	1/5	0/2
	A7	1/3	1/5	0/2	1/3	1/5	0/2
TP	B8	1/3	1/5	0/2	1/3	1/5	0/2
	A8	1/3	1/5	0/2	1/3	1/5	0/2
TPH	B9	1/3	1/5	0/2	1/3	1/5	0/2
	A9	1/3	1/5	0/2	1/3	1/5	0/2
TPB	BB	1/3	1/5	0/2	1/3	1/5	0/2
	AB	1/3	1/5	0/2	1/3	1/5	0/2

Table 1. Cont'd

Instruction		Time Units from ROM			Time Units from CM		
		T = 0-7	T = 8-F	each a additional level of indirect	T = 0-7	T = 8-F	each additional level of indirect
TM	BC	1/3	1/5	0/2	1/3	1/5	0/2
	AC	1/3	1/5	0/2	1/3	1/5	0/2
TMH	BD	1/3	1/5	0/2	1/3	1/5	0/2
	AD	1/3	1/5	0/2	1/3	1/5	0/2
TMB	BF	1/3	1/5	0/2	1/3	1/5	0/2
	AF	1/3	1/5	0/2	1/3	1/5	0/2
INDEX MODIFY AND BRANCH							
IBZ	B2	1/3	1/5	0/2	1/3	1/5	0/2
IBN	B6	1/3	1/5	0/2	1/3	1/5	0/2
DBZ	BA	1/3	1/5	0/2	1/3	1/5	0/2
DBN	BE	1/3	1/5	0/2	1/3	1/5	0/2
BRANCH UNCONDITIONAL							
BPCS	AE	2	4	2	2	4	2
BRS	46	2	4	2	2	4	2
BCS	12	2	4	2	2	4	2
BCAS	52	2	4	2	2	4	2
BPC	5A	2	4	2	2	4	2
	5E	2	4	2	2	4	2
	42	2	4	2	2	4	2
BC	02	2	4	2	2	4	2
	32	2	4	2	2	4	2
BCA	4A	2	4	2	2	4	2
	4E	2	4	2	2	4	2
BRSM	56	3	5	2	3	5	2
VP SET/RESET FLAG							
VPS	86	1	-	-	1	-	-
VPR	82	1	-	-	1	-	-
VPTO	8E	1/2	-	-	1/2	-	-
VPTZ	8A	1/2	-	-	1/2	-	-

(1) A shift of 0 places takes 1 time slot. For nonzero cases, the total shift is made up of a series of incremental shifts of 1, 4, or 8-bit positions. For each increment required to make up the total shift, count 1 time slot. For example, a shift of 15 requires increments of 8, 4, 1, 1, 1. Thus 5 time slots are required to shift 15 places. Worst case time is 7 time slots for a shift of 31 places.

(2) Similar to SHL & SHA except shift increments are 1, 4, 8, & 16.

(3) Table entry is for time required to acquire the instruction to be executed. Add the execution time for the acquired instruction.

(4) Add appropriate time units for each level of indirect addressing exhibited by the object instruction.

PERIPHERAL PROCESSOR

INSTRUCTION SET

TABLE OF CONTENTS

The Peripheral Processor instructions are described in the following sixteen groups;

* STORE INSTRUCTIONS move data from a VPR or CR to a CR, VPR, or to a CM location.

* LOAD INSTRUCTIONS move data from a CR, VPR, or from a CM location to a VPR or CR.

* ARITHMETIC INSTRUCTIONS add or subtract CM or VPR data to or from VPR data and place the results in the VPR.

* LOGICAL INSTRUCTIONS perform logical operations between VPR data and CM, CR, or VPR data and place the results in the VPR.

* COMPARE AND SKIP INSTRUCTIONS compare CM, CR, or VPR data with VPR data and can perform a program skip, depending on the results.

* STACK INSTRUCTIONS store or retrieve operands in a reserved area of memory into which operands are stored (pushed) and retrieved (pulled) on a last-in, first-out basis.

* MISCELLANEOUS INSTRUCTIONS include six various instructions.

* IMMEDIATE INSTRUCTIONS perform load, logical, arithmetic, and compare operations with the immediate operand.

* SET/RESET CR BITS INSTRUCTIONS set or reset bits in a CR.

* TEST CR UNDER MASK AND SKIP INSTRUCTIONS test bits in a CR and perform a skip if the test is satisfied.

* SHIFT INSTRUCTIONS shift the contents of a VPR a specified number of bits.

* TEST AND SET INSTRUCTIONS test bits in a CR and perform a program skip if the test is satisfied. Independent of the test results, the specified bit(s) are set or reset.

* ARITHMETIC TEST INSTRUCTIONS test a VPR or a CR and perform a conditional branch if the test is satisfied.

* INDEX MODIFY AND BRANCH INSTRUCTIONS modify the contents of a VPR, test the result and perform a program branch if the test is satisfied.

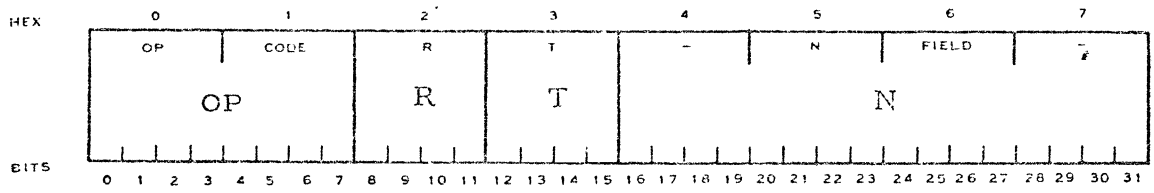
* BRANCH UNCONDITIONAL INSTRUCTIONS provide unconditional branching.

* VP FLAG INSTRUCTIONS modify or test a CR bit, and the addressed bit is a function of the identity of the VP executing the instruction.

WORD FORMAT

INSTRUCTION WORD FORMAT

The instruction word is divided into four fields;



OP FIELD

The OP field, consisting of eight bits and represented by the two digit hexadecimal OP code, always specifies the operation to be performed. Most operations fall into families of three where numbers of the family specify whole word, half word, or byte class of operation.

R FIELD

The R field usually specifies a location in a VPR or CR file.

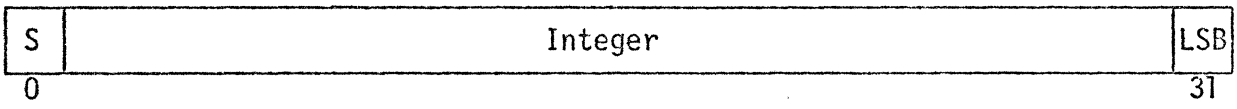
T,N FIELD

The T,N fields function together to specify an immediate operand, an operand address, or a branch address.

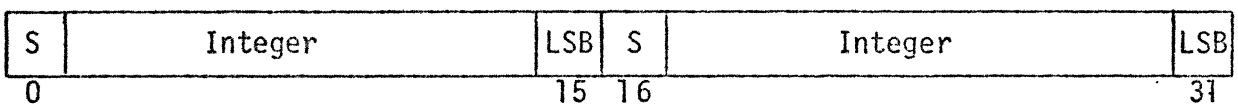
DATA WORD FORMAT

All arithmetic instructions interpret data in two's complement representation for negative numbers, and overflow are not detected. Data formats are as follows;

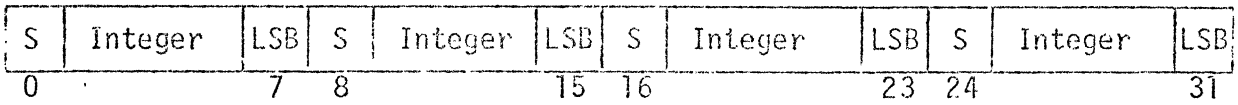
whole word



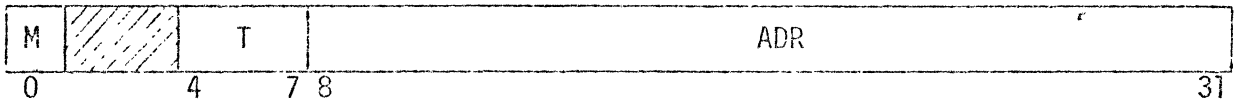
half word



byte



Indirect Cell Format



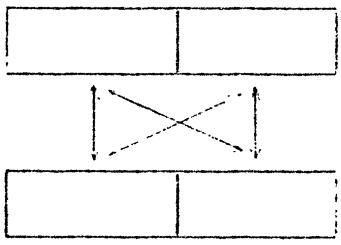
The indirect cell word may specify an operand address or a branch address.

OPERAND SOURCE - DESTINATION RELATIONSHIP

The operands specified by the R field and by the T,N fields may be whole word, half word, or byte operands depending on the operation. When half word or byte addressing is used, the combinations of source and destination locations are grouped to the source-destination relationship of both operands in CR or VPR, one operand in CM, and immediate operand.

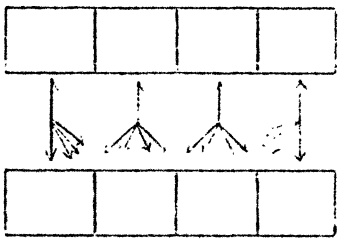
BOTH OPERANDS IN CR OR VPR

When both operand locations are in the VPR or the CR files, the two halfwords or bytes may be independently taken from any position within the VPR or CR.



VPR or CR HALFWORD

VPR or CR HALFWORD

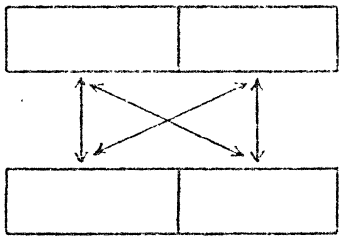


VPR or CR BYTE

VPR or CR BYTE

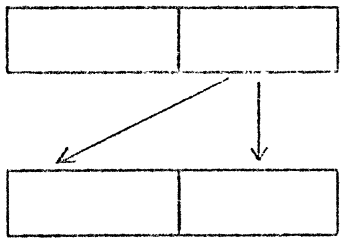
ONE OPERAND IN CM

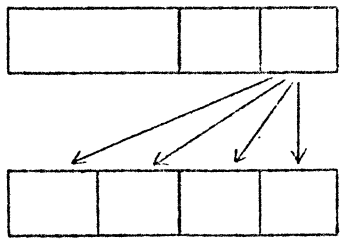
When one of the operand location is in CM and the other is in the VPR or the CR files, byte addressing does not exist. Halfword locations may be independently taken from any position within the CM word and the VPR or CR.



IMMEDIATE OPERAND

When the operand source is an immediate operand, and the destination is either a VPR or a CR, the destination may be in any halfword or byte position. The operand source is always the least significant byte or halfword.





R FIELD ADDRESSING

The R field can address a Virtual Processor Register or a Communications Register as specified by the operation.

VPR ADDRESSING

When the OP code specifies that the R field is addressing a VPR, the VPR location is determined as follows:

R FIELD	ADDRESSING LEVEL		
	BYTE	HALF WORD	WORD
0	VPR_0, H_{0-1}	VPR_0, H_{0-3}	VPR_0
1	VPR_0, H_{2-3}		
2	VPR_0, H_{4-5}	VPR_0, H_{4-7}	
3	VPR_0, H_{6-7}		
4	VPR_1, H_{0-1}	VPR_1, H_{0-3}	VPR_1
5	VPR_1, H_{2-3}		
6	VPR_1, H_{4-5}	VPR_1, H_{4-7}	
7	VPR_1, H_{6-7}		
8	VPR_2, H_{0-1}	VPR_2, H_{0-3}	VPR_2
9	VPR_2, H_{2-3}		
A	VPR_2, H_{4-5}	VPR_2, H_{4-7}	
B	VPR_2, H_{6-7}		
C	VPR_3, H_{0-1}	VPR_3, H_{0-3}	VPR_3
D	VPR_3, H_{2-3}		
E	VPR_3, H_{4-5}	VPR_3, H_{4-7}	
F	VPR_3, H_{6-7}		

CR ADDRESSING

When the operation specifies that the R field is addressing a CR, the address is developed by adding the R field and the least significant byte of VPR_3. The address developed references one of the 64 CR file words to the byte, half word, or whole word level in a manner similar to VPR addressing. The contents of VPR_3 remain unchanged.

T,N FIELD ADDRESSING

The T,N field can be developed into an immediate operand, a direct or indirect operand address, or a direct or indirect branch address as specified by the operation. A few operations require that the usual T,N development is modified by "augmenting". Augmenting is defined as replacing the three LSB's of the effective address with the identity of the VP executing the instruction. When augmenting occurs in combination with indirect addressing, only the first level indirect address is augmented. The T field may specify indexed and/or indirect addressing. Any VPR half word except for the left half of VPR_0 may be designated as the index register by the T field.

The eight types of T,N field development and indirect addressing are shown in tabular form, using the following symbolic notations:

n^{16} - the 16 bits of the N field

n^{24} or n^{32} - a 24 or 32 bit signed number developed, by sign extension, from the 16 bit N field.

b^{24} - the 24 bits of the CM base register located in the CR file.

t^{24} or t^{32} - a 24 or 32 bit signed number developed, by sign extension, from the VPR half word designated by the three LSB's of the T field.

pc^{24} - the 24 bit program counter address which referenced the current instruction.

ADR^{24} - the 24 bits of the ADR field in an indirect cell.

IMMEDIATE OPERANDS

T	Operand
0,8	n^{32}
1-7, 9-F	$n^{32} + t^{32}$

OPERAND ADDRESSES, α

CM Operands (word indexing)

T	α
0	$n^{16} + b^{24}$
1-7	$n^{16} + b^{24} + t^{24}$
8	$(n^{16} + b^{24})$
9-F	$(n^{16} + b^{24} + t^{24})$

CM Absolute Operands (word indexing)

T	α
0	n^{16}
1-7	$n^{16} + t^{24}$
8	(n^{16})
9-F	$(n^{16} + t^{24})$

VPR or CR Operands (byte indexing)

T	α
0	n^{16}
1-7	$n^{16} + t^{24}$
8	(n^{16})
9-F	$(n^{16} + t^{24})$

Note: The LSB's (4 for VPR operand, 8 for CR operand) reference the registers in a manner identical to R field addressing.

BRANCH ADDRESSES, β

PC Relative (word indexing)

T	β	
0	$n^{24} + pc^{24} + 1$	β references same memory
1-7	$n^{24} + pc^{24} + 1 + t^{24}$	
8	$(n^{24} + pc^{24} + 1)$	first indirect address references CM. For terminal address, see indirect address development.
9-F	$(n^{24} + pc^{24} + 1 + t^{24})$	

ROM (word indexing)

T	β	
0	n^{16}	β references ROM
1-7	$n^{16} + t^{24}$	
8	(n^{16})	first indirect address references VPR file. For terminal address, see indirect address development.
9-F	$(n^{16} + t^{24})$	

Base Relative (word indexing)

T	β	
0	$n^{16} + b^{24}$	references CM
1-7	$n^{16} + b^{24} + t^{24}$	references CM
8	$(n^{16} + b^{24})$	first indirect address references CM. For terminal address, see indirect address development
9-F	$(n^{16} + b^{24} + t^{24})$	first indirect address references CM. For terminal address, see indirect address development

Absolute (word indexing)

T	β	
0	n^{16}	references CM
1-7	$n^{16} + t^{24}$	references CM
8	(n^{16})	first level indirect references CM. For terminal addresses, see indirect address development.
9-F	$(n^{16} + t^{24})$	first level indirect references CM. For terminal addresses, see indirect address development.

INDIRECT ADDRESSES

Multi-level indirect addressing and indexing is possible in the PPU instruction set. As shown on page 2, each indirect cell contains a T field which is interpreted in a manner similar to the T field in an instruction word.

The first level indirect address developed from the original T,N fields may reference CM, VPR or CR, depending on the T,N development by the operation. Additional levels of indirect addressing always reference CM. Terminal operand addresses always reference CM, but terminal branch addresses may reference either CM or ROM. If the MSB of the final indirect cell is "1", then the branch address references CM. If the MSB is "0", then the branch address references ROM.

In some instructions, indirect addressing is undefined, and is noted in the instruction descriptions. For these instructions, the indirect tag bit is ignored.

Operand Address (word indexing)

T	α
0	ADR^{24}
1-7	$ADR^{24} + t^{24}$
8	(ADR^{24})
9-F	$(ADR^{24} + t^{24})$

Branch Address (word indexing)

T	β	
0	ADR^{24}	If MSB of indirect cell = 0, references ROM. If MSB of indirect cell = 1, references CM.
1-7	$ADR^{24} + t^{24}$	
8	(ADR^{24})	MSB of indirect cell ignored.
9-F	$(ADR^{24} + t^{24})$	

INSTRUCTION DESCRIPTION FORMAT

The PPU instructions are presented in the following format:

Symbolic Statement
(Table 1.)

This paragraph contains a general discussion of the instruction type.

Op Code	*R Field*	*T,N Field*
Op Code number	R field specified location	T,N field specified location

A discussion of this operation may be required.

Op Code	*R Field*	*T,N Field*
Op Code number	R field specified location	T,N field specified location

A discussion of this operation may be required.

Table 1. Glossary of Symbolic Terms used in PPU Instructions

Term	Meaning
α	operand address developed by the interpretation of the T and N fields.
β	branch location developed by the interpretation of the T and N fields.
(location)	the contents of "location" where location is an address.
r	a register; either VPR or CR
VPR	virtual processor register
CR	communication register
$\cdot\rightarrow$	"is transferred to"
+	plus (arithmetic addition)
-	minus (arithmetic subtraction)
OR	logical OR function
$\cdot$	logical AND function
$\oplus$	logical EXCLUSIVE OR function
$\odot$	logical EQUIVALENCE function
PC	program counter containing the address of the current instruction
=	equal (not "is replaced by" in this document)
$\neq$	not equal
EA	effective address developed by the interpretation of the T and N fields.
R	R field of the instruction
R_i	the $i\underline{th}$ bit of the R field
$(r)_i$	the $i\underline{th}$ bit of a hex character contained in a register
Σ	"the logical summation of"
π	"the logical product of"
$\geq$	equal or greater than
<	less than

STORE WORD (ST)

(r) → α

The operand specified by the R field is stored in the location specified by the T,N field.

Op Code	*R Field*		*T,N Field*
14	VPR	→	CM
1C	VPR	→	CM augmented
90	VPR	→	VPR
98	VPR	→	CR
10	CR	→	CM
18	CR	→	CM augmented
94	CR	→	VPR
9C	CR	→	CR

STORE WORD ABSOLUTE (STA)

(r) → α

The operand specified by the R field is stored in the location specified by the T,N field.

Op Code	*R Field*		*T,N Field*
1E	VPR	→	CM absolute, agmented
16	VPR	→	CM absolute

STORE HALFWORD (STH)

(r) → α

The operand specified by the R field is stored in the location specified by the T,N field. Indirect addressing is undefined.

Op Code	*R Field*		*T,N Field*
91	VPR	→	VPR
99	VPR	→	CR
95	CR	→	VPR
9D	CR	→	CR

STORE BYTE (STB)

(r) → α

The operand specified by the R field is stored in the location specified by the T,N field. Indirect addressing is undefined.

Op Code	*R Field*		*T,N Field*
93	VPR	→	VPR
9B	VPR	→	CR
97	CR	→	VPR
9F	CR	→	CR

STORE LEFT HALFWORD (STL)

(r) → α

The operand specified by the R field is moved to the left half of the memory location specified by the T,N field.

Op Code	*R Field*		*T,N Field*
15	VPR	→	CM
1D	VPR	→	CM augmented
11	CR	→	CM
19	CR	→	CM augmented

STORE RIGHT HALFWORD (STR)

(r) → α

The operand specified by the R field is stored in the right half of the location specified by the T,N field.

Op Code	*R Field*		*T,N Field*
17	VPR	→	CM
1F	VPR	→	CM augmented
13	CR	→	CM
1B	CR	→	CM augmented

STORE VP FILE (STF)

$(r_0) \rightarrow \alpha$

$(r_1) \rightarrow \alpha + 1$

$(r_2) \rightarrow \alpha + 2$

$(r_3) \rightarrow \alpha + 3$

The contents of all four VPRs are stored into four consecutive locations in CM. T,N field development is modified such that the first of the four CM addresses is forced to be a multiple of four.

Op Code	*R Field*	*T,N Field*
1A	(not used)	CM augmented

If T = 0 - 7, the 2 LSB's of the effective address are forced to zero and augmenting occurs in the 3 bits adjacent to the LSB's. If T = 8 - F, augmenting occurs normally, however, the 2 LSB's of the effective address indirectly acquired are forced to zero.

Op Code	*R Field*	*T,N Field*
2A	(not used)	CM

If T = 0 - 7, the two least significant bits of the effective address are forced to zero. If T = 8 - F, indirect addressing occurs normally, however, the two least significant bits of the effective address indirectly acquired are forced to zero.

LOAD WORD (LD)

$(\alpha) \rightarrow r$

The operand indicated by the T,N field is loaded into the register specified by the R field.

Op Code	*R Field*		*T,N Field*
04	VPR	←	CM
0C	VPR	←	CM augmented
80	VPR	←	VPR
88	VPR	←	CR
38	CR	←	CM
08	CR	←	CM augmented
84	CR	←	VPR
8C	CR	←	CR

LOAD WORD ABSOLUTE (LDA)

$(\alpha) \rightarrow r$

The operand indicated by the T,N field is loaded into the register indicated by the R field.

Op Code	*R Field*		*T,N Field*
0E	VPR	←	CM absolute augmented
06	VPR	←	CM absolute

LOAD HALFWORD (LDH)

$(\alpha) \rightarrow$

The operand indicated by the T,N field is loaded into the register specified by the R field. Indirect addressing is undefined.

Op Code	*R Field*		*T,N Field*
81	VPR	←	VPR
89	VPR	←	CR
85	CR	←	VPR
8D	CR	←	CR

LOAD BYTE (LDB)

$(\alpha) \rightarrow r$

The operand indicated by the T,N field is loaded into the register specified by the R field. Indirect addressing is undefined.

Op Code	*R Field*		*T,N Field*
83	VPR	←	VPR
8B	VPR	←	CR
87	CR	←	VPR
8F	CR	←	CR

LOAD LEFT HALFWORD (LDL)

$(\alpha) \rightarrow r$

The left half of the operand indicated by the T,N field is loaded into the register specified by the R field.

Op Code	*R Field*		*T,N Field*
05	VPR	←	CM
0D	VPR	←	CM augmented
39	CR	←	CM
09	CR	←	CM augmented

LOAD RIGHT HALFWORD (LDR)

$$(\alpha) \rightarrow r$$

The right half of the operand indicated by the T,N field is loaded into the half of the register specified by the R field.

LOAD VP FILE (LDF)

$$(\alpha) \rightarrow r_0$$
$$(\alpha + 1) \rightarrow r_1$$
$$(\alpha + 2) \rightarrow r_2$$
$$(\alpha + 3) \rightarrow r_3$$

The four VPR's are loaded from four consecutive CM locations. T,N field development is modified such that the first of the four CM addresses is forced to be a multiple of four.

Op Code	*R Field*		*T,N Field*
07	VPR	←	CM
0F	VPR	←	CM augmented
3B	CR	←	CM
0B	CR	←	CM augmented

Op Code	*R Field*	*T,N Field*
0A	(not used)	CM augmented

If T = 0 - 7, the 2 LSB's of the effective address are forced to zero and augmenting occurs in the 3 bits adjacent to the LSB's. If T = 8 - F, augmenting occurs normally, however, the 2 LSB's of the effective address indirectly acquired are forced to zero.

Op Code	*R Field*	*T,N Field*
3A	(not used)	CM

If T = 0 - 7, the 2 LSB's of the effective address are forced to zero. If T = 8 - F, indirect addressing occurs normally, however, the two least significant bits of the effective address indirectly acquired are forced to zero.

ADD WORD (AD)

$$(r) + (\alpha) \rightarrow r$$

The operand specified by the T,N field is added to the contents of the VPR specified by the R field and the result replaces the contents of the VPR. Overflows are ignored.

Op Code	*R Field*	*T,N Field*
50	VPR	CM
D0	VPR	VPR

ADD HALFWORD (ADH)

$$(r) + (\alpha) \rightarrow r$$

The halfword specified by the T,N field is added to the contents of the VPR halfword specified by the R field and the result replaces the contents of the VPR halfword. Overflows are ignored. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
D1	VPR	VPR

ADD BYTE (ADB)

$$(r) + (\alpha) \rightarrow r$$

The byte indicated by the T,N field is added to the contents of the VPR byte specified by the R field and the result replaces the contents of the VPR byte. Overflows are ignored. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
D3	VPR	VPR

ADD LEFT HALFWORD (ADL)

$$(r) + (\alpha) \rightarrow r$$

The left half of the operand specified by the T,N field is added to the contents of the VPR halfword specified by the R field and the result replaces the contents of the VPR halfword. Overflows are ignored.

Op Code	*R Field*	*T,N Field*
51	VPR	CM

ADD RIGHT HALFWORD (ADR)

$(r) + (\alpha) \rightarrow r$

The right half of the operand specified by the T,N field is added to the contents of a VPR halfword specified by the R field and the result replaces the contents of the VPR halfword. Overflows are ignored.

Op Code	*R Field*	*T,N Field*
53	VPR	CM

SUBTRACT WORD (SU)

$(r) - (\alpha) \rightarrow r$

The operand specified by the T,N field is subtracted from the contents of the VPR specified by the R field and the result replaces the contents of the VPR. Overflows are ignored.

Op Code	*R Field*	*T,N Field*
54	VPR	CM
D4	VPR	VPR

SUBTRACT HALFWORD (SUH)

$(r) - (\alpha) \rightarrow r$

The halfword specified by the T,N field is subtracted from the contents of the VPR halfword specified by the R field and the result replaces the contents of the VPR halfword. Overflows are ignored. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
D5	VPR	VPR

SUBTRACT BYTE (SUB)

$(r) - (\alpha) \rightarrow r$

The byte specified by the T,N field is subtracted from the contents of the VPR byte specified by the R field, and the result replaces the contents of the VPR byte. Overflows are ignored. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
D7	VPR	VPR

SUBTRACT LEFT HALFWORD (SUL)

$(r) - (\alpha) \rightarrow r$

The left half of the operand specified by the T,N field is subtracted from the contents of the VPR halfword specified by the R field and the result replaces the contents of the VPR halfword. Overflows are ignored.

Op Code	*R Field*	*T,N Field*
55	VPR	CM

SUBTRACT RIGHT HALFWORD (SUR)

$$(r) - (\alpha) \rightarrow r$$

The right half of an operand specified by the T,N field is subtracted from the contents of a VPR halfword specified by the R field and the result replaces the contents of the VPR halfword. Overflows are ignored.

Op Code	*R Field*	*T,N Field*
57	VPR	CM

LOGICAL OR WORD (OR)

(r) OR (α) → r

The operand specified by the T,N field is logically combined using the OR function with the operand specified by the R field and the result replaces the contents of the VPR.

Op Code	*R Field*	*T,N Field*
44	VPR	CM
C4	VPR	VPR
E4	VPR	CR

LOGICAL OR HALFWORD (ORH)

(r) OR (α) → r

The operand halfword specified by the T,N field is logically combined using the OR function with the operand halfword specified by the R fi eld and the result replaces the contents of the VPR halfword. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
C5	VPR	VPR
E5	VPR	CR

LOGICAL OR BYTE (ORB)

(r) OR (α) → r

The operand byte specified by the T,N field is logically combined using the OR function with the operand byte specified by the R field and the result replaces the contents of the VPR byte. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
C7	VPR	VPR
E7	VPR	CR

LOGICAL OR LEFT HALFWORD (ORL)

(r) OR (α) → r

The left half of the operand word specified by the T,N field is logically combined using the OR function with the halfword specified by the R field and the result replaces the contents of the VPR.

Op Code	*R Field*	*T,N Field*
45	VPR	CM

LOGICAL OR RIGHT HALFWORD (ORR)

(r) OR (α) → r

The right half of the operand word specified by the T,N field is logically combined using the OR function with the halfword specified by the R field and the result replaces the contents of the VPR halfword.

Op Code	*R Field*	*T,N Field*
47	VPR	CM

LOGICAL AND WORD (AN)

$(r) \cdot (\alpha) \rightarrow r$

Op Code	*R Field*	*T,N Field*
40	VPR	CM
C0	VPR	VPR
E0	VPR	CR

The operand specified by the T,N fields is logically combined using the AND function with the operand specified by the R field and the result replaces the contents of the VPR.

LOGICAL AND HALFWORD (ANH)

$(r) \cdot (\alpha) \rightarrow r$

Op Code	*R Field*	*T,N Field*
C1	VPR	VPR
E1	VPR	CR

The operand halfword specified by the T,N field is logically combined using the AND function with the halfword in a VPR specified by the R field. The result replaces the contents of the VPR halfword. Indirect addressing is undefined.

LOGICAL AND BYTE (ANB)

$(r) \cdot (\alpha) \rightarrow r$

Op Code	*R Field*	*T,N Field*
C3	VPR	VPR
E3	VPR	CR

The operand byte specified by the T,N field is logically combined using the AND function with the operand byte specified by the R field. The result replaces the contents of the VPR byte. Indirect addressing is undefined.

LOGICAL AND LEFT HALFWORD (ANL)

$(r) \cdot (\alpha) \rightarrow r$

Op Code	*R Field*	*T,N Field*
41	VPR	CM

The left half of the operand specified by the T,N field is logically combined using the AND function with the halfword in a VPR specified by the R field and the result replaces the contents of the VPR halfword.

LOGICAL AND RIGHT HALFWORD (ANR)

$(r) \cdot (\alpha) \rightarrow r$

Op Code	*R Field*	*T,N Field*
43	VPR	CM

The right half of the operand word specified by the T,N field is logically combined using the AND function with the VPR halfword specified by the R field. The result replaces the contents of the VPR halfword.

LOGICAL EXCLUSIVE OR WORD (EX)

$$(r) \oplus (\alpha) \rightarrow r$$

The operand specified by the T,N field is logically combined using the exclusive OR function with the operand specified by the R field. The result replaces the contents of the VPR.

Op Code	*R Field*	*T,N Field*
4C	VPR	CM
CC	VPR	VPR
EC	VPR	CR

LOGICAL EXCLUSIVE OR HALFWORD (EXH)

$$(r) \oplus (\alpha) \rightarrow r$$

The operand halfword specified by the T,N field is logically combined with the VPR halfword specified by the R field using the exclusive OR function. The result replaces the contents of the VPR halfword. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
CD	VPR	VPR
ED	VPR	CR

LOGICAL EXCLUSIVE OR BYTE (EXB)

$$(r) \oplus (\alpha) \rightarrow r$$

The operand byte specified by the T,N field is logically combined using the exclusive OR function with the operand byte specified by the R field. The result replaces the contents of the VPR byte. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
CF	VPR	VPR
EF	VPR	CR

LOGICAL EXCLUSIVE OR LEFT HALFWORD (EXL)

$$(r) \oplus (\alpha) \rightarrow r$$

The left half of the operand word specified by the T,N field is logically combined using the exclusive OR function with the halfword specified by the R field and the result replaces the contents of the VPR halfword.

Op Code	*R Field*	*T,N Field*
4D	VPR	CM

LOGICAL EXCLUSIVE OR RIGHT HALFWORD (EXR)

$$(r) \oplus (\alpha) \rightarrow r$$

The right half of the operand word specified by the T,N field is logically combined using the exclusive OR function with the halfword specified by the R field. The result replaces the contents of the VPR halfword.

Op Code	*R Field*	*T,N Field*
4F	VPR	CM

LOGICAL EQUIVALENCE WORD (EQ)

$$(r) \odot (\alpha) \rightarrow r$$

The operand specified by the T,N field is logically combined using the equivalence function with the operand in the VPR specified by the R field. The result replaces the contents of the VPR.

Op Code	*R Field*	*T,N Field*
48	VPR	CM
C8	VPR	VPR
E8	VPR	CR

LOGICAL EQUIVALENCE HALFWORD (EQH)

$$(r) \odot (\alpha) \rightarrow r$$

The operand halfword indicated by the T,N field is logically combined using the equivalence function with the operand halfword specified by the R field. The result replaces the contents of the VPR halfword. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
C9	VPR	VPR
E9	VPR	CR

LOGICAL EQUIVALENCE BYTE (EQB)

$$(r) \odot (\alpha) \rightarrow r$$

A byte specified by the T,N field is logically combined using the equivalence function with the byte in a VPR specified by the R field. The result replaces the contents of the VPR byte. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
CB	VPR	VPR
EB	VPR	CR

LOGICAL EQUIVALENCE LEFT HALFWORD (EQL)

$$(r) \odot (\alpha) \rightarrow r$$

The left half of the operand word specified by the T,N field is logically combined using the equivalence function with the halfword in a VPR specified by the R field. The result replaces the contents of the VPR halfword.

Op Code	*R Field*	*T,N Field*
49	VPR	CM

LOGICAL EQUIVALENCE RIGHT HALFWORD (EQR)

$$(r) \odot (\alpha) \rightarrow r$$

The right halfword of the operand specified by the T,N field is logically combined using the equivalence function with the VPR halfword specified by the R field. The result replaces the contents of the VPR halfword.

Op Code	*R Field*	*T,N Field*
4B	VPR	CM

COMPARE WORD, SKIP IF EQUAL (CE)

(PC) + 2 → PC if (r) = (α)

The contents of the register indicated by the T,N field is compared with the contents of the VPR indicated by the R field. A skip is made depending on the result.

Op Code	*R Field*	*T,N Field*
30	VPR	CM
D8	VPR	VPR
F8	VPR	CR

COMPARE HALFWORD, SKIP IF EQUAL (CEH)

(PC) + 2 → PC if (r) = (α)

The halfword indicated by the T,N field is compared with the contents of the VPR halfword indicated by the R field. A skip is made depending on the result. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
D9	VPR	VPR
F9	VPR	CR

COMPARE BYTE, SKIP IF EQUAL (CEB)

(PC) + 2 → PC if (r) = (α)

The byte indicated by the T,N field is compared with the contents of the VPR byte indicated by the R field. A skip is made depending on the result. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
DB	VPR	VPR
FB	VPR	CR

COMPARE LEFT HALFWORD, SKIP IF EQUAL (CEL)

(PC) + 2 → PC if (r) = (α)

The left half of the word addressed by the T,N field is compared with the half word addressed by the R field. A skip is made depending on the result.

Op Code	*R Field*	*T,N Field*
31	VPR	CM

COMPARE RIGHT HALFWORD, SKIP IF EQUAL (CER)

(PC) + 2 → PC if (r) = (α)

The right halfword addressed by the T,N field is compared with the halfword addressed by the R field. A skip is made depending on the result.

Op Code	*R Field*	*T,N Field*
33	VPR	CM

COMPARE WORD, SKIP IF NOT EQUAL (CN)

(PC) + 2 → PC if (r) ≠ (α)

The contents of the register indicated by the T,N field is compared with the contents of the VPR indicated by the R field. A skip is made depending on the result.

Op Code	*R Field*	*T,N Field*
34	VPR	CM
DC	VPR	VPR
FC	VPR	CR

COMPARE HALFWORD, SKIP IF NOT EQUAL (CNH)

(PC) + 2 → PC if (r) ≠ (α)

Op Code	*R Field*	*T,N Field*
DD	VPR	VPR
FD	VPR	CR

The halfword indicated by the T,N field is compared with the contents of the VPR word indicated by the R field. A skip is made depending on the result. Indirect addressing is undefined.

COMPARE BYTE, SKIP IF NOT EQUAL (CNB)

(PC) + 2 → PC if (r) ≠ (α)

Op Code	*R Field*	*T,N Field*
DF	VPR	VPR
FF	VPR	CR

The byte indicated by the T,N field is compared with the contents of the VPR byte indicated by the R field. A skip is made depending on the result. Indirect addressing is undefined.

COMPARE LEFT HALFWORD, SKIP IF NOT EQUAL (CNL)

(PC) + 2 → PC if (r) ≠ (α)

Op Code	*R Field*	*T,N Field*
35	VPR	CM

The left half of the word addressed by the T,N field is compared with the halfword addressed by the R field. A skip is made depending on the result.

COMPARE RIGHT HALFWORD, SKIP IF NOT EQUAL (CNR)

(PC) + 2 → PC if (r) ≠ (α)

Op Code	*R Field*	*T,N Field*
37	VPR	CM

The right halfword addressed by the T,N field is compared with the halfword addressed by the R field. A skip is made depending on the result.

Two consecutive parameter words are maintained in CM by the stack instructions to provide status of the last-in, first-out stack of operands.

The first parameter word provides a word count parameter in the 16 most significant bits. The word count indicates the number of operands currently in the stack. The 16 least significant bits of the first parameter word are the space count. The space count indicates the remaining stack capacity. The maximum space allowable for a stack is $2^{15} - 1$.

The second parameter word, at the CM address one greater than the address of the first parameter word, contains the stack pointer. The next available, unused stack location is given by the pointer. The pointer occupies the 24 least significant bits of the parameter word, and the 8 most significant bits are unused.

PUSH STACK (PUSH)

Op Code	*R Field*	*T,N Field*
58	VPR	CM

The first parameter word is read from the CM location specified by the T,N fields. A test for zero is performed on the space count. If the space count is zero, the execution terminates, and the next sequential instruction is taken. If the space count is non-zero, the space count is decremented, the word count is incremented, and the resultant parameter word replaces the original parameter word in CM.

The second parameter word is then read from CM. This word, the stack pointer, is incremented, and the result replaces the original value in CM. The value of the stack pointer before incrementing is the effective address into which the operand is stored. When execution is completed, the next sequential instruction is skipped.

PULL STACK (PULL)

Op Code	*R Field*	*T,N Field*
59	VPR	CM

The first parameter word is read from the CM location specified by the T,N fields. A test for zero is performed on the word count. If the word count is non-zero, the space count is incremented, the word count is decremented, and the resultant parameter word replaces the original parameter word in CM.

The second parameter word is then read from CM. The stack pointer is decremented, and the result replaces the original value in CM. The new value of the stack pointer is the effective address from which the operand is taken. When execution is completed, the next sequential instruction is skipped.

MODIFY STACK (MOD)

Op Code	*R Field*	*T,N Field*
5B	VPR	CM

The amount of the modification is denoted by the contents of the VPR halfword designated by the R field. If the halfword modification value is negative (2's complement), deletion of the most recent stack entries, results. If the halfword modification value is positive, a gap of unused stack locations is created.

The first parameter word is read from the CM location specified by T,N field addressing. The modification value is added to the word count and subtracted from the space count. If either result is negative, the execution terminates, and the next sequential instruction is tak en. If both results are non-negative, the new word and space counts replace the original parameter word in CM.

The second parameter word is then read from CM. The modification value is added to the stack pointer and replaces the original parameter word in CM. When execution is completed, the next sequential instruction is skipped.

LOAD EFFECTIVE ADDRESS (LDEA)

EA → r

Op Code	*R Field*	*T,N Field*
5D	VPR	CM

The effective address of this instruction is developed from the T,N fields as if to reference CM, and is loaded into the VPR specified by the R field.

ANALYZE EFFECTIVE ADDRESS (ANAZ)

EA of object instruction → r

Op Code	*R Field*	*T,N Field*
5F	VPR	CM

The effective address of this instruction, developed from the T,N fields, points to an object instruction in CM. The effective address of the object instruction is d eveloped according to the T,N interpretation normally employed by the object instruction, and the resultant address is loaded into the VPR specified by the R field. If the object instruction is an immediate, then the immediate operand (32 bits) is loaded into the VPR.

The result of development of the effective address of the object instruction is as if the object instruction were in the location of the ANAZ with the following exception. If the object instruction is any program counter relative branch, then the quantity "(PC)" employed for development of β is one greater than it would be if the object instruction were in the location of the ANAZ.

EST POLL BITS (POLL)

If the tested byte = 0,
then (PC) + 1 → PC

and 0 → r.

If the tested byte ≠ 0,
then (PC) + 2 → PC

and Code → r.

Op Code	*R Field*	*T,N Field*
F5	VPR	CR

A code of 0 results if the most significant bit of the tested byte is a "1", and a code of 7 results if only the least significant bit of the tested byte is a "1".

Test the byte specified by the T,N fields for a "1" in any bit position, and skip if any "1's" are present. A code is planted in the halfword of the VPR specified by the R field. The planted code is a binary representation of the bit position of the most significant "1" present in the tested byte.

If no "1's" are present in the tested byte, the next sequential instruction is taken, and the halfword specified by the R field is cleared to zero. Indirect addressing is undefined.

EXECUTE CENTRAL MEMORY (EXEC)

Op Code	*R Field*	*T,N Field*
5C	(not used)	CM

The instruction in the CM location specified by the T,N fields is executed. If the object instruction (the instruction pointed to) is a branch or skip and the condition for branching or skipping is satisfied, the branch or skip will be taken. Note that the instruction pointed to may be located by direct or indirect addressing and may also be an EXEC to continue pointing. When the final object instruction is located, the result of its execution will be as if it were in the location of the original EXEC with the following exception. If the object instruction is any program counter relative branch, then the quantity "(PC)" employed for development of β is one greater than it would be if the object instruction were in the location of the original EXEC.

LOAD VP BASE IN CR FROM VPR (LDMB)

$(\alpha) \rightarrow r$

Op Code	*R Field*	*T,N Field*
F4	(not used)	VPR

The three least significant bytes of the VPR operand specified by the T,N fields is entered into the three least significant bytes of one of the first eight CR's. The most significant byte of the recipient CR remains unchanged. The particular one of eight CR's addressed by this instruction is determined by the identity of the VP executing the instruction. VP_0 loads CR_{00}, VP_1 loads CR_{01}, etc. The R field of the instruction is ignored. This instruction is exempt from the CR protection mechanism.

NO OPERATION (NOP)

Op Code	*R Field*	*T,N Field*
00	(not used)	(not used)

LOAD WORD IMMEDIATE (LDI)

Immediate Operand → r

The immediate operand indicated by the T,N fields is loaded into the register specified by the R field.

Op Code	*R Field*	*T,N Field*
72	VPR	Immediate Operand
62	CR	Immediate Operand

LOAD HALFWORD IMMEDIATE (LDHI)

Immediate Operand → r

The immediate halfword operand indicated by the T,N fields is loaded into the halfword of the register specified by the R field.

Op Code	*R Field*	*T,N Field*
76	VPR	Immediate Operand
66	CR	Immediate Operand

LOAD BYTE IMMEDIATE (LDBI)

Immediate Operand → r

The immediate byte operand indicated by the T,N fields is loaded into the byte of the register indicated by the R field.

Op Code	*R Field*	*T,N Field*
7E	VPR	Immediate Operand
6E	CR	Immediate Operand

LOGICAL OR HALFWORD IMMEDIATE (ORHI)

Immediate Operand OR (r) → r

The immediate halfword specified by T,N field development is combined with the VPR halfword specified by the R field using the OR function. The result replaces the contents of the VPR

Op Code	*R Field*	*T,N Field*
65	VPR	Immediate Operand

LOGICAL OR BYTE IMMEDIATE (ORBI)

Immediate Operand OR (r) → r

The immediate byte specified by T,N field development is combined with the VPR byte specified by the R field using the OR function. The result replaces the contents of the VPR.

Op Code	*R Field*	*T,N Field*
67	VPR	Immediate Operand

LOGICAL AND HALFWORD IMMEDIATE (ANHI)

Immediate Operand · (r) → r

Op Code	*R Field*	*T,N Field*
61	VPR	Immediate Operand

The immediate halfword specified by T,N field development is combined with the VPR halfword specified by the R field using the AND function. The result replaces the contents of the VPR.

LOGICAL AND BYTE IMMEDIATE (ANBI)

Immediate Operand · (r) → r

Op Code	*R Field*	*T,N Field*
63	VPR	Immediate Operand

The immediate byte specified by T,N field development is combined with the VPR byte specified by the R field using the AND function. The result replaces the contents of the VPR.

LOGICAL EXCLUSIVE OR HALFWORD IMMEDIATE (EXHI)

Immediate Operand ⊕ (r) → r

Op Code	*R Field*	*T,N Field*
6D	VPR	Immediate Operand

The immediate halfword specified by T,N field developme nt is combined with the VPR byte specified by the R field using the exclusive OR function. The result replaces the contents of the VPR.

LOGICAL EXCLUSIVE OR BYTE IMMEDIATE (EXBI)

Immediate Operand ⊕ (r) → r

Op Code	*R Field*	*T,N Field*
6F	VPR	Immediate Operand

The immediate byte specified by T,N field development is combined with the VPR byte specified by the R field using the exclusive OR function. The result replaces the contents of the VPR.

LOGICAL EQUIVALENCE HALFWORD IMM.. (EQHI)

Immediate Operand $\odot$ (r) → r

Op Code		
69	VPR	

The immediate halfword specified by T,N field development is combined with the VPR halfword specified by the R field using the equivalence function. The result replaces the contents of the VPR.

LOGICAL EQUIVALENCE BYTE IMMEDIATE (EQBI)

Immediate Operand $\odot$ (r) → r

Op Code	*R Field*	*T,N Field*
6B	VPR	Immediate Operand

The immediate byte specified by T,N field development is combined with the VPR byte specified by the R field using the equivalence function. The result replaces the contents of the VPR.

ADD WORD IMMEDIATE (ADI)

(r) + immediate operand → r

Op Code	*R Field*	*T,N Field*
70	VPR	Immediate Operand

The immediate operand word specified by T,N field development is added to the VPR word specified by the R field. The result replaces the contents of the VPR. Overflows are ignored.

ADD HALFWORD IMMEDIATE (ADHI)

(r) + immediate operand → r

Op Code	*R Field*	*T,N Field*
71	VPR	Immediate Operand

The immediate operand halfword specified by T,N field development is added to the VPR halfword specified by the R field. The result replaces the contents of the VPR. Overflows are ignored.

ADD BYTE IMMEDIATE (ADBI)

(r) + immediate operand → r

The immediate operand byte specified by T,N field development is added to the VPR byte specified by the R field. The result replaces the contents of the VPR. Overflows are ignored.

Op Code	*R Field*	*T,N Field*
73	VPR	Immediate Operand

SUBTRACT WORD IMMEDIATE (SUI)

(r) - immediate operand → r

The immediate operand word specified by T,N field development is subtracted from the VPR word specified by the R field. The result replaces the contents of the VPR. Overflows are ignored.

Op Code	*R Field*	*T,N Field*
74	VPR	Immediate Operand

SUBTRACT HALFWORD IMMEDIATE (SUHI)

(r) - immediate operand → r

The immediate operand halfword specified by T,N field development is subtracted from the VPR halfword specified by the R field. The result replaces the contents of the VPR. Overflows are ignored.

Op Code	*R Field*	*T,N Field*
75	VPR	Immediate Operand

SUBTRACT BYTE IMMEDIATE (SUBI)

(r) - immediate operand → r

The immediate operand byte specified by T,N field development is subtracted from the VPR byte specified by the R field. The result replaces the contents of the VPR. Overflows are ignored.

Op Code	*R Field*	*T,N Field*
77	VPR	Immediate Operand

COMPARE WORD IMMEDIATE, SKIP IF EQUAL (CEI)

(PC) + 2 → PC if immediate operand = (r)

The immediate operand word specified by the T,N field development is compared with the operand word addressed by the R field. A skip is made depending on the result.

Op Code	*R Field*	*T,N Field*
78	VPR	Immediate Operand

COMPARE HALFWORD IMMEDIATE, SKIP IF EQUAL (CEHI)

(PC) + 2 → PC if immediate operand - (r)

The immediate operand halfword specified by the T,N field development is compared with the operand halfword addressed by the R field. A skip is made depending on the result.

Op Code	*R Field*	*T,N Field*
79	VPR	Immediate Operand

COMPARE BYTE IMMEDIATE, SKIP IF EQUAL (CEBI)

(PC) + 2 → PC if immediate operand ≐ (r)

The immediate operand byte specified by the T,N field development is compared with the operand byte addressed by the R field. A skip is made depending on the result.

Op Code	*R Field*	*T,N Field*
7B	VPR	Immediate Operand

COMPARE WORD IMMEDIATE, SKIP IF NOT EQUAL (CNI)

(PC) + 2 → PC if immediate operand ≠ (r)

The immediate operand word specified by the T,N field development is compared with the operand word addressed by the R field. A skip is made depending on the result.

Op Code	*F Field*	*T,N Field*
7C	VPR	Immediate Operand

COMPARE HALFWORD IMMEDIATE, SKIP IF NOT EQUAL (CNHI)

(PC) + 2 → PC if immediate operand ≠ (r)

Op Code	*R Field*	*T,N Field*
7D	VPR	Immediate Operand

The immediate operand halfword specified by the T,N field development is compared with the operand halfword addressed by the R field. A skip is made depending on the result.

COMPARE BYTE IMMEDIATE,SKIP IF NOT EQUAL (CNBI)

(PC) + 2 → PC if immediate operand ≠ (r)

Op Code	*R Field*	*T,N Field*
7F	VPR	Immediate Operand

The immediate operand byte specified by the T,N field development is compared with the operand byte addressed by the R field. A skip is made depending on the result.

SET LEFT HALF (SL)

R OR (r) → r

"1's" are set in those bit positions marked by "1's" in the R field, in the left half of the CR byte operand specified by the address in the T,N fields. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
FA	Mask	CR

SET RIGHT HALF (SR)

R OR (r) → r

"1's" are set in those bit positions marked by "1's" in the R field, in the right half of the CR byte operand specified by the address in the T,N fields. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
FE	Mask	CR

RESET LEFT HALF (RL)

$\bar{R} \cdot (r) \rightarrow r$

"0's" are set in those bit positions marked by "1's" in the R field, in the left half of the CR byte operand specified by the address in the T,N fields. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
F2	Mask	CR

RESET RIGHT HALF (RR)

$\bar{R} \cdot (r) \rightarrow r$

"0's" are set in those bit positions marked by "1's" in the R field, in the right half of the CR byte operand specified by the address in the T,N fields. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
F6	Mask	CR

TEST FOR ANY 1 (TOL)

$$(PC) + 2 \rightarrow \text{if } \Sigma \left[R_i \cdot (r)_i \right] = 1$$

The left half of the byte operand specified by the T,N fields is tested for a "1" in any bit position(s) marked by "1's" in the R field. The next sequential instruction is skipped if the test is satisfied. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
CA	Mask	CR

TEST FOR ANY 1 (TOR)

$$(PC) + 2 \rightarrow PC \text{ if } \Sigma \left[R_i \cdot (r)_i \right] = 1$$

The right half of the byte operand specified by the T,N fields is tested for a "1" in any bit position(s) marked by "1's" in the R field. The next sequential instruction is skipped if the test is satisfied. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
CE	Mask	CR

TEST FOR ANY 0 (TZL)

$$(PC) + 2 \rightarrow PC \text{ if } \Pi \left[R_i \cdot (r)_i \text{ OR } \overline{R}_i \right] = 0$$

The left half of the byte operand specified by the T,N fields is tested for a "0" in any bit position(s) marked by "1's" in the R field. The next sequential instruction is skipped if the test is satisfied. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
C2	Mask	CR

TEST FOR ANY 0 (TZR)

$$(PC) + 2 \rightarrow PC \text{ if } \Pi \left[R_i \cdot (r)_i \text{ OR } \overline{R}_i \right] = 0$$

The right half of the byte operand specified by the T,N fields is tested for a "0" in any bit position(s) marked by "1's" in the R field. The next sequential instruction is skipped if the test is satisfied. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
C6	Mask	CR

TEST FOR ALL 1 (TAOL)

$$(PC) + 2 \rightarrow PC \text{ if } \Pi \left[R_i \cdot (r) \text{ OR } \overline{R_i} \right] = 1$$

Op Code	*R Field*	*T,N Field*
EA	Mask	CR

The left half of the byte operand specified by the T,N fields is tested for all "1's" in bit positions marked by "1's" in the R field. The next sequential instruction is skipped if the test is satisfied. Indirect addressing is undefined.

TEST FOR ALL 1 (TAOR)

$$(PC) + 2 \rightarrow PC \text{ if } \Pi \left[R_i - (r) \text{ OR } \overline{R_i} \right] = 1$$

Op Code	*R Field*	*T,N Field*
EE	Mask	CR

The right half of the byte operand specified by the T,N fields is tested for all "1's" in bit positions marked by "1's" in the R field. The next sequential instruction is skipped if the test is satisfied. Indirect addressing is undefined.

TEST FOR ALL 0 (TAZL)

$$(PC) + 2 \rightarrow PC \text{ if } \Sigma \left[R_i \cdot (r)_i \right] = 0$$

Op Code	*R Field*	*T,N Field*
E2	Mask	CR

The left half of the byte operand specified by the T,N fields is tested for all "0's" in bit positions marked by "1's" in the R field. The next sequential instruction is skipped if the test is satisfied. Indirect addressing is undefined.

TEST FOR ALL 0 (TAZR)

$$(PC) + 2 \rightarrow PC \text{ if } \Sigma \left[R_i \cdot (r)_i \right] = 0$$

Op Code	*R Field*	*T,N Field*
E6	Mask	CR

The right half of the byte operand specified by the T,N fields is tested for all "0's" in bit positions marked by "1's" in the R field. The next sequential instructi on is skipped if the test is satisfied. Indirect addressing is undefined.

SHIFT LOGICAL (SHL)

Op Code	*R Field*	*T,N Field*
64	VPR	Immediate Operand

The contents of the VP specified by the R field are shifted the number of bit positions specified by the immediate operand. The six LSB's of the immediate operand are treated as a signed number with negative values represented in 2's complement form. Positive values result in left shifts and negative values result in right shifts. Shift range: +31 to -32. "0's" are shifted into the vacated portion of the VPR. Indirect addressing is undefined.

SHIFT ARITHMETIC (SHA)

Op Code	*R Field*	*T,N Field*
60	VPR	Immediate Operand

The contents of the VPR specified by the R field are shifted the number of bit positions specified by the immediate operand. The six LSB's of the immediate operand are treated as a signed number with negative values represented in 2's complement form. Positive values result in left shifts and negative values result in right shifts. Shift range: +31 to -32. Left shifts are identical to left logical shifts. For right shifts, the most significant bit position remains unchanged and is shifted into the vacated portions of the VPR. Indirect addressing is undefined.

SHIFT CYCLIC (SHC)

Op Code	*R Field*	*T,N Field*
6C	VPR	Immediate Operand

The contents of the VPR specified by the R field are shifted the number of bit positions specified by the immediate operand. The six LSB's of the immediate operand are treated as a signed number with the negative values represented in 2's complement form. Positive values result in left shifts and negative values result in right shifts. Shift range: +31 to -32. With both right and left shifts, bits are shifted into one end of the VPR as they exit from the other end. Indirect addressing is undefined.

TEST FOR ANY 0, SET AND SKIP (TSZL)

$R \text{ OR } (r) \rightarrow r$

$(PC) + 2 \rightarrow PC \text{ if } \pi \left[R_i \cdot (r) \text{ OR } \overline{R}_i \right] = 0$

Test the left half of the byte operand specified by the T,N fields for a "0" in any position(s) marked by "1's" in the R field; skip the next sequential instruction if the test is satisfied. Independent of the test result, set "1's" into those positions marked by the R field. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
D2	Mask	CR

TEST FOR ANY 1, SET AND SKIP (TSOL)

$R \text{ OR } (r) \rightarrow r$

$(PC) + 2 \rightarrow PC \text{ if } \Sigma \left[R_i \cdot (r)_i \right] = 1$

Test the left half of the byte operand specified by the T,N fields for a "1" in any position(s) marked by "1's" in the R field; skip the next sequential instruction if the test is satisfied. Independent of the test result, set "1's" into those positions marked by the R field. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
DA	Mask	CR

TEST FOR ANY 0, RESET AND SKIP (TRZL)

$\overline{R} \cdot (r) \rightarrow r$

$(PC) + 2 \rightarrow PC \text{ if } \pi \left[R_i \cdot (r)_i \text{ OR } \overline{R}_i \right] = 0$

Test the left half of the byte operand specified by the T,N fields for a "0" in any position(s) marked by "1's" in the R field; skip the next sequential instruction if the test is satisfied. Independent of the test result, set "0's" into those positions marked by the R field. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
92	Mask	CR

TEST FOR ANY 1, RESET AND SKIP (TROL)

$\overline{R} \cdot (r) \rightarrow r$

$(PC) + 2 \rightarrow PC$ if $\Sigma \left[R_i \cdot (r)_i \right] = 1$

Test the left half of the byte operand specified by the T,N fields for a "1" in any position(s) marked by "1's" in the R field; skip the next sequential instruction if the test is satisfied. Independent of the test result, set "0's" into those positions marked by the R field. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
9A	Mask	CR

TEST FOR ANY 0, SET AND SKIP (TSZR)

R OR $(r) \rightarrow r$

$(PC) + 2 \rightarrow PC$ if $\pi \left[R_i \cdot (r)_i \text{ OR } \overline{R}_i \right] = 0$

Test the right half of the byte operand specified by the T,N fields for a "0" in any position(s) marked by "1's" in the R field; skip the next sequential instruction if the test is satisfied. Independent of the test result, set "1's" into those positions marked by the R field. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
D6	Mask	CR

TEST FOR ANY 1, SET AND SKIP (TSOR)

R OR $(r) \rightarrow r$

$(PC) + 2 \rightarrow PC$ if $\Sigma \left[R_i \cdot (r)_i \right] = 1$

Test the right half of the byte operand specified by the T,N fields for a "1" in any position(s) marked by "1's" in the R field; skip the next sequential instruction if the test is satisfied. Independent of the test result, set "1's" into those positions marked by the R field. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
DE	Mask	CR

TEST FOR ANY 0, RESET AND SKIP (TRZR)

$\overline{R} \cdot (r) \rightarrow r$

$(PC) + 2 \rightarrow PC$ if $\pi \left[R_i \cdot (r)_i \text{ OR } R_i \right] = 0$

Test the right half of the byte operand specified by the T,N fields for a "0" in any position(s) marked by "1's" in the R field; skip the next sequential instruction if the test is satisfied. Independent of the test result, set "0's" into those positions marked by the R field. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
96	Mask	CR_r

TEST FOR ANY 1, RESET AND SKIP (TROR)

$\overline{R} \cdot (r) \rightarrow r$

$(PC) + 2 \rightarrow PC$ if $\Sigma \left[R_i \cdot (r)_i \right] = 1$

Test the right half of the byte operand specified by the T,N fields for a "1" in any position(s) marked by "1's" in the R field; skip the next sequential instruction if the test is satisfied. Independent of the test result, set "0's" into those positions marked by the R field. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
9E	Mask	CR

TEST WHOLE FOR ZERO, BRANCH (TZ)

$\beta \rightarrow PC$ if $(r) = 0$

Test the contents of the whole word specified by the R field and branch to the memory location specified by the T,N fields if the test condition is satisfied.

Op Code	*R Field*	*T,N Field*
B0	VPR	β, PC Rel
A0	CR	β, PC Rel

TEST HALF FOR = ZERO, BRANCH (TZH)

$\beta \rightarrow PC$ if $(r) = 0$

Test the contents of the half word specified by the R field and branch to the memory location specified by the T,N fields if the test condition is satisfied.

Op Code	*R Field*	*T,N Field*
B1	VPR	β, PC Rel
A1	CR	β, PC Rel

TEST BYTE FOR = ZERO, BRANCH (TZB)

$\beta \rightarrow PC$ if $(r) = 0$

Test the contents of the byte specified by the R field and branch to the memory location specified by the T,N fields if the test condition is satisfied.

Op Code	*R Field*	*T,N Field*
B3	VPR	β, PC Rel
A3	CR	β, PC Rel

TEST WHOLE FOR ≠ ZERO, BRANCH (TN)

$\beta \rightarrow PC$ if $(r) \neq 0$

Test the contents of the whole word specified by the R field and branch to the memory location specified by the T,N fields if the test condition is satisfied.

Op Code	*R Field*	*T,N Field*
B4	VPR	β, PC Rel
A4	CR	β, PC Rel

TEST HALF FOR ≠ ZERO, BRANCH (TNH)

$\beta \rightarrow PC$ if $(r) \neq 0$

Test the contents of the halfword specified by the R field and branch to the memory location specified by the T,N fields if the test condition is satisfied.

Op Code	*R Field*	*T,N Field*
B5	VPR	β, PC Rel
A5	CR	β, PC Rel

TEST BYTE FOR ≠ ZERO, BRANCH (TNB)

$\beta \rightarrow PC$ if $(r) \neq 0$

Test the contents of the byte specified by the R field and branch to the memory location specified by the T,N fields if the test condition is satisfied.

Op Code	*R Field*	*T,N Field*
B7	VPR	β, PC Rel
A7	CR	β, PC Rel

TEST WHOLE FOR ≥ ZERO, BRANCH (TP)

$\beta \rightarrow PC$ if $(r) \geq 0$

Test the contents of the whole word specified by the R field and branch to the memory location specified by the T,N fields if the test condition is satisfied.

Op Code	*R Field*	*T,N Field*
B8	VPR	β, PC Rel
A8	CR	β, PC Rel

TEST HALF FOR $\geq$ ZERO, BRANCH (TPH)

$\beta \rightarrow PC$ if $(r) \geq 0$

Test the contents of the halfword specified by the R field and branch to the memory location specified by the T,N fields if the test condition is satisfied.

Op Code	*R Field*	*T,N Field*
B9	VPR	β, PC Rel
A9	CR	β, PC Rel

TEST BYTE FOR $\geq$ ZERO, BRANCH (TPB)

$\beta \rightarrow PC$ if $(r) \geq 0$

Test the contents of the byte specified by the R field and branch to the memory location specified by the T,N fields if the test condition is satisfied.

Op Code	*R Field*	*T,N Field*
BB	VPR	β, PC Rel
AB	CR	β, PC Rel

TEST WHOLE FOR < ZERO, BRANCH (TM)

$\beta \rightarrow PC$ if $(r) < 0$

Test the contents of the whole word specified by the R field and branch to the memory location specified by the T,N fields if the test condition is satisfied.

Op Code	*R Field*	*T,N Field*
BC	VPR	β, PC Rel
AC	CR	β, PC Rel

TEST HALF FOR < ZERO, BRANCH (TMH)

$\beta \rightarrow PC$ if $(r) < 0$

Test the contents of the halfword specified by the R field and branch to the memory location specified by the T,N fields if the test condition is satisfied.

Op Code	*R Field*	*T,N Field*
BD	VPR	β, PC Rel
AD	CR	β, PC Rel

TEST BYTE FOR < ZERO, BRANCH (TMB)

$\beta \rightarrow PC$ if (r) < 0

Test the contents of the byte specified by the R field and branch to the memory location specified by the T,N fields if the test condition is satisfied.

Op Code	*R Field*	*T,N Field*
BF	VPR	β, PC Rel
AF	CR	β, PC Rel

INCREMENT VPR BY 1, BRANCH IF RESULT = 0 (IBZ)

$(r) + 1 \rightarrow r$, $\beta \rightarrow PC$ if $(r) + 1 = 0$

The VPR halfword specified by the R field is incremented by 1 and the result replaces the contents of the VPR halfword. If the result is zero, a branch is taken to the location specified by the T,N fields.

Op Code	*R Field*	*T,N Field*
B2	VPR	β, PC Rel

INCREMENT VPR BY 1, BRANCH IF RESULT ≠ 0 (IBN)

$(r) + 1 \rightarrow r$, $\beta \rightarrow PC$ if $(r) + 1 \neq 0$

The VPR halfword specified by the R field is incremented by 1 and the result replaces the contents of the VPR halfword. If the result is non zero, a branch is taken to the location specified by the T,N fields.

Op Code	*R Field*	*T,N Field*
B6	VPR	β, PC Rel

DECREMENT VPR BY 1, BRANCH IF RESULT = 0 (DBZ)

$(r) - 1 \rightarrow r$, $\beta \rightarrow PC$ if $(r) - 1 = 0$

The VPR halfword specified by the R field is decremented by 1 and the result replaces the contents of the VPR halfword. If the result is zero, a branch is taken to the location specified by the T,N fields.

Op Code	*R Field*	*T,N Field*
BA	VPR	β, PC Rel

DECREMENT VPR BY 1, BRANCH IF RESULT ≠ 0 (DBN)

$(r) - 1 \rightarrow r$, $\beta \rightarrow PC$ if $(r) - 1 \neq 0$

The VPR halfword specified by the R field is decremented by 1 and the result replaces the contents of the VPR halfword. If the result is non zero, a branch is taken to the location specified by the T,N fields.

Op Code	*R Field*	*T,N Field*
BE	VPR	β, PC Rel

BRANCH, PC RELATIVE SAVE PC (BPCS)

(PC) + 1 → r, β → PC

The (PC) + 1 is saved in the VPR specified by the R field. The most significant bit of the VPR is loaded to indicate from which memory the instruction string is currently being accessed. A "1" indicates CM, and a "0" indicates ROM. β replaces the contents of the PC.

Op Code	*R Field*	*T,N Field*
AE	VPR	β, PC Rel

BRANCH TO CM, BASE RELATIVE SAVE PC (BCS)

(PC) + 1 → r, β → PC

The (PC) + 1 is saved in the VPR specified by the R field. The most significant bit of the VPR is loaded to indicate from which memory the instruction string is currently being accessed. A "1" indicates CM, and a "0" indicates ROM. β replaces the contents of the PC.

Op Code	*R Field*	*T,N Field*
12	VPR	β, Base Rel

BRANCH TO ROM, SAVE PC (BRS)

(PC) + 1→ r, β → PC

The (PC) + 1 is saved in the VPR specified by the R field. The most significant bit of the VPR is loaded to indicate from which memory the instruction string is currently being accessed. A "1" indicates CM, and a "0" indicates ROM. β replaces the contents of the PC.

Op Code	*R Field*	*T,N Field*
46	VPR	β, ROM

BRANCH TO CM, ABSOLUTE, SAVE PC (BCAS)

(PC) + 1 → r, β → PC

The (PC) + 1 is saved in the VPR specified by the R field. The most significant bit of the VPR is loaded to indicate from which memory the instruction string is currently being accessed. A "1" indicates CM, and a "0" indicates ROM. β replaces the contents of the PC.

Op Code	*R Field*	*T,N Field*
52	VPR	β, Absolute

BRANCH, PC RELATIVE (BPC)

β → PC

β replaces the contents of the PC.

Op Code	*R Field*	*T,N Field*
5A	(not used)	β, PC Rel
5E	(not used)	β, PC Rel

BRANCH TO ROM (BR)

β → PC

β replaces the contents of the PC.

Op Code	*R Field*	*T,N Field*
42	(not used)	β, ROM

BRANCH TO CM, BASE RELATIVE (BC)

β → PC

β replaces the contents of the PC.

Op Code	*R Field*	*T,N Field*
02	(not used)	β, Base Rel
32	(not used)	β, Base Rel

BRANCH TO CM, ABSOLUTE (BCA)

β → PC

β replaces the contents of the PC.

Op Code	*R Field*	*T,N Field*
4A	(not used)	β, Absolute
4E	(not used)	β, Absolute

BRANCH TO ROM, STORE PC (BRSM)

(PC) + 1 → fixed CM location

β → PC

Op Code	*R Field*	*T,N Field*
56	(not used)	β, ROM

The (PC) + 1 is stored into one of eight CM locations depending on the identity of the VP executing the instruction. The most significant bit of the CM location is modified to indicate from which memory the instruction string is currenily being accessed. A "1" indicates CM, and a "0" indicates ROM. The eight CM locations are contiguous and begin at 20_{16}. β replaces the contents of the PC. Indirect addressing is undefined.

SET VP FLAG (VPS)

A flag bit is set in the CR byte specified by the T,N fields. The position of the bit within the byte is determined by the number of the VP executing the instruction. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
86	(not used)	CR

RESET VP FLAG (VPR)

A flag bit is reset in the CR byte specified by the T,N fields. The position of the bit within the byte is determined by the number of the VP executing the instruction. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
82	(not used)	CR

TEST CR FOR 1 AND SKIP IF = 1 (VPTO)

(PC) + 2 → PC, if flag = 1

A flag bit in the CR byte specified by the T,N fields is tested for "1". The position of the bit within the byte is determined by the number of the VP executing the instruction. A skip is taken if the test condition is satisfied. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
8E	(not used)	CR

TEST CR FOR 0 AND SKIP IF = 0 (VPTZ)

(PC) + 2 → PC, if flag = 0

A flag bit in the CR byte specified by the T,N fields is tested for "0". The position of the bit within the byte is determined by the number of the VP executing the instruction. A skip is taken if the test condition is satisfied. Indirect addressing is undefined.

Op Code	*R Field*	*T,N Field*
8A	(not used)	CR

SEQUENTIAL INDEX OF INSTRUCTIONS

SEQUENTIAL INDEX (CONTINUED)

SEQUENTIAL INDEX (CONTINUED)

MNEM CODE	INSTRUCTION	OP CODE	PAGE NO.
CE	COMPARE W CR TO VPR, SIE	F8	23
CEH	COMPARE HW VPR TO VPR, SIE	D9	23
CEH	COMPARE HW CR TO VPR, SIE	F9	23
CEB	COMPARE B VPR TO VPR, SIE	DB	23
CEB	COMPARE B CR TO VPR, SIE	FB	23
CEL	COMPARE LH CM TO VPR, SIE	31	23
CER	COMPARE RH CM TO VPR, SIE	33	23
CN	COMPARE W CM TO VPR, SINE	34	23
CN	COMPARE W VPR TO VPR, SINE	DC	23
CN	COMPARE W CR TO VPR, SINE	FC	23
CNH	COMPARE HW VPR TO VPR, SINE	DD	24
CNH	COMPARE HW CR TO VPR, SINE	FD	24
CNB	COMPARE B VPR TO VPR, SINE	DF	24
CNB	COMPARE B CR TO VPR, SINE	FF	24
CNL	COMPARE LH CM TO VPR, SINE	35	24
CNR	COMPARE RH CM TO VPR, SINE	37	24
PUSH	PUSH INTO STACK	58	25
PULL	PULL FROM STACK	59	26
MOD	MOD STACK	5B	26
LDEA	LOAD EFFECTIVE ADDRESS	5D	27
ANAZ	ANALYZE CM	5F	27
POLL	POLL CR & SET VPR	F5	28
EXEC	EXECUTE CM	5C	28
LDMB	LOAD VP BASE FROM VPR TO CR	F4	29
NOP	NO OPERATION	00	29
LDI	LOAD IMMED W INTO VPR	72	30
LDI	LOAD IMMED W INTO CR	62	30
LDHI	LOAD IMMED HW INTO VPR	76	30
LDHI	LOAD IMMED HW INTO CR	66	30
LDBI	LOAD IMMED B INTO VPR	7E	30
LDBI	LOAD IMMED B INTO CR	6E	30
CRHI	OR LOGICAL IMMED HW TO VPR	65	30
CRBI	OR LOGICAL IMMED B TO VPR	67	30
ANHI	AND LOGICAL IMMED HW TO VPR	61	31
ANBI	AND LOGICAL IMMED B TO VPR	63	31
EXHI	EXCLUSIVE OR IMMED HW TO VPR	6D	31
EXBI	EXCLUSIVE OR IMMED B TO VPR	6F	31
EQHI	LOGICAL EQUIVALENCE IMMED HW TO VPR	69	32
EQBI	LOGICAL EQUIVALENCE IMMED B TO VPR	6B	32
ADI	ADD IMMED W TO VPR	70	32
ADHI	ADD IMMED HW TO VPR	71	32
ADBI	ADD IMMED B TO VPR	73	33
SUI	SUBT IMMED W FROM VPR	74	33
SUHI	SUBT IMMED HW FROM VPR	75	33
SUBI	SUBT IMMED B FROM VPR	77	33
CEI	COMPARE IMMED W WITH VPR, SIE	78	34
CEHI	COMPARE IMMED HW WITH VPR, SIE	79	34
CEBI	COMPARE IMMED B WITH VPR, SIE	7B	34
CNI	COMPARE IMMED W WITH VPR, SINE	7C	34
CNHI	COMPARE IMMED HW WITH VPR, SINE	7D	35
CNBI	COMPARE IMMED B WITH VPR, SINE	7F	35
SL	SET BITS IN CR, LH	FA	36
SR	SET BITS IN CR, RH	FE	36

MNEM CODE	INSTRUCTION	OP CODE	PAGE NO.
RL	RESET BITS IN CR, LH	F2	36
RR	RESET BITS IN CR, RH	F6	36
TOL	TEST UNDER MASK IN CR FOR ANY 1 LH & SKIP	CA	37
TOR	TEST UNDER MASK IN CR FOR ANY 1 RH & SKIP	CE	37
TZL	TEST UNDER MASK IN CR FOR ANY 0 LH & SKIP	C2	37
TZR	TEST UNDER MASK IN CR FOR ANY 0 RH & SKIP	C6	37
TAOL	TEST UNDER MASK IN CR FOR ALL 1 LH & SKIP	EA	38
TAOR	TEST UNDER MASK IN CR FOR ALL 1 RH & SKIP	EE	38
TAZL	TEST UNDER MASK IN CR FOR ALL 0 LH & SKIP	E2	38
TAZR	TEST UNDER MASK IN CR FOR ALL 0 RH & SKIP	E6	38
SHL	SHIFT LOGICAL W	64	39
SHA	SHIFT ARITH W	60	39
SHC	SHIFT CYCLIC W	6C	39
TSZL	TEST FOR ANY 0 IN CR LH SET & SKIP	D2	40
TSOL	TEST FOR ANY 1 IN CR LH SET	DA	40
TRZL	TEST FOR ANY 0 IN CR LH RESET	92	40
TROL	TEST FOR ANY 1 IN CR LH RESET	9A	41
TSZR	TEST FOR ANY 0 IN CR RH SET	D6	41
TSOR	TEST FOR ANY 1 IN CR RH SET	DE	41
TRZR	TEST FOR ANY 0 IN CR RH RESET	96	42
TROR	TEST FOR ANY 1 IN CR RH RESET	9E	42
TZ	TEST W ARITH, BRANCH IF VPR EQ 0	B0	43
TZ	TEST W ARITH, BRANCH IF CR EQ 0	A0	43
TZH	TEST HW ARITH, BRANCH IF VPR EQ 0	B1	43
TZH	TEST HW ARITH, BRANCH IF CR EQ 0	A1	43
TZB	TEST B ARITH, BRANCH IF VPR EQ 0	B3	43
TZB	TEST B ARITH, BRANCH IF CR EQ 0	A3	43
TN	TEST W ARITH, BRANCH IF VPR NEQ 0	B4	44
TN	TEST W ARITH, BRANCH IF CR NEQ 0	A4	44
TNH	TEST HW ARITH, BRANCH IF VPR NEQ 0	B5	44
TNH	TEST HW ARITH, BRANCH IF CR NEQ 0	A5	44
TNB	TEST B ARITH, BRANCH IF VPR NEQ 0	B7	44
TNB	TEST B ARITH, BRANCH IF CR NEQ 0	A7	44
TP	TEST W ARITH, BRANCH IF VPR GR OR EQ 0	B8	44
TP	TEST W ARITH, BRANCH IF CR GR OR EQ 0	A8	44
TPH	TEST HW ARITH, BRANCH IF VPR GR OR EQ 0	B9	45
TPH	TEST HW ARITH, BRANCH IF CR GR OR EQ 0	A9	45
TPB	TEST B ARITH, BRANCH IF VPR GR OR EQ 0	BB	45
TPB	TEST B ARITH, BRANCH IF CR GR OR EQ 0	AB	45
TM	TEST W ARITH, BRANCH IF VPR LS 0	BC	45
TM	TEST W ARITH, BRANCH IF CR LS 0	AC	45
TMH	TEST HW ARITH, BRANCH IF VPR LS 0	BD	45
TMH	TEST HW ARITH, BRANCH IF CR LS 0	AD	45
TMB	TEST B ARITH, BRANCH IF VPR LS 0	BF	46
TMB	TEST B ARITH, BRANCH IF CR LS 0	AF	46
IBZ	INCR VPR BY 1, BRANCH IF RESULT EQ 0	B2	47
IBN	INCR VPR BY 1, BRANCH IF RESULT NEQ 0	B6	47
DBZ	DECR VPR BY 1, BRANCH IF RESULT EQ 0	BA	47
DBN	DECR VPR BY 1, BRANCH IF RESULT NEQ 0	BE	47
BPCS	BRANCH UNCONDIT TO REL PC, SAVE PC IN VPR	AE	48
BCS	BRANCH UNCONDIT TO CM REL BASE, SAVE PC IN VPR	12	48
BRS	BRANCH UNCONDIT TO RCM, SAVE PC IN VPR	46	48
BCAS	BRANCH UNCONDIT TO CM ABSOL, SAVE PC IN VPR	52	48

SEQUENTIAL INDEX (CONTINUED)

MNEMONIC INDEX OF INSTRUCTIONS

MNEM CODE	INSTRUCTION	OP CODE	PAGE NO.
AD	ADD W IN CM TO VPR	50	15
AD	ADD W IN VPR TO VPR	D0	15
ADB	ADD B IN VPR TO VPR	D3	15
ADBI	ADD IMMED B TO VPR	73	33
ADH	ADD HW IN VPR TO VPR	D1	15
ADHI	ADD IMMED HW TO VPR	71	32
ADI	ADD IMMED W TO VPR	70	32
ADL	ADD LH IN CM TO VPR	51	15
ADR	ADD RH IN CM TO VPR	53	16
AN	AND LOGICAL W IN CM TO VPR	40	19
AN	AND LOGICAL W IN VPR TO VPR	C0	19
AN	AND LOGICAL W IN CR TO VPR	E0	19
ANAZ	ANALYZE CM	5F	27
ANB	AND LOGICAL B IN VPR TO VPR	C3	19
ANB	AND LOGICAL B IN CR TO VPR	E3	19
ANBI	AND LOGICAL IMMED B TO VPR	63	31
ANH	AND LOGICAL HW IN VPR TO VPR	C1	19
ANH	AND LOGICAL HW IN CR TO VPR	E1	19
ANHI	AND LOGICAL IMMED HW TO VPR	61	31
ANL	AND LOGICAL LH IN CM TO VPR	41	19
ANR	AND LOGICAL RH IN CM TO VPR	43	19
BC	BRANCH UNCONDIT TO REL CM WITH BASE AUG	32	49
BC	BRANCH UNCONDIT TO CM WITH REL BASE	02	49
BCA	BRANCH UNCONDIT TO AUG ABSOL CM	4E	49
BCA	BRANCH UNCONDIT TO ABSOL CM	4A	49
BCAS	BRANCH UNCONDIT TO CM ABSOL, SAVE PC IN VPR	52	48
BCS	BRANCH UNCONDIT TO CM REL BASE, SAVE PC IN VPR	12	48
BPC	BRANCH UNCONDIT TO REL PC	5A	49
BPC	BRANCH UNCONDIT TO AUG REL PC	5E	49
BPCS	BRANCH UNCONDIT TO REL PC, SAVE PC IN VPR	AE	48
BR	BRANCH UNCONDIT TO ROM	42	49
BRS	BRANCH UNCONDIT TO ROM, SAVE PC IN VPR	46	48
BRSM	BRANCH UNCONDIT TO ABSOL ROM, SAVE PC IN CM LOC ID, AUG	56	50
CE	COMPARE W CM TO VPR, SIE	30	23
CE	COMPARE W VPR TO VPR, SIE	D8	23
CE	COMPARE W CR TO VPR, SIE	F8	23
CEB	COMPARE B VPR TO VPR, SIE	DB	23
CEB	COMPARE B CR TO VPR, SIE	FB	23
CEBI	COMPARE IMMED B WITH VPR, SIE	7B	34
CEH	COMPARE HW VPR TO VPR, SIE	D9	23
CEH	COMPARE HW CR TO VPR, SIE	F9	23
CEHI	COMPARE IMMED HW WITH VPR, SIE	79	34
CEI	COMPARE IMMED W WITH VPR, SIE	78	34
CEL	COMPARE LH CM TO VPR, SIE	31	23
CER	COMPARE RH CM TO VPR, SIE	33	23
CN	COMPARE W CM TO VPR, SINE	34	23
CN	COMPARE W VPR TO VPR, SINE	DC	23
CN	COMPARE W CR TO VPR, SINE	FC	23
CNB	COMPARE B VPR TO VPR, SINE	DF	24
CNB	COMPARE B CR TO VPR, SINE	FF	24
CNBI	COMPARE IMMED B WITH VPR, SINE	7F	35
CNH	COMPARE HW VPR TO VPR, SINE	DD	24

MNEMONIC INDEX (CONTINUED)

MNEMONIC INDEX (CONTINUED)

MNEMONIC INDEX (CONTINUED)

OP CODE INDEX OF INSTRUCTIONS

	OP CODE	INSTRUCTION	MNEM CODE	PAGE NO.
	00	NO OPERATION	NOP	29
	02	BRANCH UNCONDIT TO CM WITH REL BASE	BC	49
	04	LOAD WD TO VPR FROM CM	LD	13
	05	LOAD LH FROM CM TO VPR	LDL	13
	06	LOAD WD TO VPR FROM CM ABS	LDA	13
	07	LOAD RH FROM CM TO VPR	LDR	14
	08	LOAD AUG WD TO CR FROM CM	LD	13
	09	LOAD AUG LH FROM CM TO CR	LDL	13
	0A	LOAD FILE FROM CM AUG INTO VPR	LDF	14
	0B	LOAD AUG RH FROM CM TO CR	LDR	14
	0C	LOAD AUG WD TO VPR FROM CM	LD	13
	0D	LOAD AUG LH FROM CM TO VPR	LDL	13
	0E	LOAD AUG WD TO VPR FROM CM	LDA	13
	0F	LOAD AUG RH FROM CM TO VPR	LDR	14
	10	STORE WD FROM CR TO CM	ST	11
	11	STORE CM LH FROM CR	STL	11
	12	BRANCH UNCONDIT TO CM REL BASE, SAVE PC IN VPR	BCS	48
	13	STORE CM RH FROM CR	STR	11
	14	STORE WD FROM VPR TO CM	ST	11
	15	STORE LH FROM VPR TO CM	STL	11
	16	STORE WD FROM VPR TO CM ABS	STA	11
	17	STORE CM RH FROM VPR	STR	11
	18	STORE AUG WD FROM CR TO CM	ST	11
	19	STORE AUG CM LH FROM CR	STL	11
	1A	STORE FILE FROM VPR INTO AUG CM	STF	12
	1B	STORE AUG CM RH FROM CR	STR	11
	1C	STORE AUG WD FROM VPR TO CM	ST	11
	1D	STORE AUG LH OF CM FROM VPR(L OR R HALF)	STL	11
	1E	STORE AUG WD FROM VPR TO CM ABS	STA	11
	1F	STORE AUG CM RH FROM VPR	STR	11
*	2A	STORE FILE FROM VPR INTO CM	STF	12
	30	COMPARE W CM TO VPR, SIE	CE	23
	31	COMPARE LH CM TO VPR, SIE	CEL	23
	32	BRANCH UNCONDIT TO REL CM WITH BASE AUG	BC	49
	33	COMPARE RH CM TO VPR, SIE	CER	23
	34	COMPARE W CM TO VPR, SINE	CN	23
	35	COMPARE LH CM TO VPR, SINE	CNL	24
	37	COMPARE RH CM TO VPR, SINE	CNR	24
	38	LOAD WD FROM CM TO CR	LD	13
	39	LOAD LH FROM CM TO CR	LDL	13
*	3A	LOAD FILE FROM CM INTO VPR	LDF	14
	3B	LOAD RH FROM CM TO CR	LDR	14
	40	AND LOGICAL W IN CM TO VPR	AN	19
	41	AND LOGICAL LH IN CM TO VPR	ANL	19
	42	BRANCH UNCONDIT TO ROM	BR	49
	43	AND LOGICAL RH IN CM TO VPR	ANR	19
	44	OR LOGICAL W IN CM TO VPR	OR	18
	45	OR LOGICAL LH IN CM TO VPR	ORL	18
	46	BRANCH UNCONDIT TO ROM, SAVE PC IN VPR	BRS	48
	47	OR LOGICAL RH IN CM TO VPR	ORR	18
	48	LOGICAL EQUIVALENCE W CM TO VPR	EQ	21
	49	LOGICAL EQUIVALENCE LH CM TO VPR	EQL	21
	4A	BRANCH UNCONDIT TO ABSOL CM	BCA	49

OP CODE INDEX (CONTINUED)

OP CODE	INSTRUCTION	MNEM CODE	PAGE NO.
4B	LOGICAL EQUIVALENCE RH CM TO VPR	EQR	22
4C	EXCLUSIVE OR W IN CM TO VPR	EX	20
4D	EXCLUSIVE OR LH CM TO VPR	EXL	20
4E	BRANCH UNCONDIT TO AUG ABSOL CM	BCA	49
4F	EXCLUSIVE OR RH CM TO VPR	EXR	20
50	ADD W IN CM TO VPR	AD	15
51	ADD LH IN CM TO VPR	ADL	15
52	BRANCH UNCONDIT TO CM ABSOL, SAVE PC IN VPR	BCAS	48
53	ADD RH IN CM TO VPR	ADR	16
54	SUBT W IN CM FROM VPR	SU	16
55	SUBT LH IN CM FROM VPR	SUL	16
56	BRNCH UNCOND TO ABS ROM,SV PC IN CM LOC ID,AUG	BRSM	50
57	SUBT RH IN CM FROM VPR	SUR	17
58	PUSH INTO STACK	PUSH	25
59	PULL FROM STACK	PULL	26
5A	BRANCH UNCONDIT TO REL PC	BPC	49
5B	MOD STACK	MOD	26
5C	EXECUTE CM	EXEC	28
5D	LOAD EFFECTIVE ADDRESS	LDEA	27
5E	BRANCH UNCONDIT TO AUG REL PC	BPC	49
5F	ANALYZE CM	ANAZ	27
60	SHIFT ARITH W	SHA	39
61	AND LOGICAL IMMED HW TO VPR	ANHI	31
62	LOAD IMMED W INTO CR	LDI	30
63	AND LOGICAL IMMED B TO VPR	ANBI	31
64	SHIFT LOGICAL W	SHL	39
65	OR LOGICAL IMMED HW TO VPR	ORHI	30
66	LOAD IMMED HW INTO CR	LDHI	30
67	OR LOGICAL IMMED B TO VPR	ORBI	30
69	LOGICAL EQUIVALENCE IMMED HW TO VPR	EQHI	32
6B	LOGICAL EQUIVALENCE IMMED B TO VPR	EQBI	32
6C	SHIFT CYCLIC W	SHC	39
6D	EXCLUSIVE OR IMMED HW TO VPR	EXHI	31
6E	LOAD IMMED B INTO CR	LDBI	30
6F	EXCLUSIVE OR IMMED B TO VPR	EXBI	31
70	ADD IMMED W TO VPR	ADI	32
71	ADD IMMED HW TO VPR	ADHI	32
72	LOAD IMMED W INTO VPR	LDI	30
73	ADD IMMED B TO VPR	ADBI	33
74	SUBT IMMED W FROM VPR	SUI	33
75	SUBT IMMED HW FROM VPR	SUHI	33
76	LOAD IMMED HW INTO VPR	LDHI	30
77	SUBT IMMED B FROM VPR	SUBI	33
78	COMPARE IMMED W WITH VPR, SIE	CEI	34
79	COMPARE IMMED HW WITH VPR, SIE	CEHI	34
7B	COMPARE IMMED B WITH VPR, SIE	CEBI	34
7C	COMPARE IMMED W WITH VPR, SINE	CNI	34
7D	COMPARE IMMED HW WITH VPR, SINE	CNHI	35
7E	LOAD IMMED B INTO VPR	LDBI	30
7F	COMPARE IMMED B WITH VPR, SINE	CNBI	35
80	LOAD WD TO VPR FROM VPR	LD	13
81	LOAD HW FROM VPR TO VPR	LDH	13
82	RESET VP FLAG IN CR	VPR	51

OP CODE INDEX (CONTINUED)

OP CODE	INSTRUCTION	MNEM CODE	PAGE NO.
83	LOAD B FROM VPR TO VPR	LDB	13
84	LOAD WD FROM VPR TO CR	LD	13
85	LOAD HW FROM VPR TO CR	LDH	13
86	SET VP FLAG IN CR	VPS	51
87	LOAD B FROM VPR TO CR	LDB	13
88	LOAD WD FROM CR TO VPR	LD	13
89	LOAD HW FROM CR TO VPR	LDH	13
8A	TEST CR FOR 0 AND SKIP IF EQ TO 0	VPTZ	51
8B	LOAD B FROM CR TO VPR	LDB	13
8C	LOAD WD FROM CR TO CR	LD	13
8D	LOAD HW FROM CR TO CR	LDH	13
8E	TEST CR, SKIP IF EQ TO 1	VPTO	51
8F	LOAD B FROM CR TO CR	LDB	13
90	STORE WD FROM VPR TO VPR	ST	11
91	STORE HW FROM VPR TO VPR	STH	11
92	TEST FOR ANY 0 IN CR LH RESET	TRZL	40
93	STORE B FROM VPR TO VPR	STB	11
94	STORE WD FROM CR TO VPR	ST	11
95	STORE HW FROM CR TO VPR	STH	11
96	TEST FOR ANY 0 IN CR RH RESET	TRZR	42
97	STORE B FROM CR TO VPR	STB	11
98	STORE WD FROM VPR TO CR	ST	11
99	STORE HW FROM VPR TO CR	STH	11
9A	TEST FOR ANY 1 IN CR LH RESET	TROL	41
9B	STORE B FROM VPR TO CR	STB	11
9C	STORE WD FROM CR TO CR	ST	11
9D	STORE HW FROM CR TO CR	STH	11
9E	TEST FOR ANY 1 IN CR RH RESET	TROR	42
9F	STORE B FROM CR TO CR	STB	11
A0	TEST W ARITH, BRANCH IF CR EQ 0	TZ	43
A1	TEST HW ARITH, BRANCH IF CR EQ 0	TZH	43
A3	TEST B ARITH, BRANCH IF CR EQ 0	TZB	43
A4	TEST W ARITH, BRANCH IF CR NEQ 0	TN	44
A5	TEST HW ARITH, BRANCH IF CR NEQ 0	TNH	44
A7	TEST B ARITH, BRANCH IF CR NEQ 0	TNB	44
A8	TEST W ARITH, BRANCH IF CR GR OR EQ 0	TP	44
A9	TEST HW ARITH, BRANCH IF CR GR OR EQ 0	TPH	45
AB	TEST B ARITH, BRANCH IF CR GR OR EQ 0	TPB	45
AC	TEST W ARITH, BRANCH IF CR LS 0	TM	45
AD	TEST HW ARITH, BRANCH IF CR LS 0	TMH	45
AE	BRANCH UNCONDIT TO REL PC, SAVE PC IN VPR	BPCS	48
AF	TEST B ARITH, BRANCH IF CR LS 0	TMB	46
B0	TEST W ARITH, BRANCH IF VPR EQ 0	TZ	43
B1	TEST HW ARITH, BRANCH IF VPR EQ 0	TZH	43
B2	INCR VPR BY 1, BRANCH IF RESULT EQ 0	IBZ	47
B3	TEST B ARITH, BRANCH IF VPR EQ 0	TZB	43
B4	TEST W ARITH, BRANCH IF VPR NEQ 0	TN	44
B5	TEST HW ARITH, BRANCH IF VPR NEQ 0	TNH	44
B6	INCR VPR BY 1, BRANCH IF RESULT NEQ 0	IBN	47
B7	TEST B ARITH, BRANCH IF VPR NEQ 0	TNB	44
B8	TEST W ARITH, BRANCH IF VPR GR OR EQ 0	TP	44
B9	TEST HW ARITH, BRANCH IF VPR GR OR EQ 0	TPH	45
BA	DECR VPR BY 1, BRANCH IF RESULT EQ 0	DBZ	47

OP CODE	INSTRUCTION	MNEM CODE	PAGE NO.
BB	TEST B ARITH, BRANCH IF VPR GR OR EQ 0	TPB	45
BC	TEST W ARITH, BRANCH IF VPR LS 0	TM	45
BD	TEST HW ARITH, BRANCH IF VPR LS 0	TMH	45
BE	DECR VPR BY 1, BRANCH IF RESULT NEQ 0	DBN	47
BF	TEST B ARITH, BRANCH IF VPR LS 0	TMB	46
C0	AND LOGICAL W IN VPR TO VPR	AN	19
C1	AND LOGICAL HW IN VPR TO VPR	ANH	19
C2	TEST UNDER MASK IN CR FOR ANY 0 LH & SKIP	TZL	37
C3	AND LOGICAL B IN VPR TO VPR	ANB	19
C4	OR LOGICAL W IN VPR TO VPR	OR	18
C5	OR LOGICAL HW IN VPR TO VPR	ORH	18
C6	TEST UNDER MASK IN CR FOR ANY 0 RH & SKIP	TZR	37
C7	OR LOGICAL B IN VPR TO VPR	ORB	18
C8	LOGICAL EQUIVALENCE W VPR TO VPR	EQ	21
C9	LOGICAL EQUIVALENCE HW VPR TO VPR	EQH	21
CA	TEST UNDER MASK IN CR FOR ANY 1 LH & SKIP	TOL	37
CB	LOGICAL EQUIVALENCE B VPR TO VPR	EQB	21
CC	EXCLUSIVE OR W IN VPR TO VPR	EX	20
CD	EXCLUSIVE OR HW IN VPR TO VPR	EXH	20
CE	TEST UNDER MASK IN CR FOR ANY 1 RH & SKIP	TOR	37
CF	EXCLUSIVE OR B IN VPR TO VPR	EXB	20
D0	ADD W IN VPR TO VPR	AD	15
D1	ADD HW IN VPR TO VPR	ADH	15
D2	TEST FOR ANY 0 IN CR LH SET & SKIP	TSZL	40
D3	ADD B IN VPR TO VPR	ADB	15
D4	SUBT W IN VPR FROM VPR	SU	16
D5	SUBT HW IN VPR FROM VPR	SUH	16
D6	TEST FOR ANY 0 IN CR RH SET	TSZR	41
D7	SUBT B IN VPR FROM VPR	SUB	16
D8	COMPARE W VPR TO VPR, SIE	CE	23
D9	COMPARE HW VPR TO VPR, SIE	CEH	23
DA	TEST FOR ANY 1 IN CR LH SET	TSOL	40
DB	COMPARE B VPR TO VPR, SIE	CEB	23
DC	COMPARE W VPR TO VPR, SINE	CN	23
DD	COMPARE HW VPR TO VPR, SINE	CNH	24
DE	TEST FOR ANY 1 IN CR RH SET	TSOR	41
DF	COMPARE B VPR TO VPR, SINE	CNB	24
E0	AND LOGICAL W IN CR TO VPR	AN	19
E1	AND LOGICAL HW IN CR TO VPR	ANH	19
E2	TEST UNDER MASK IN CR FOR ALL 0 LH & SKIP	TAZL	38
E3	AND LOGICAL B IN CR TO VPR	ANB	19
E4	OR LOGICAL W IN CR TO VPR	OR	18
E5	OR LOGICAL HW IN CR TO VPR	ORH	18
E6	TEST UNDER MASK IN CR FOR ALL 0 RH & SKIP	TAZR	38
E7	OR LOGICAL B IN CR TO VPR	ORB	18
E8	LOGICAL EQUIVALENCE W CR TO VPR	EQ	21
E9	LOGICAL EQUIVALENCE HW CR TO VPR	EQH	21
EA	TEST UNDER MASK IN CR FOR ALL 1 LH & SKIP	TAOL	38
EB	LOGICAL EQUIVALENCE B CR TO VPR	EQB	21
EC	EXCLUSIVE OR W IN CR TO VPR	EX	20
ED	EXCLUSIVE OR HW IN CR TO VPR	EXH	20
EE	TEST UNDER MASK IN CR FOR ALL 1 RH & SKIP	TAOR	38
EF	EXCLUSIVE OR B IN CR TO VPR	EXB	20

DATA CHANNELS

SECTION D

TABLE OF CONTENTS

GENERAL

The Data Channel Unit (DCU), figure 1, provides the interfacing required between the CM and the disc storage units. The DCU consists of an expandable Memory Bus Control (MBC), two general purpose Data Channels (DC) for data buffering and memory addressing, and two Disc Channel Controllers (DCC) to match the storage device to the Data Channel. The DCCs are controlled by Communication Register bits in the PPU.

MEMORY BUS CONTROL

The MBC is a 1x4 expansion unit providing four ports, each identical to the CM bus, serviced in a round robin sequence. The maximum data rate of a particular DCC is dependent on the number of active channels since they all share the MBC. The MBC can be expanded to provide 16 data ports, as shown in figure 2 by using four more 1x4 expansion units.

DATA CHANNEL

The DC, figure 3, provides data buffering between the 256 bit CM interface and the 32 bit device interface. File 1 consists of eight 32 bit registers (R00 thru R07) that have bidirectional access to the CM and the eight registers (R10 thru R17) in File 2. The General Register File consists of four 32 bit registers, R20 thru R23. The DC Data Buffer is a 32 bit register whose data inputs are controlled by the DCC.

The DC is controlled by an 8 bit Control Byte from the DCC. The code consists of two hexidecimal digits. The leftmost digit identifies a source register, the right digit identifies a destination register. Table 1 lists the hexdigit-register assignments. Examples of typical control bytes and the resulting action are given in Table 2. The source digit in the control byte controls the selection of a register onto the General Data Bus, and the destination digit enables the appropriate register to accept the data. Deviations from this procedure occur for transfer between File 1 and File 2; and CM accesses.

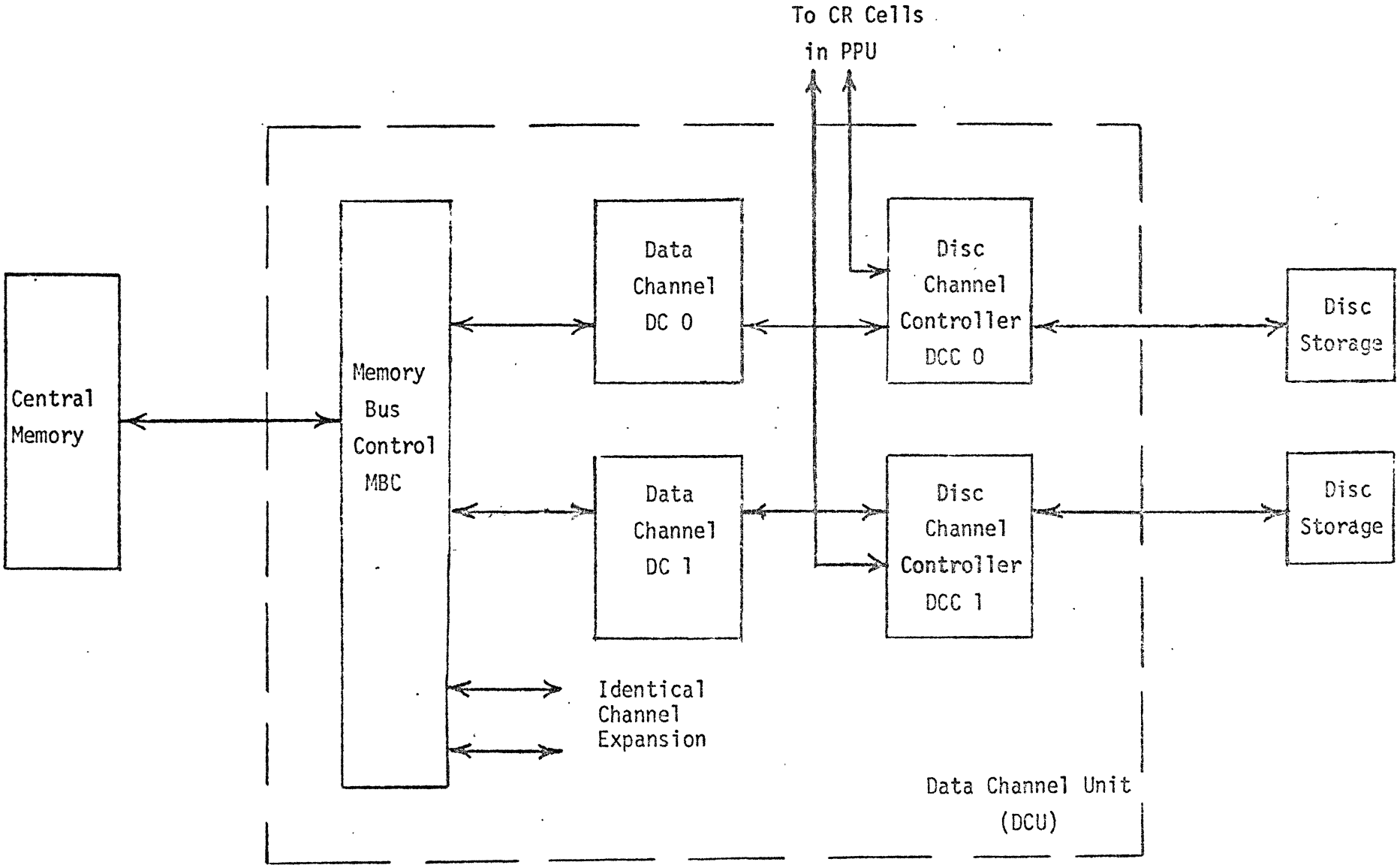


Figure 1. Data Channel Unit Block Diagram

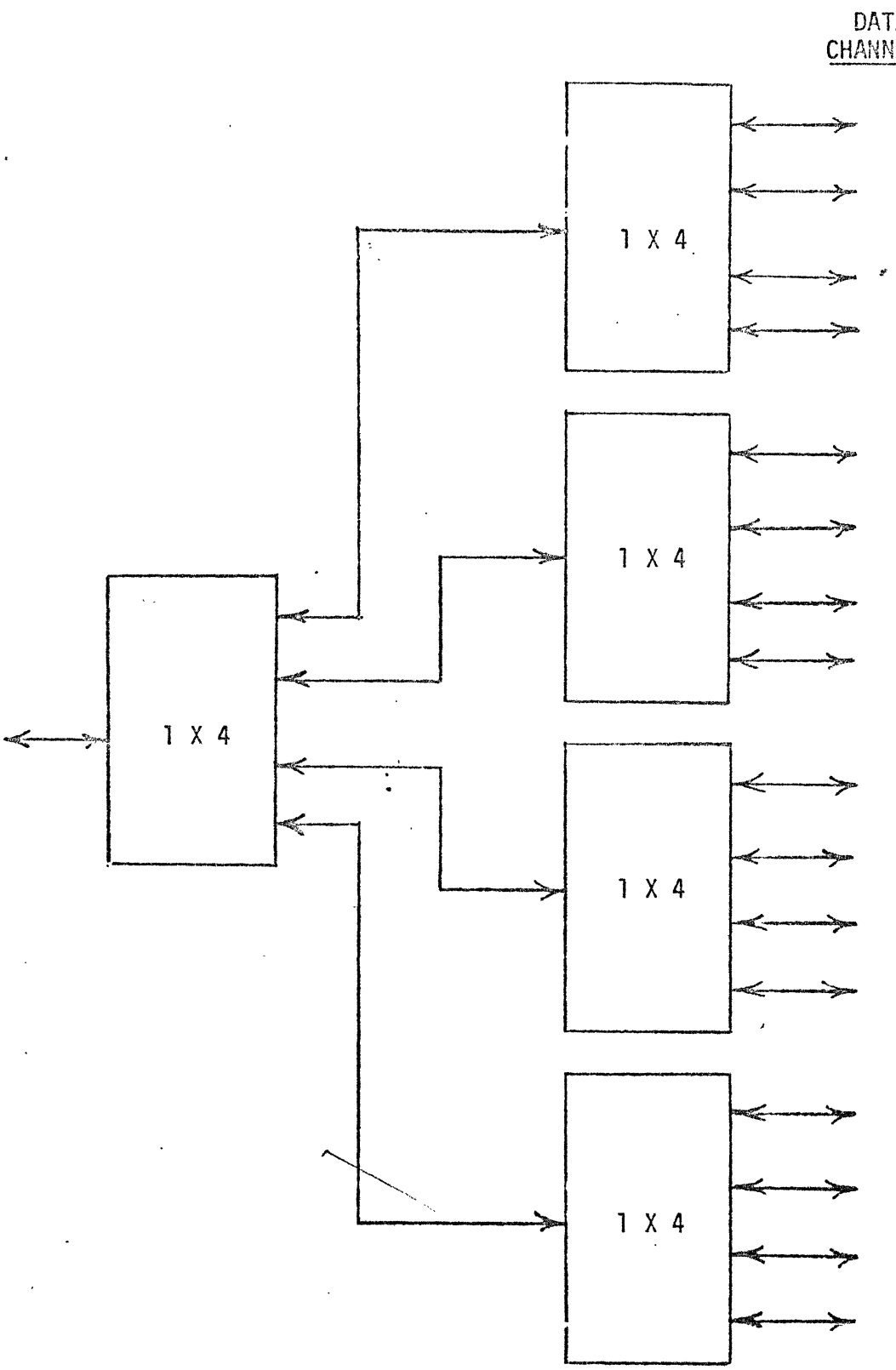


Figure 2. Expanded Memory Bus Control Block Diagram

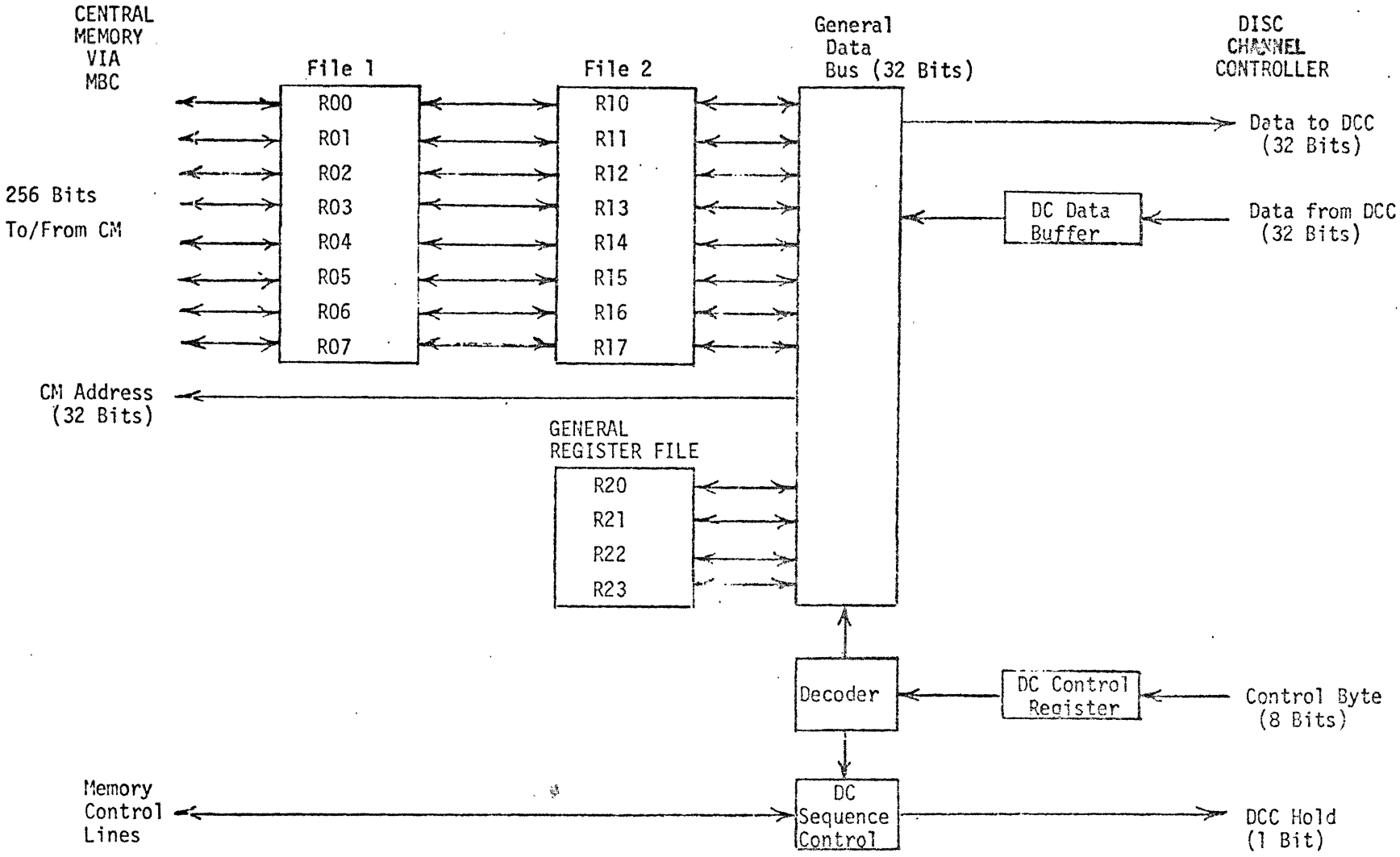


Figure 3. Data Channel Block Diagram

Table 1. Hexdigit-Register Assignments

Hexdigit	Register
0	DCC
1	File 1
2	File 2
3	CM
4	R20
5	R21
6	R22
7	R23
8	R10
9	R11
A	R12
B	R13
C	R14
D	R15
E	R16
F	R17

Table 2. Typical Control Bytes and Resulting DC Action

Typical Control Bytes	Resulting Action
59	R21 transferred to R11
95	R11 transferred to R21
F4	R17 transferred to R20
0A	DCC transferred to R12
A0	R12 transferred to DCC
12	File 1 (256 bits) transferred to File 2
21	File 2 (256 bits) transferred to File 1
1B	R03 (32 bits) transferred to the adjacent register in File 2 (R13)
B1	R13 (32 bits) transferred to the adjacent register in File 1 (R03)
35	Read CM at the address contained in R21
53	Write in CM at the address contained in R21
Special Codes	
00	No-op

The two permitted transfers between File 1 and File 2 include complete transfers and register transfers. All 256 bits of File 1 may be parallel transferred to File 2 or all 256 bits of File 2 may be parallel transferred to File 1. Any one of the eight registers in File 1 (32 bits) may be parallel transferred to the adjacent register in File 2 or conversely any one of the 8 registers in File 2 may be parallel transferred to the adjacent register in File 1.

The position of the digit "3" in the Control Byte specifies the type of CM access, either CM Read or CM Write. The other digit in the Control Byte identifies the register which contains the CM address for the memory access. Sequence Control determines the type of CM access and sets appropriate control lines to the MBC. For CM Read, when Read Data is true, all 256 bits of an octet in CM will be loaded into File 1. For CM Write, the data in File 1 will be written in CM as permitted by the zone control byte in the memory address register.

It is possible that the DCC will send some Control Bytes faster than the DC can execute them. When its occurs the DC will send the DCC a Hold signal, stopping the DCC at that point in its program. When the DC executes the command, the Hold signal will be cleared and the DCC can continue its program.

DISC CHANNEL CONTROLLER

The DCC, figure 4, is designed to match the characteristics of the Disc Module to the requirements of the ASC system. In order to do this, the DCC must accomplish the following tasks:

1) Monitor control lines connected to CR bits in the PPU.

2) Obtain from CM a Communications Area, set up by PPU software, describing the job the DCU is to perform.

3) Monitor the condition of the disc module at all times and issue instructions to the disc as required.

4) Supply Control Bytes to the DC.

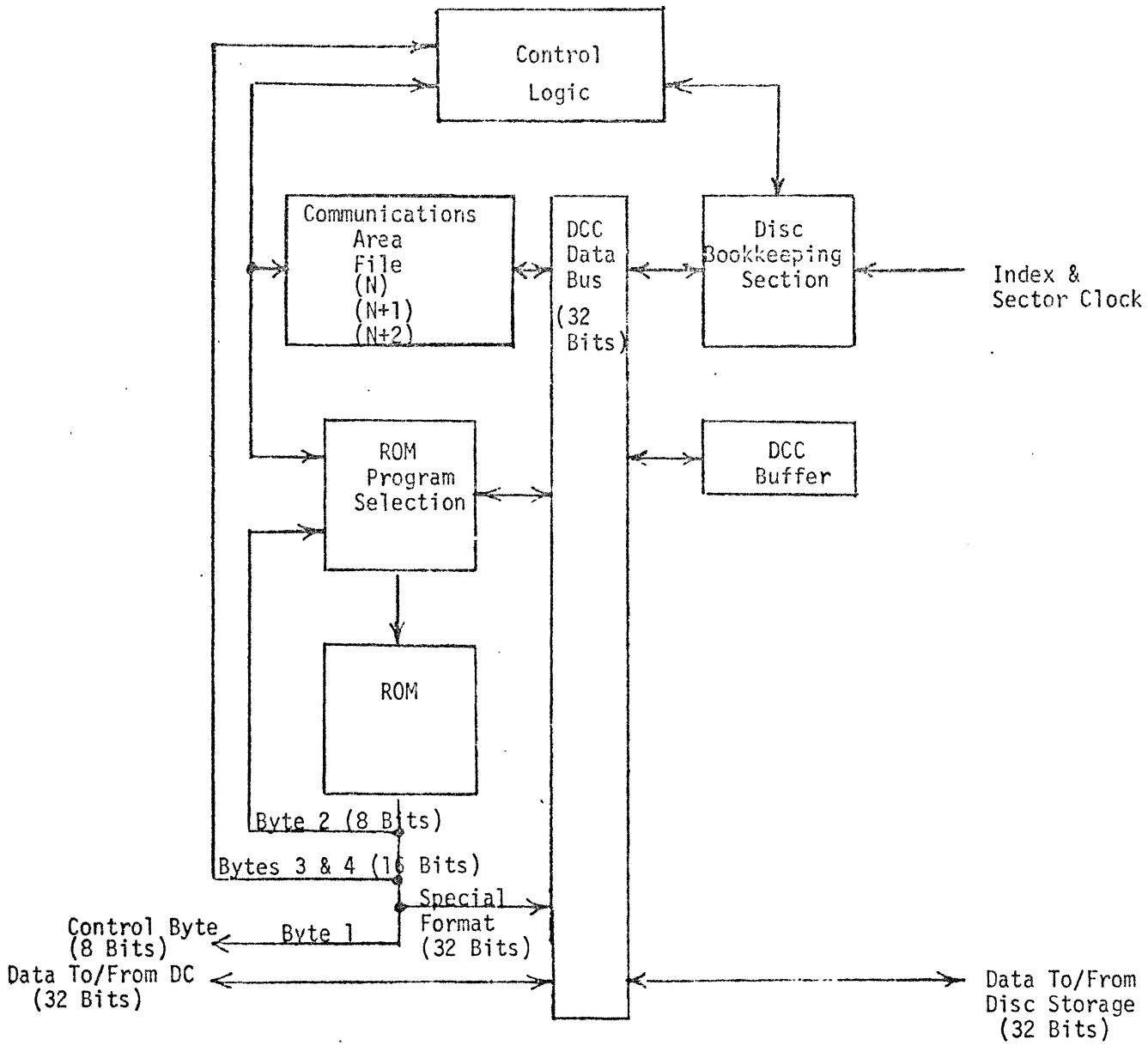


Figure 4. Disc Channel Controller Block Diagram

5) Accept and send data to the DC.

6) Accept and send data to the disc.

COMMUNICATIONS AREA FILE

Each job that the DCU is to perform is described by a communications area in the CM as follows:

A. Operation -- Type of job to be performed

1) Read - Read the disc and transfer the data to the CM

2) Write - Write on the disc the data obtained from the CM.

3) Erase - Write "0's" on the disc.

4) Read Check - Read the disc and check for parity, do not transfer the data to CM.

B. CM Address -- The address in CM where the data is to be stored for the Read operation or the address in CM where the data is to be found for the Write operation. This address may be absolute or virtual. If virtual, the DCC must perform the Map Routine (ROM) to obtain the absolute address. CM address has no meaning for Erase and Read Check operations.

C. Disc Address -- The address on the disc where the data is to be stored for the Write operation, obtained for the Read operation, zeroed for the Erase operation, or checked for parity on the Read Check operation.

D. Map Pointer -- An address in CM where a map image of the memory is located. It is used when the CM address is virtual. See Map Routine, ROM.

E. Link Address -- The address in CM where the Communications Area for the next job is located.

F. Storage area pointers (12) each containing a CM address and word count so that DM data may be written or read from several different noncontiguous areas of CM.

DISC BOOKKEEPING SECTION

The DCC Disc Bookkeeping Section interfaces the DCC Data Bus and a Disc Interface Unit (DIU). A DIU will support two Disc Modules which are asynchronous to each other. Each DM sends the DCC a sector clock, indicating the beginning of a new sector, and an index mark, indicating sector zero on the disc. The DCC counts the sector clocks and always knows the disc position. This is done by counters AAR "0" and "1". The DM must be addressed one sector in advance, therefore, the AARs are actually one sector ahead of the true DM position. The DM being used for a job is selected by SELDM from the control logic. The selected AAR is compared with the sector portion of the Disc Address for the current job. The output of the Comparator Comp is sent to the Control Logic. The Control Logic determines if the job is to be processed in this sector and which ROM programs are required.

READ ONLY MEMORY

The DCC uses a Read Only Memory (ROM) to direct the channel to perform fixed sequences of operations. The ROM will also supply special fixed CM addresses which are unique to each channel. Control Logic is responsible for decoding and executing the ROM instruction and performing the tests and branches called for. The ROM has a capacity of 256 words. Each word consists of 33 bits. The least significant bit is used to determine the parity (true or compliment) of the word. The ROM word may have either of 2 formats.

Normal Format	Byte 0	Byte 1	Byte 2	Byte 3
Special Format	32 bit word			

In the Normal Format the 32 bits will be divided into four 8 bit bytes. The bytes will be used as follows:

Byte 1	DC Control Byte
Byte 2	Next ROM Address
Bytes 3 and 4	DCC Internal Control

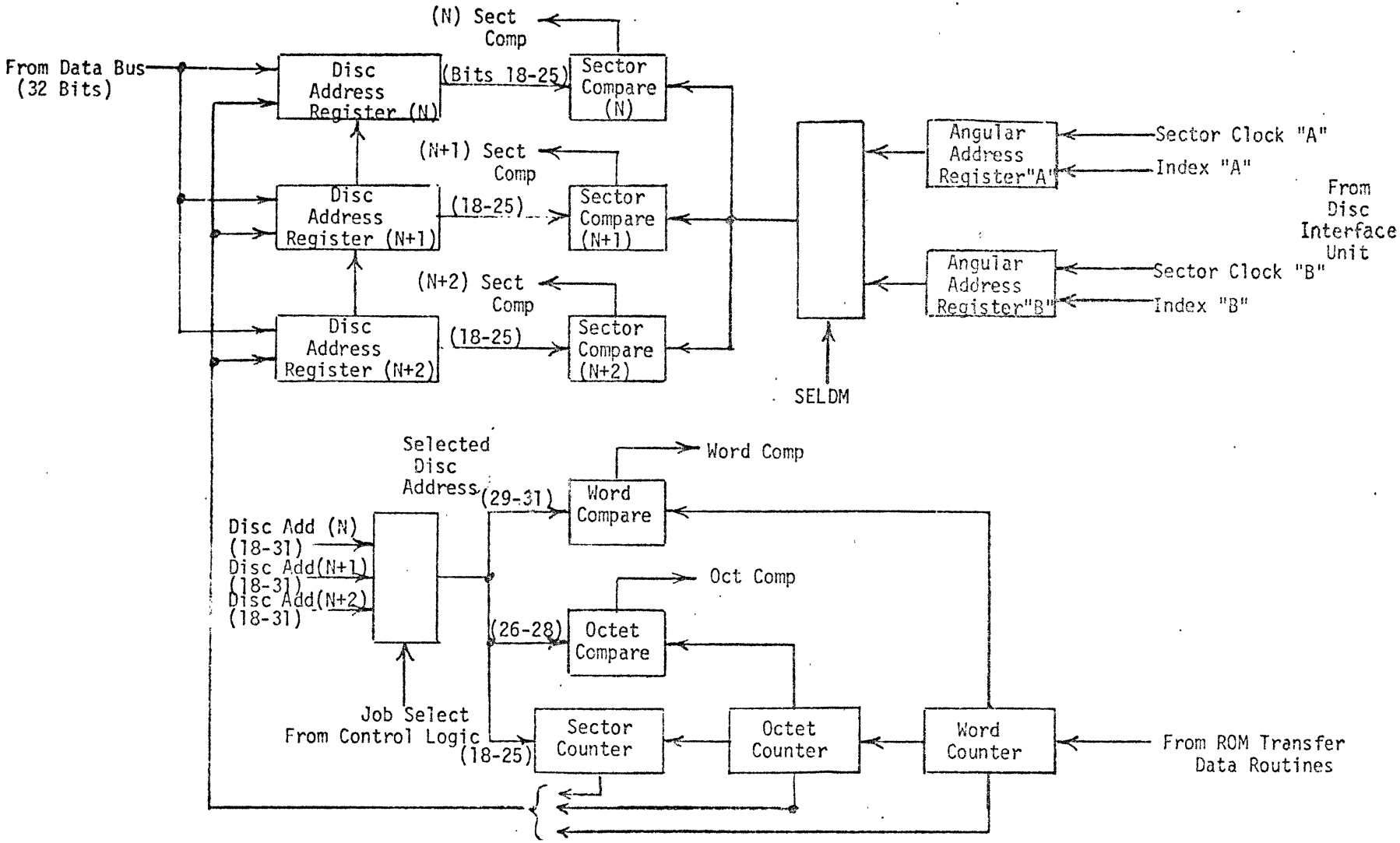


Figure 5. Disc Bookkeeping Block Diagram

In the Special Format the bits 29-31 are 111. The 32 bit word is loaded into the DC Register R20, the ROM address is incremented.

Routines programmed into the ROM are:

1) LA - Look Ahead

2) SW - Status Word

3) DS - Initial Start

4) AA - Angular Address Update

5) TDW - Transfer Data Write (CM→DM)

6) TDR - Transfer Data Read (DM→CM)

7) MAP - CM address Mapping

Control Logic forces the ROM address to the beginning of each program as required. Byte 2 of the ROM word then dictates the next address. Control Logic is responsible for synchronizing the ROM address with the DC and DU. In the case of conditional branches, the control logic may force a jump in a program's normal addressing sequence.

THE TERMINAL CHANNEL CONTROLLER (TCC)

SCOPE

Identification

This description covers the requirements of the Terminal Channel Controller.

Usage as Part of the Total System

The TCC is part of the Data Channel Unit (DCU) which provides the required interfacing between Central Memory (CM) and external units. The DCU consists of general purpose Data Channels (DC) which communicate with CM and are attached to the external units via special purpose channel controllers. The present DCU configuration contains two types of channel controllers. The Disc Channel Controller (DCC) matches a Disc Interface Unit (DIU) and consequently Disc Modules to the DC. The Terminal Channel Controller (DCC) matches Selector Interface Units (SIU) to the DC. In this document SIU will be used to refer to any device that might be attached to the TCC. The incorporation of the TCC irto the DCU is shown in figure 6.

The TCC has two SIU ports. However, the TCC port addressing allows for expanding up to 32 ports.

Usage

In order to match the characteristics of the SIU to the requirements of the ASC, the TCC must accomplish the following:

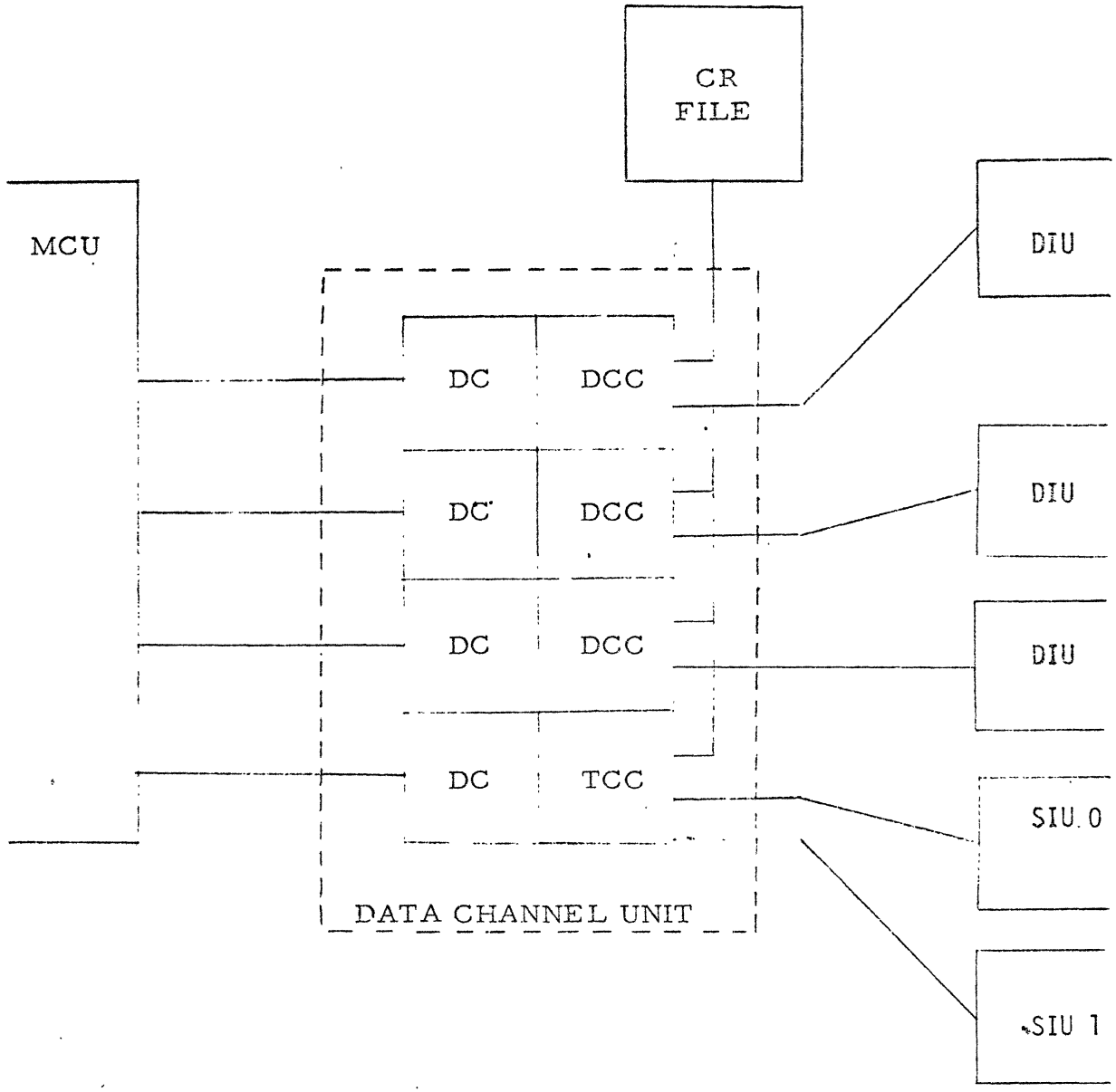


Figure 6 TCC Incorporation In Data Channel

(1) communicate with the PPU via CR bits,

(2) obtain from CM a Communications Area (CA), set up by PPU Software describing the job the channel is to perform,

(3) supply commands to the DC,

(4) accept and send data and status to the DC, and

(5) accept and send data and status to the SIU.

Component Identification

A block diagram of the TCC is shown in figure 7.

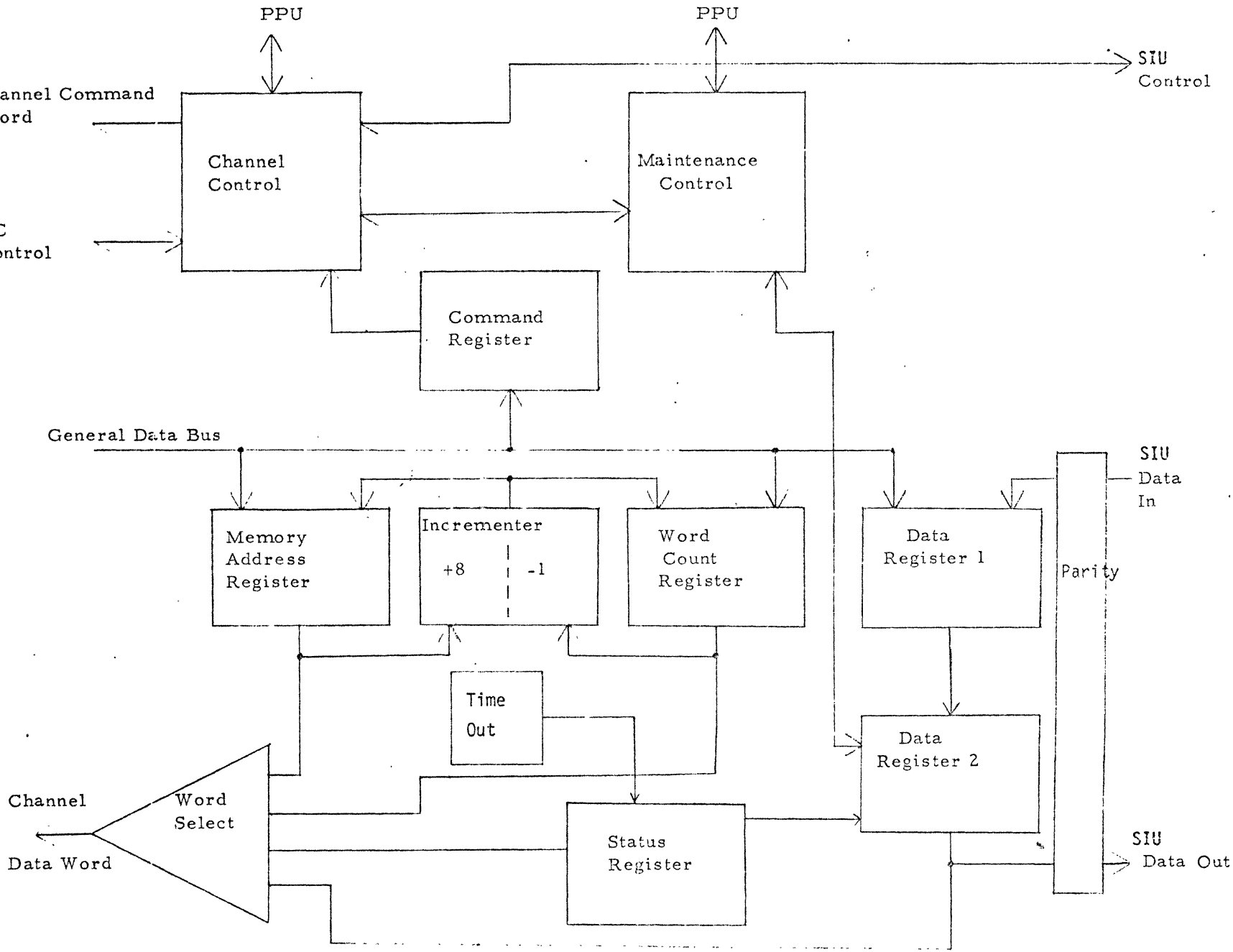


Figure 7 TCC Block Diagram

REQUIREMENTS

Performance

1. Primary Performance Characteristics are:

   a. All TCC Operations are controlled by ASC software with bits in the CR file and with Communication Areas (CA) in CM.

   b. Control bits in the CR file give ASC software the ability to:

      (1) Start the TCC

      (2) Cause the TCC to abort any chain of jobs

      (3) Accept status reports from the TCC when part of the DC or CM is down

      (4) Communicate with the TCC regarding the need to reorder jobs

      (5) Control maintenance operations within the TCC

      (6) Switch the TCC from a Low Priority CA chain to a High Priority chain.

   c. ASC software uses CAs to assign jobs to the TCC.

   d. Communication between the TCC and SIU consists of 32-bit data words and 32-bit control words. Each 32-bit word is accompanied by two additional bits, one to differentiate between data and control words and one parity bit. A control word consists of up to 16 hardware status bits in the most significant half (bits 0-15) and a 16-bit control message in the remaining portion (bits 16-31).

   e. The TCC operates on a half duplex basis. However, it does have the capability of accepting a control

word from the SIU when in the Write mode. Such an occurrence always constitutes an abort condition.

f. The TCC is capable of supporting a sustained word transfer rate of 500,000 32-bit words per second (2M bytes/sec).

g. The TCC responds to ASC system initialization and sends a clear signal to the DC and to the SIU.

2. Physical Characteristics.

a. The TCC is a TTL device which operates on the 500 ns TTL clock.

b. The TCC hardware is located in the ASC column DC3-TTL1 which is adjacent to the DC-DCC ECL column.

TCC Definition

1. Terminal Data Channel Programming Interface

The Terminal Data Channel provides the interface between the ASC Programming System and the hardware attached to the channel. The Programming System communicates directly with hardware through the Communications Area (CA) and the CR File.

a. Communications Area (CA)

The CA format is shown in figure 8. The areas in the CA which are not presently being used may be used by software for its own purposes.

Word 0 of the CA is always located on an an octet boundary. The most significant bit of this word (bit 0) is always a 1. The next 7 bits (bits 1-7) are not used by the TCC. The

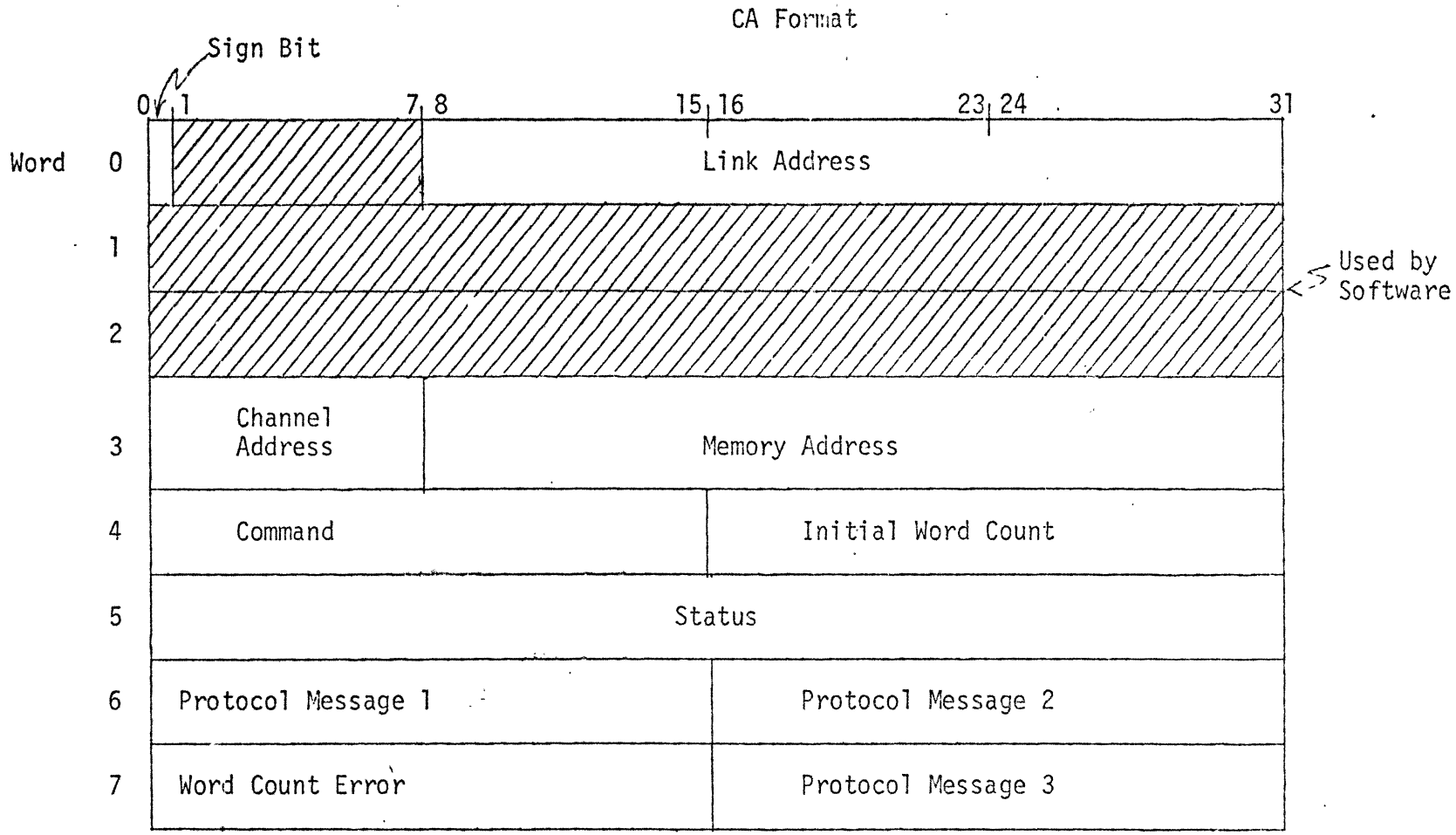


Figure 8 Communication Area Format

remaining 24 bits (bits 8-31) contain the Link Address which points to the next CA in the chain. This Link Address is stored in a DC General Register at the beginning of the execution of the CA.

Word 1 of the CA is not used by the TCC. The MSB's (bits 0-7) of Word 3 contain the Channel Address (CAD) for the TCC. This is the address of the TCC port that is to be involved in this transfer. It is stored in the TCC at the beginning of execution of the CA. The remaining 24 bits (bits 8-31) of this word contain the Memory Address where the data for this CA is to be obtained or stored. This address is stored in a DC General Register at the beginning of execution of the CA.

The most significant half of Word 4 (bits 0-15) contains the Command Field shown in Figure 9.

Control Word options 1, 2, and 3 refer to options which the TCC may take with regard to the sending and receiving of Control Words before and after block transfers. Control Word 1 option indicates to the TCC that it is to send a Control Word to the SIU before any data transfer in either direction is started. This Control Word is formed by the TCC and consists of Control Message 1

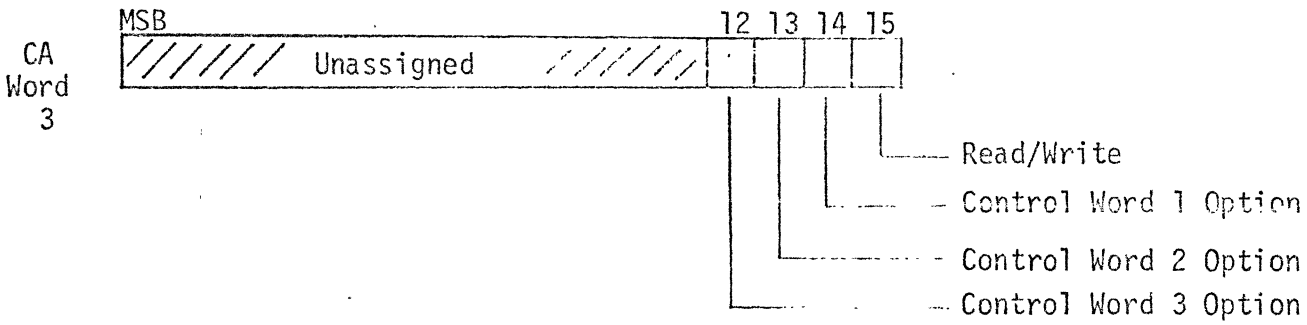


Figure 9. Command Field

taken from CA word 5. Control Word 2 option indicates to the TCC that it should take the Control Message in Control Message 2 position and send it to the SIU at the completion of data transfer in the case of a Write and at the completion of data transfer plus possibly one Control Word transfer (Option 3) in the case of a Read. Control Word 3 option indicates to the TCC that it should expect to receive a Control Word at the completion of data transfer in the case of a Read and at the completion of data transfer plus possibly one Control Word transfer (Option 2) in the case of a Write. This Control Message is to be placed in Control Message 3 position of the CA.

Read/Write is indicated by bit 15 of CA word 2. Read is indicated by a one in position 15 and a Write is indicated by a zero.

The right half of Word 4 (bits 16-31) contains Initial Word Count. This is the number of words that are to be transferred by execution of this CA.

Word 5 of the CA contains the Status field. The meaning of the 4 MSBs (bits 0-3) is fixed. The 16 LSBs are used for reporting status from the SIU.

Bit 0 of the Status field contains the complete bit. The TCC sets this bit after it

has completed execution of a CA (either satisfactorily or unsatisfactorily).

Bit 1 contains Illegal Word Count. This flag is set to indicate that the TCC received a Control Word from the SIU when the count contained in the TCC Word Count Register was not zero or that a Data Word was received when count was zero. In the first case, the remaining Word Count is written in the CA in the Word Count Error field along with the Control Message that was received.

Bit 2 contains Time Out (TO). This flag is set to indicate that the TCC timed out during a TCC - SIU transfer.

Bit 3 contains the Parity Error (PE) flag which indicates that a parity error was detected in an SIU to TCC transfer.

Status from the SIU is written into the 16 LSBs of the status word upon receiving a control word (Control Word 3) from the SIU. The status is located in the 16 MSBs of this control word.

Word 6 of the CA contains Control Message 1 and Control Message 2. These are "output" Control Messages (i.e., they will always be sent to the SIU). The contents of the Control Messages are determined by software, and their use is controlled by Options 1 and 2 in the Command field. Word 6 is stored in DC General

Registers at the beginning of the CA.

Word 7 contains the Word Count Error in its 16 MSBs (bits 0-15). Count in the TCC Word Count Register is loaded into this halfword any time the TCC receives a Control Word, when Word Count in the register is not 0. The 16 LSBs (bits 16-31) contain Control Message 3. Its contents are software designated, and its use is software controlled via Option 3 in the Command.

Any control word received by the TCC at a time other than that specified by Option 3 is also placed in Control Message 3 position. The 16 MSBs are placed in the low order position of the status word. This is followed by an abort of the CA.

Communications Area Chains

TCC CAs are looped by the Programming System as shown in figure 10. The TCC begins the loop at the Header which is the dedicated memory address holding the address of the first CA. It stops processing CAs when it comes back to the Header after processing the CAs in the loop. The CAs may be relinked by the Programming System in order to process new requests. However, since the TCC only goes through a loop once, all relinking must be done ahead of where the TCC is working in the loop. Neither the Programming System nor the TCC are committed to

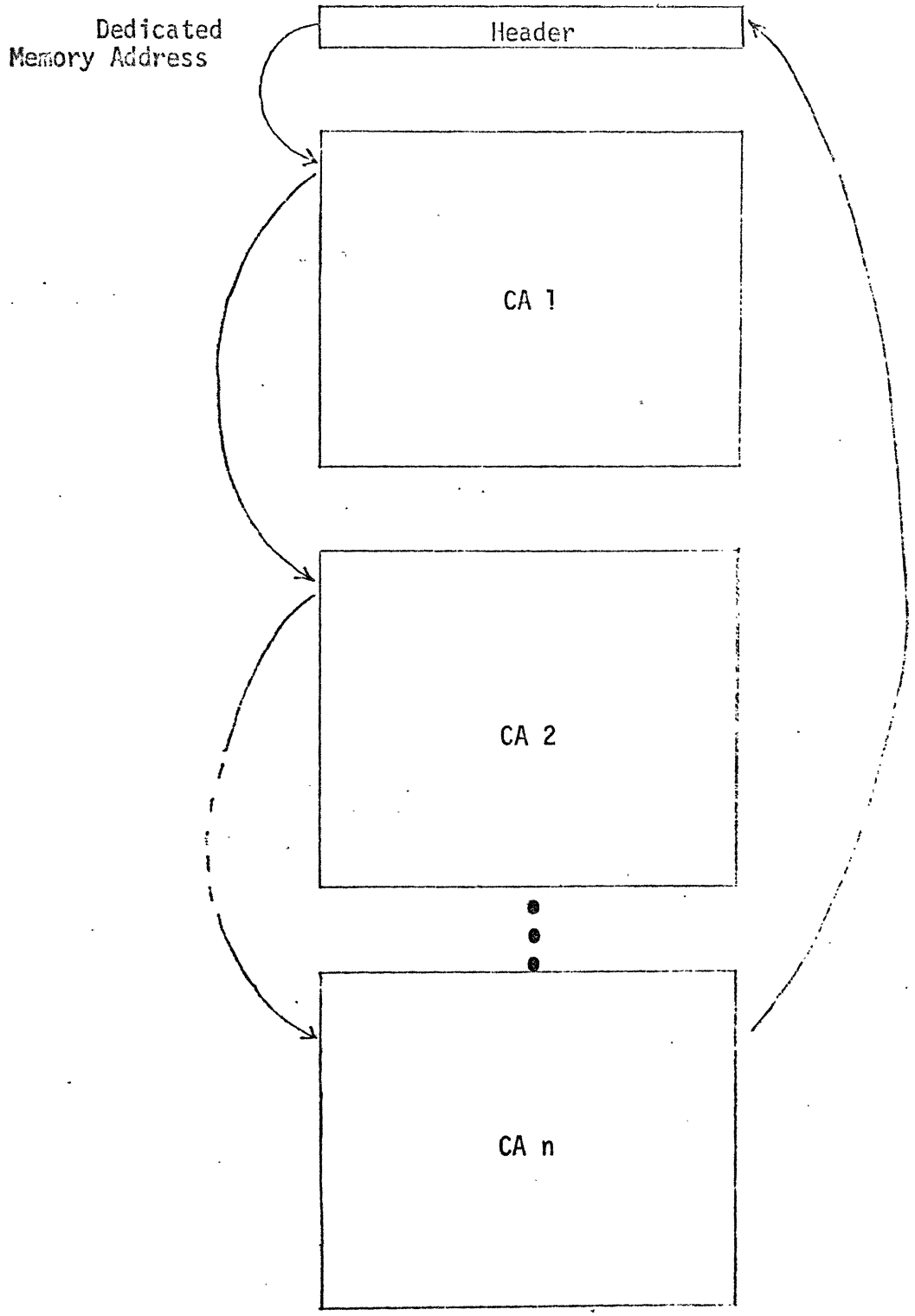


Figure 10 CA Linked Lists Structures

do a job until the CA is fetched.

The sign bit (bit 0) of the Link Address (word 0 of the CA or Header) indicates whether the accessed memory area is actually a CA or is in reality a Header. All CAs have a sign bit of one in the Link Address. A zero in this position indicates a Header. After the TCC has initially accessed the Header to get it started in the loop, it will treat the Header as a CA when it accesses it again. However, upon checking the sign bit and finding it to be a zero, the TCC will stop.

There are low priority and high priority CA chains built for the TCC. Each type of chain has its own Start bit in the CR File and its own Header in a dedicated memory address.

If the TCC is operating in the low priority chain when it receives the high priority Start, it will finish the CA that it is presently processing, save the low priority link address, go to the high priority chain (via its Header) and do all of the CAs in it, and then return to the low priority chain using the link address that it had saved previously. If both Start bits are set at the same time, the TCC will complete the high priority chain before going to the low priority header.

Both Headers always have a zero in bit 0 and are located on an octet boundary.

When the Programming System is ready to relink CAs, it sets the Relink Request bit in the CR File. It cannot continue with relinking until the TCC sets the Relink OK bit in the CR File. This bit is set as soon as the TCC receives Relink Request if the TCC is in the process of performing a CA. However, if the TCC is involved in linking when it receives Relink Request, it continues to the next CA and fetches it before setting Relink OK.

When the TCC finishes the data transfer associated with a given CA and finds Relink OK set, it stops and does not complete execution of the CA until Relink OK is reset by the Programming System (Relink Request is to be reset before Relink OK is reset). The Programming System resets these bits when relinking has been completed. When Relink OK is reset, the TCC reaccesses the CA in order to get the Link Address, which may have been changed, and then continues processing the CA.

During relinking, the Programming System recognizes that the TCC is operating on the first CA in the loop which does not have a complete bit set in the status field of the CA. The Programming System is not allowed to remove this CA from the chain. Note that if there is a high priority and a low priority chain the possibility exists that there will be a CA in each chain that the Programming System will have to leave in.

Protocol

To permit a certain amount of flexibility in the communications Protocol, the TCC accepts protocol options based on the three control word option bits of the CA Command. The three optional control words and their relationship to data transfer are shown in figure 11.

When Control Word 1 Option in the CA Command is 0, it indicates to the TCC that a Control Word should be transferred to the SIU at the beginning of the CA before any data is transferred. The TCC forms this Control Word from Control Message 1 in the CA.

Control Word 2 Option being 0 indicates to the TCC that Control Word 2 is to be sent to the SIU. The actual time at which Control Word 2 is sent depends on whether the TCC is in the Read or Write mode and whether Control Word 3 Option is exercised.

When Option 2 is 0, the TCC is in the Read mode; and Option 3 is 1; Control Word 2 is sent to the SIU after the last data word is received by the TCC (as indicated by the count in the Word Count Register being zero).

When Option 2 is 0, the TCC is in the Read mode; and Option 3 is 1; Control Word 2 is sent to the SIU after Control Word 3 is received.

When Option 2 is 0 and the TCC is in the Write mode, Control Word 2 is sent to the

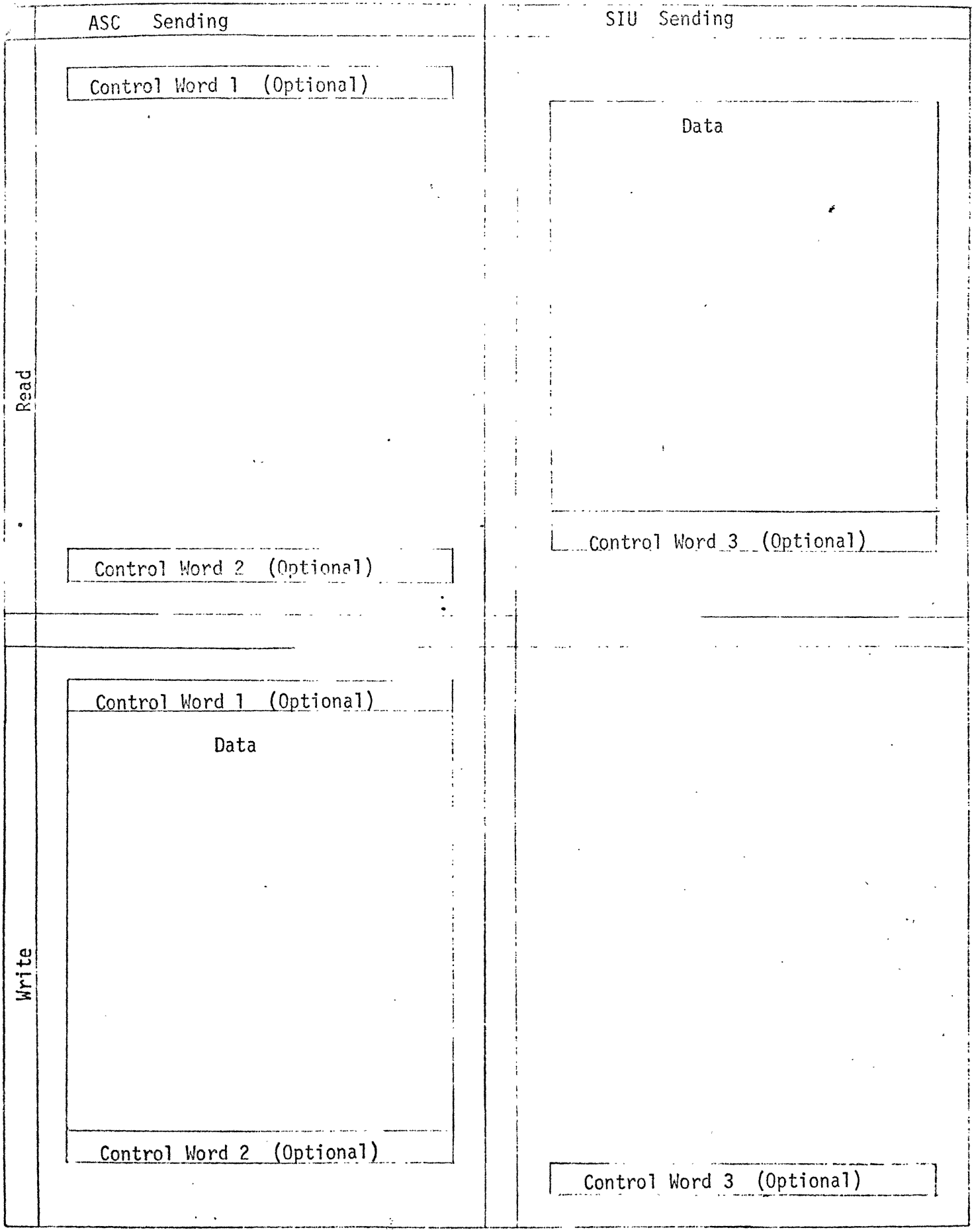


Figure 11 Protocol

SIU after the last Data Word is sent (as indicated by the count in the Word Count Register being zero).

Control Word 3 Option being 0 indicates that the TCC is to receive a Control Word and place the Control Message in Control Message 3 position in the CA. The exact position of Control Word 3 in the protocol is determined by Read/Write and Option 2.

When in the Read mode with Option 3 equal zero, Control Word 3 is received by the TCC immediately after the last data word (as indicated by a zero count in the Word Count Register).

When in the Write mode with Option 3 equal zero and Option 2 equal zero, Control Word 3 is received after sending Control Word 2.

When in the Write mode with Option 3 equal zero and Option 2 equal one, Control Word 3 is received after sending the last Data Word (indicated by zero count in Word Count Register).

Any time the TCC finds it necessary to send status to the SIU, it does so by sending a 32-bit control word with status in the 16 MSBs and Control Word 2 (whether Option 2 is set or not) in the 16 LSBs. Any time Control Word 2 is sent because of Option 2, it is accompanied by status. The status bits sent are Time Out, Parity Error, and Illegal Word Count.

(6) Status from the SIU is always to be accompanied by Control Word 3 and vice versa. Status is in the 16 MSBs with Control Word 3 in the 16 LSBs.

The ASC software can control the TCC via the CR file.

Two bits of CR will be used for Low Priority Start (LPS) and High Priority Start (HPS).

If LPS is 1 and HPS is 0, the TCC uses the Low Priority Header in a dedicated memory location and processes the CA chain. After the TCC finishes all CAs in the chain and comes back to the Header, it resets the LPS and stops.

If HPS is 1 and LPS is 0, the TCC uses the High Priority Header and processes the CA chain, resetting HPS and stopping after it works its way back to the Header.

If LPS is 1 and the TCC has already worked down part of the CA chain when it receives HPS, it will finish the CA that it is currently working on, save its Link Address, and go to the High Priority Header. After completing the High Priority chain, the TCC returns to the Low Priority chain via the Link Address saved out of that chain. If the TCC is between CAs when it receives HPS, it will save the Link Address that it is currently using and go to the High Priority Header.

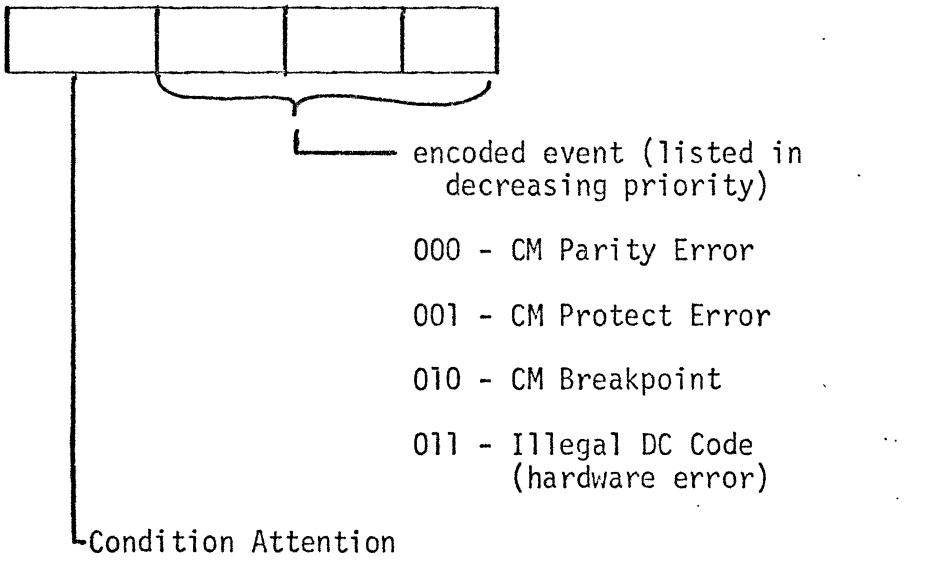
If HPS is 1 and LPS is 1, either because it was set at the same time as HPS or because it was set after operation in the High Priority chain began, the TCC will complete the High Priority chain and then enter the Low Priority chain via the Low Priority Header.

One bit of CR is used for the TCC Abort bit. The following interpretation will be assigned to TCC Abort in conjunction with the Start bits:

LPS or HPS	Abort	
0	0	Channel Inactive/Active
1	0	Start and Channel Active
0	1	Abort Channel Operations
1	1	Abort current CA and proceed to appropriate Header

Software can set both starts and Abort and reset both starts. The TCC can reset both starts and Abort.

A four-bit channel condition field in the CR file is used by the TCC to report certain abnormal events to the PPU. The field is formatted as follows:



When one of the events occurs, the TCC will set the Condition Attention bit, will enter the appropriate event code, and will abort. The Programming System will reset the Condition Attention when appropriate

software action has been taken.

If more than one event is simultaneously detected by the TCC, only the highest priority event is reported.

Two CR bits will be used to communicate with the TCC during a relinking of CAs. When software desires to relink the CAs, a Request Relink bit will be set in the CR file. Within 10 microseconds (generally within 1 microsecond), the TCC will reply by setting the Relink OK bit. The time difference depends on whether the TCC is executing a CA or linking between CAs when it gets Request Relink. If it is between CAs, it will go to the next CA before setting Relink OK. Software resets Request Relink and Relink OK when it finishes relinking.

The TCC communicates with the Common Maintenance Registers (CMR) of the CR file for maintenance purposes. This includes communication with the Common Command Register (CCR) and the Unit Register (UR) allocated for the DCU. The CCR is used to issue maintenance commands and to exercise CM protect and breakpoint options. The UR is used to transfer data between the PPU and the TCC during maintenance.

ASC software will be responsible for performing software timeouts on CA chains to detect TCC and/or DC failures.

OPERATING PANEL

TABLE OF CONTENTS

INTRODUCTION

The ac and dc power control segments of the ASC operating panel can be noted at the upper left of Figure 1. The key turns on the power to the entire computer system, and this status is indicated by the ac ON indicator. To apply dc power to the processors, the dc ON switch is pressed. When all voltages, pressures, and temperatures are within operating ranges and if all interlocks are engaged, the standby indicator is lighted, indicating that the system is ready for operation. When the power is increased, the processors are automatically cleared. Upon reaching standby status, the operator can cause the processors to begin execution by pressing the START button in the system control area of the panel. The system status indicator will then show that the system is in the operating condition. Similarly, the STOP button causes the processors to stop execution and the system status to return to the standby condition.

The normal/abnormal indicators in the system status area relate to switches throughout the system which must be in particular positions if the system is to operate in a normal fashion. Included are switches to provide maintenance functions throughout the system such as those for the power supply margining control and those on the maintenance panel. A digital clock is provided at the top center of the operating panel. The EMERGENCY OFF is a pull-type switch for emergency removal of primary power to the system. It requires maintenance personnel for resetting. In the lower left-hand portion of the panel, the bootstrap facilities are shown. The system may be loaded initially from disk or from the card reader by setting the system initialization switch to the desired initialization source and depressing the LOAD button. An alarm horn in the power control unit sounds in the event of power failure. The ALARM RESET button in the lower right-hand area of the operating panel allows the operator to turn off the alarm horn. A loudspeaker is provided within the operating panel for signaling from the operating system. The volume control is in the lower right-hand corner of the operating panel.

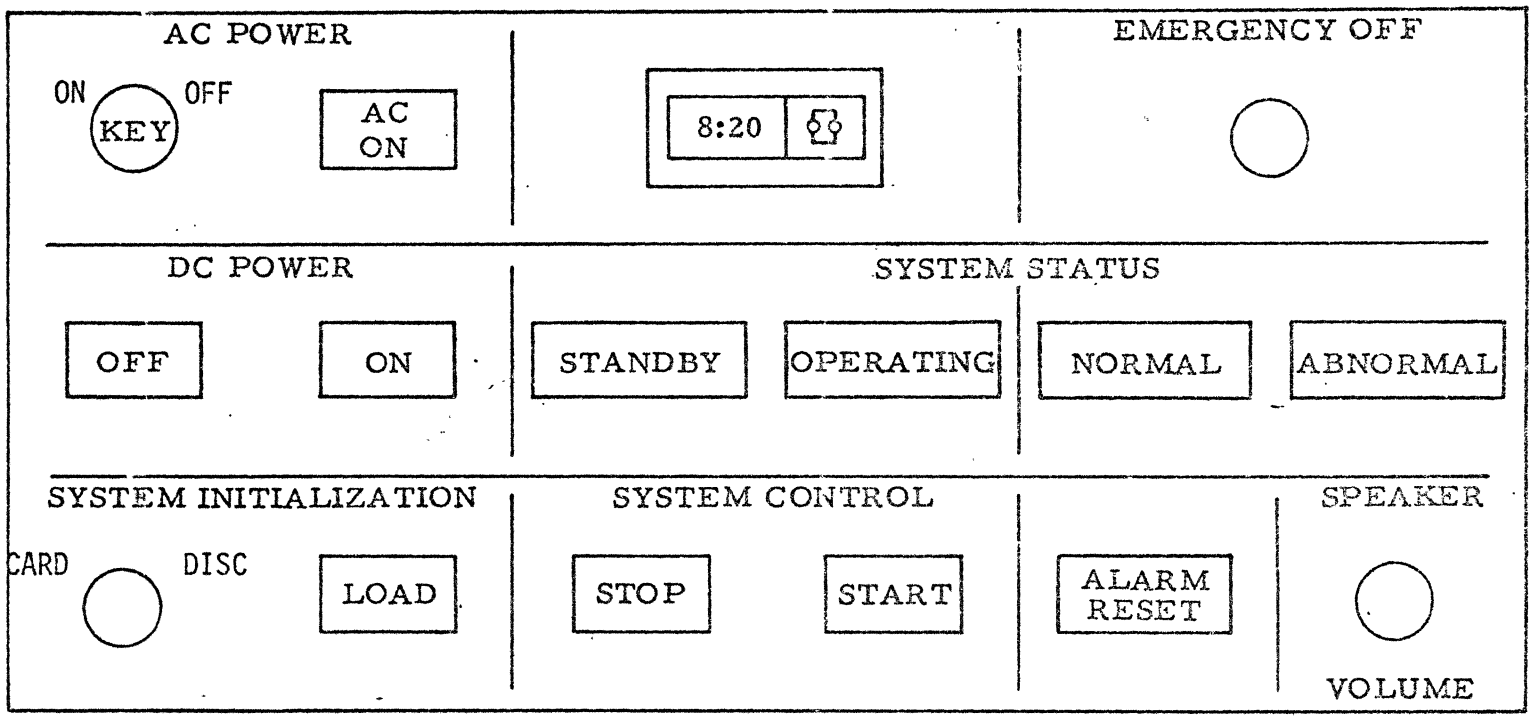


Figure 1[illegible]. ASC Operating Panel

The KEY switch applies primary AC power through the main three phase power relay to the peripheral equipment (tape and disc units, printer, card punch, card reader, etc.) and to the Power Control Unit (PCU). At this time either the NORMAL or ABNORMAL indicator is lighted depending on conditions. No other lamps are lighted. The DC ON indicator/switch is then pressed to apply DC power to the logic columns. The DC OFF indicator/switch lamp is turned off. Lighting of the DC ON lamp confirms DC power, but voltages, pressures and temperatures may not be within operating ranges. In this stage, the ASC cannot be turned off with the KEY: the operator must first press the DC OFF switch then turn the KEY off.

The STANDBY indicator is an environmental status monitor. The standby state and subsequent system operation is inhibited unless all monitored subsystems are within tolerance. The following signals indicated by a level or a pulse are required:

1) "go" from each of the four disc modules (discs up to speed)
2) "go" from thin film environmental control 1
3) "go" from thin film environmental control 2
4) System Standby from the PCU - a signal provided by PCU indicating that all environmental conditions monitored by PCU are within operational limits.

The Override Panel in the PCU cabinet provides a switch for each "overridable" function such as an inoperative disk module, to negate the Standby inhibit signal. The operator is notified of override operation by the display console, a bulletin board or some other type of status indicator. The Override Panel is accessible only to maintenance personnel.

The LOAD, START and STOP lamps are controlled by signals from the ECL columns. When the system reaches the standby state, the LOAD and START lamps will light. When either the LOAD or START switch is pressed, both of their lamps are extinguished when the system goes to the operate state and the STOP lamp is lighted. When either the STOP switch is pressed or the program is stopped through the display console, the system reverts to the standby state, the STOP lamp turns off, and the LOAD and START lamps are lighted.

After the prescribed warm up period, the STANDBY indicator is lighted. (The ASC must first reach STANDBY before operation is possible.) A program is started by one of two means: the LOAD indicator/switch or the START indicator/switch. The program is initiated from the card reader or disc sources when the LOAD switch and the CARD DISC switch are operated simultaneously. The CARD/DISC switch is a momentary on, three position, center off rotary switch. At the simultaneous activation of these two switches, VP_0 is programmed to select the information from the designated source and then start the run. The START indicator/switch is used to resume operation of the ASC from the program currently in Central Memory instead of reloading another program from card or disc. Whenever a program is running, the OPERATING indicator is lighted and the STANDBY indicator is turned off.

STOPPING DESCRIPTION

There are three levels of turn-off: soft shut down, "quasi" emergency off and emergency off. The first is an orderly turn off and is accomplished with the CRT display keyboard. When the operator types in a halt message the ASC goes through a routine of stop operations and is placed in the standby state.

The STOP switch is used for turn-off in case of, for example, an imminent power failure or when the display is inoperative. Pressing this switch causes the primary power failure type shut down procedure to be initiated. Status of the processors is saved in Central Memory, but data retrieval from tape or card is interrupted immediately in an unorderly fashion. To complete the turning off, the ASC, DC OFF must be pressed and then the KEY turned off. The DC OFF cannot be activated unless the system is in Standby.

Emergency Off is the same as pulling the plug. No consideration is given to saving system status. The system can only be restarted after "Emergency Off" by maintenance personnel.

There is a signal from the PCU called "Switches Normal" which indicates: 1) that the four "normal" switches on the Maintenance Panel are in the normal position, and 2) that the logic column power supply switches are in remote. If all these conditions are met, the PCU sends the Operator's Panel a signal lighting the NORMAL indicator. If the conditions are not met, the ABNORMAL indicator is lighted. The two conditions cited above will be in the PCU logic. In either the Normal or Abnormal state the computer can go to Standby.

ALARM SYSTEM

The ALARM RESET switch is a command switch to the PCU to turn off the alarm horn. The alarm warning is sounded when a power phase is lost, or when a high voltage, low voltage or cabinet fault is detected.

A second audio system is provided by a loud speaker under the table which gives an aural indication under control of the operating system. The volume control regulates only the sound intensity and cannot turn the sound completely off. Provision will also be made for a headphone jack under the front of the table for communication with operating personnel at other locations.

PERIPHERAL DEVICES

TABLE OF CONTENTS

DEVICE CONTROL

The peripheral devices of the ASC are controlled by a closely related combination of special purpose hardware and software. The programs which interface with the special purpose hardware are called device handlers. These handlers are executed in the PPU. The degree of device control provided by a device handler rather than special purpose hardware varies from device to device. For devices with faster data rates, special purpose hardware is required. The peripheral device, control hardware, and handler are treated as a package by the Programming System. Specific commands are described by the Programming System through use of a list of parameters in Central Memory. The required handler is then called into use in order to control the peripheral device in such a way as to satisfy the input/output requirement.

DEVICE CHARACTERISTICS

OPERATOR CRT DISPLAY CONSOLE

1. Display
   - Characters per line: 64
   - Lines per page: 32
   - Viewing Area: 7 1/2" X 9 1/2"
   - Character Repertoire: 64 ASCII Alphanumerics
   - Storage Method: Recirculating Delay Line

2. Keyboard
   - Alphanumerics: 64 Total
   - Cursor Control: Space, Carriage Return, Tab, Vertical Tab, Home
   - Edit Control: Replace, Insert, Delete, Clear
   - Mode Control: Send, Receive, Compose

CARD READER

Rate: 1500 cards/minute
Card Type: IBM 5081 or equivalent
Input Hopper: 2500 cards
Output Stackers 1 & 2: 2000 cards each
Read Mode: photoelectric, serially by column
Operating Controls: power, clear, hold

LINE PRINTER

Rate: 1000 lines/minute, line at a time
Characters: EBCDIC repertoire, open face Gothic style
Character Code: EBCDIC
Character Spacing: 10 characters/inch, 6 lines/inch
Paper Specifications: Standard, edge punched (1/2 inch hole centers) fanfold paper 4 1/2 to 19 inches wide will be used.
Operating Controls: Paper tension, paper position, phasing controls, skip feed (top of form), line feed, clear, power.

CARD PUNCH

Rate: 100 cards/minute
Card Type: IBM 5081 or equivalent
Input Hopper: 1000 cards
Output Stacker: 1000 cards
Punch Mode: serially by column
Verification: electro-mechanical, bit by bit comparison, mispunched cards offset
Operating Control: power, clear, hold

IBM COMPATIBLE TRANSPORT (1600 bpi)

Tape Size: width - $.500\ ^{+.000}_{-.004}$ inches
thickness - 1.5 mil, mylar base
Tape Handling: Single capstan, vacuum chamber buffering
Tape Speed: 112.5 ips ± 2%
Start/Stop Time: 3.5 ms max.
Start/Stop Distance: .168 ± .028 inches
Head Ass'y: read, write, erase, 9 track
Density: 1600 bpi
Rate: 45K words/sec.

POSITIONING ARM DISC

Capacity/Channel: 25×10^6 words
Channels per disc: 2
Total capacity: 50×10^6 words
Data Rate/channel: 2.0×10^6 words
Rotational latency: 34 ms maximum
Random Average Seek time: 55 ms
Maximum Seek time: 100 ms

HEAD/TRACK DISC

Capacity/Channel: 50,528,356 words
Total Capacity: 101,056,712 words
Data Rate/Channel: 500K words/sec
Rotational Latency: 34 ms maximum
Words/Sector: 64
Sectors/Revolution: 256 data sectors, 1 maintenance sector

INPUT/OUTPUT COMMANDS

OPERATOR CRT DISPLAY CONSOLE

1. Define Page: Clear screen, output page format, position cursor
2. Write Block: Update variable fields, position cursor
3. Read Block: Input operator response fields
4. Erase Block: Clear operator response or variable fields

CARD READER

1. Read: input specified number of columns, interpreting data as
   a. Symbolic Format Normal,
   b. Symbolic Format Control, or
   c. Object Module Format depending on the data in the first column read.

LINE PRINTER

1. Write Fixed Length: Output the specified amount of data, printing 132 characters per line.
2. Write Variable Length: Output the specified amount of data, printing on each line the number of characters specified at the beginning of each line image.

CARD PUNCH

1. Write Fixed Length: Output the specified amount of data, punching 80 columns per card.
2. Write Variable Length: Output the specified amount of data, punching on each card the number of columns specified at the beginning of each card image.

IBM COMPATIBLE TRANSPORT (1600 bpi)

1. Rewind & Unload: Rewind to BOT and revert to manual control.
2. Rewind: Rewind to BOT.
3. Backspace Files: Backspace the specified number of files.
4. Backspace Records: Backspace the specified number of records.
5. Skip Forward Files: Skip forward the specified number of files.
6. Skip Forward Records: Skip forward the specified number of records.
7. Read to File Mark: Read the remaining records in the current file, transferring alternate records to alternate ones of a pair of CM data buffer regions.
8. Read One Record, One Buffer: Read the next record, transferring all data into a single CM data buffer region.
9. Read One Record, Two Buffers: Read the next record, filling alternate ones of a pair of CM data buffer regions.
10. Write One Record, One Buffer: Write one record, transferring all data from a single CM data buffer region.
11. Write One Record, Two Buffers: Write one record, emptying data from alternate ones of a pair of CM data buffer regions.
12. Write Many Records: Write many records, acquiring alternate records from alternate ones of a pair of CM data buffer regions.
13. Write Tape Marks: Write the specified number of IBM compatible tape marks.
14. Erase: Erase the specified amount of tape.

HEAD/TRACK DISC

1. Read: Read the specified amount of data beginning at the specified word location on the disc, transferring data into a single CM data buffer region which may be defined by actual or virtual addressing.
2. Write: Write the specified amount of data beginning at the specified sector location on the disc, transferring data from a single CM data buffer region which may be defined by actual or virtual addressing. Partial sectors cannot be written, and when a nonintegral number of sectors is specified, the last portion of the sector is filled with zeros.
3. Erase: Write the specified number of zeros beginning at the specified sector location. An integral number of sectors is erased.
4. Read Check: Identical to "Read" except no data is transferred to CM. Parity is checked and results indicated to provide a quality check on previously recorded data.

MAINTENANCE

TABLE OF CONTENTS

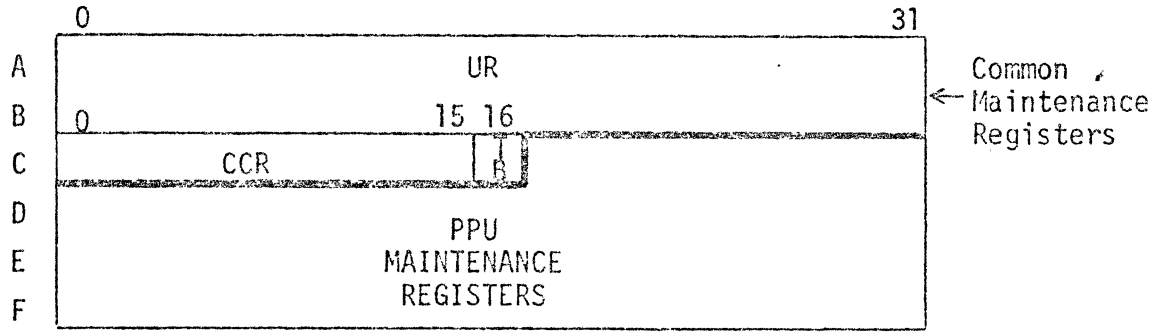
INTRODUCTION

Maintenance features in the ASC System are integrated with the operational logic. The data and control cells within each major unit can be interrogated and controlled from an external source with only minor disturbance to the state of the unit. The large number of cells involved prohibits the direct approach of providing input and output lines for each cell at the unit interface, so unit designs include a small maintenance controller which can select a subset of the unit's cells for external presentation or can force the subset to a state prescribed externally. The additional hardware required to provide this addressability is held to a minimum by sharing some selection trees between the operational logic and the maintenance logic. The result of this design approach is that a large portion of each unit can be exercised and checked from an external source providing only a small portion of the unit, the maintenance controller, is known to be operating properly. This small portion of the unit is referred to as the maintenance hardcore. The size and nature of the maintenance hardcore varies from unit to unit.

The external source used to stimulate the maintenance hardcore of the unit under test usually is one or more other units of the system. The first unit tested is the PPU, and this test is performed by use of specially designed external test equipment, and by self test features. When the PPU has been checked, it becomes the basic tool for stimulation of the maintenance hardcore of the other units. As each unit is tested, it becomes a tool to provide additional flexibility to tests for subsequent units.

The special external equipment required for testing the PPU consists of the PPU Maintenance Panel for manual test control, a card reader for semi-automatic test control and the Test Control Logic for interfacing the panel and the reader with the PPU maintenance hardcore. A port is provided for driving the Test Control Logic from a stored program processor to expedite checkout and maintenance operations. The Test Control Logic communicates with the hardcore via the PPU Maintenance Registers in the Communication

Registers shown below. These CR's provide extensive self test capability because the PPU maintenance hardcore can be stimulated by programs executed in the VP's.



Maintenance Registers of the PPU Communication Registers

After the PPU has been tested, it is used to stimulate the hardcore of the other units via the Common Maintenance Registers of the CR's. These registers include the Unit Registers (UR), the Common Command Register (CCR), and the Transfer Bit (TB). The CCR, presented to all units, contains commands addressed to a specific unit under test. The UR contains private unit registers required to augment the CCR commands. The TB controls the flow of information between the Common Maintenance Registers and the unit under test.

LOGIC CLOCKS

GENERAL

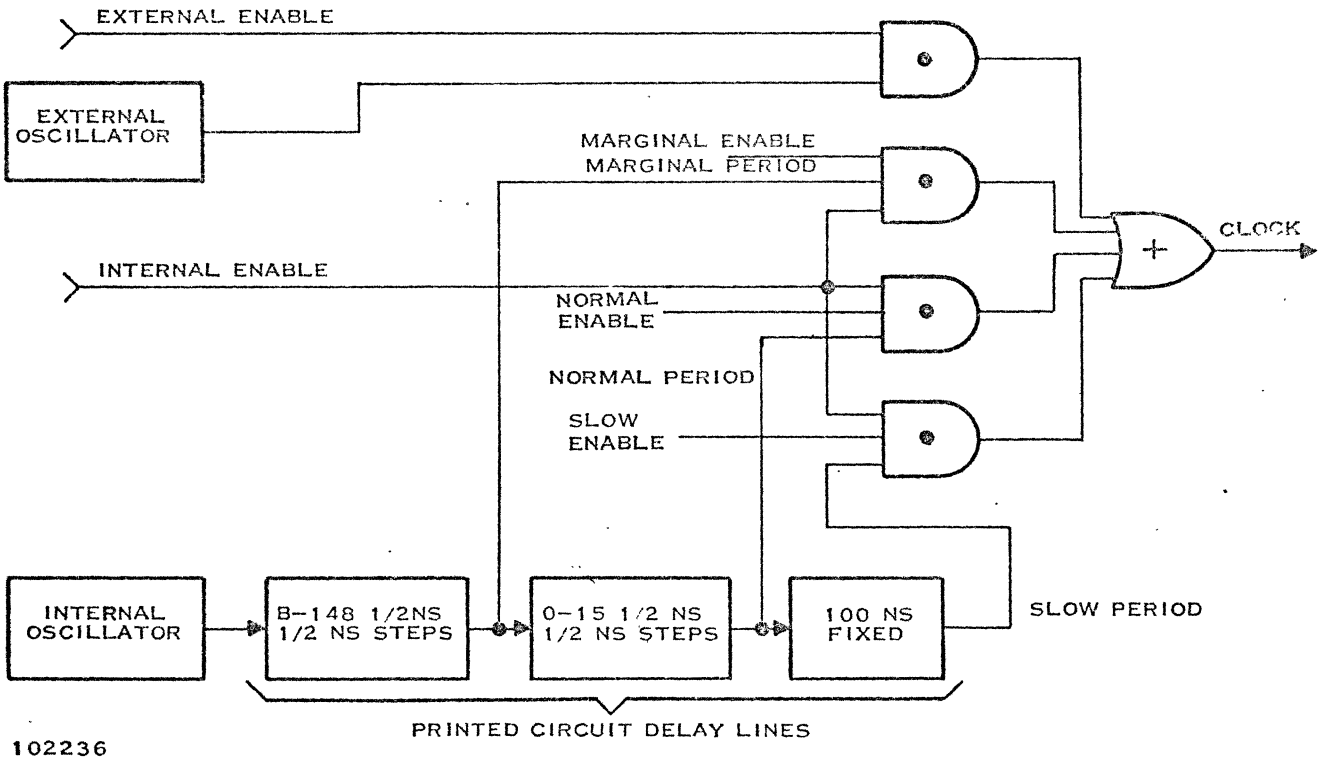
The logic clocks in the ASC system provides the syncronization pulses required by the system hardware and special maintenance capabilities. The six logic clocks are provided by Logic Clocks Module (LCM) circuit boards with the clock period adjusted to the specific location.

The maintenance feature of the LCM is the adjustability of the clock period to the three rates of normal, marginal (5% faster than normal), and slow (normal clock period + 100 nanoseconds) to facilitate checkout. In addition to these three speeds, an external signal generator may be used to

control the clock rate. The clock output of the LCM may be a continuous train of pulses or a burst of from 1 to 16 pulses as selected by the operator.

LOGIC CLOCK MODULE

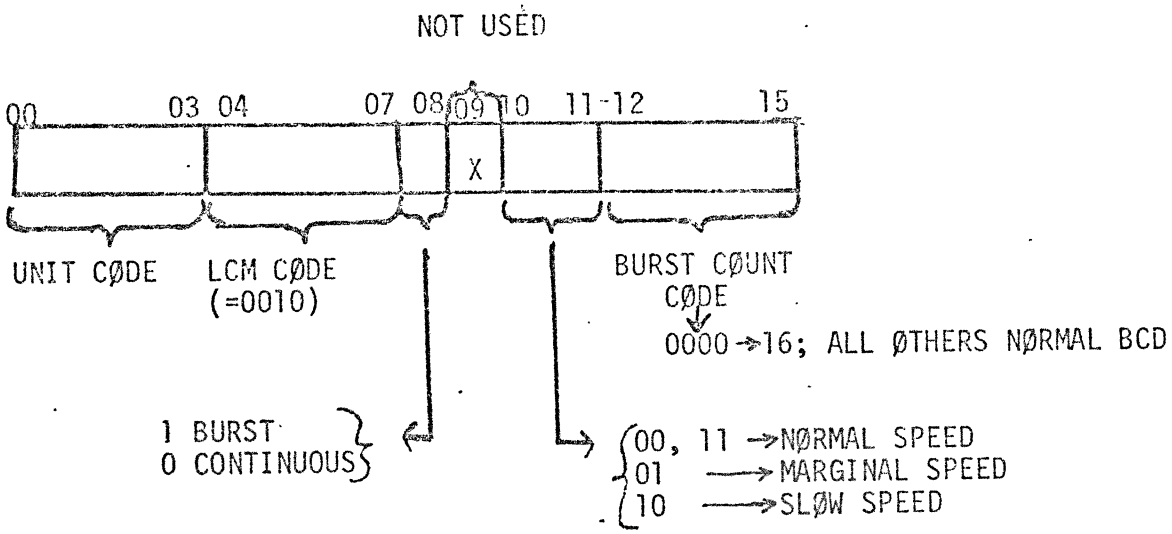
The LCM has a variable clock period. The block diagram illustrates the clock period selection. The printed circuit delays which are shown to be variable in 1/2 nanosecond steps are selected through the use of jumper wiring (in series) the desired set of delay lines. The three internally generated clock rates are called normal, marginal (5% faster than normal), and slow (normal clock period + 100 nanoseconds).



Logic Clock Module Block Diagram

The CP and DCU control commands for a particular LCM come from the Common Command Register (CCR); of the 16 bits in the CCR, only 15 are used by the LCM. When the CCR bits 0-3 match the four unit ID bits which are wired to each unit, and CCR bits 4-7 are 0, 0, 1, 0, respectively, and a Transfer Bit (TB) is received and recognized, the LCM will load and

execute the request contained in CCR bits 8-15. It should be noted here that CCR bit 12 is the msb and CCR bit 15 is the lsb for the burst count code; also, a "stop" command would be a burst of arbitrary length. The control commands for the PPU is from the Start-Up and Audio byte of the CR file.



Other inputs to the LCM are as follows:

1) An initializing or system stand-by signal, present during power turn-on, to put the LCM in its initial state which is no output, ready to accept its first command.

2) The external signal generator output, i.e., the external source.

3) A signal from a switch, → EXTR, is used to select the external signal generator as the clock source. If this switch is moved to a new position while a command is being executed, the LCM will (a) complete a burst, or (b) stop immediately upon detection of the change if running continuously, and wait until the next LCM command to start again. When operation is resumed, the source will be determined by the current switch position and CCR bits 10 and 11, with the switch being the dominant signal.

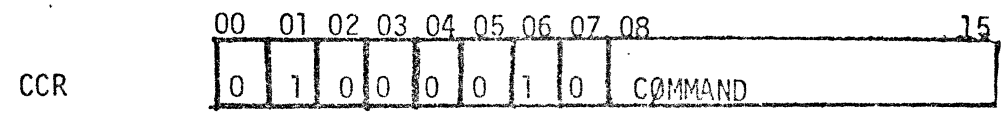
The LCM clock signal distribution network provides for driving up to 11 separate lines and a single reply signal, CACKCMD. The clock signal will be normally false and will be free of spikes and "crippled (partial)" pulses; when running, the clock signal has a 50% duty cycle. The reply line is set true for one clock period duration (then reset) at the beginning of a continuous

command or at the end of a burst, and is used to reset TB for all applications except for the PPU.

CENTRAL PROCESSOR INTERCONNECTIONS

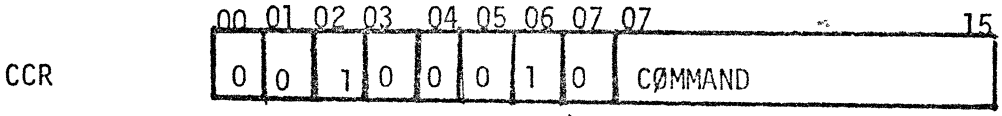
The Central Processor (CP) contains four cards, a master and three "slave" boards. The MBU, AU, and IPU each contains one of the subserviant LCM's. During maintenance and checkout procedures these three modules may be operated independently (on a unit level). During normal operation these units are set to the external, continuously running mode acting only as clock fan-out and distribution networks for the master LCM.

All four modules recognize the following CCR command format:



DATA CHANNEL UNIT INTERCONNECTIONS

The DCU clock will recognize the following format:



PERIPHERAL PROCESSOR INTERCONNECTIONS

The PP's LCM is controlled by the Maintenance Control Panel (MCP) and two CR bits, with the MCP signals being dominant.

The "CLOCK RATE" switch on the MCP is used to select normal, marginal, slow, or external clock speeds and clock source; it may also select a step mode (burst of one pulse). The absence of any of these signals in the true state is interpreted as being one of the "OFF" positions of the switch

The only program control available in the PP is effected through the use of the two CR bits which specify the selected period of the internally generated clock signal. These two bits are located in the Start-Up and Audio byte. The code for these two bits is as follows:

BITS:	4	5	
00 or	1	1	NORMAL
	0	1	MARGINAL
	1	0	SLOW

A restriction on this operation is that these two bits have clock rate control only when the MCP "CLOCK RATE" switch is in the normal position.

The LCM in the PP will stop anytime the MCP "CLOCK RATE" switch is turned, and must be restarted manually be depression of the "GENERATOR INIT" button on the MCP.

PPU MAINTENANCE REGISTERS

GENERAL

The PPU Maintenance Registers provide the communication link between the PPU maintenance hardcore and the equipment controlling PPU tests. The test equipment loads these registers with commands and addresses, and observes results in these registers. The commands which appear in the PPU Maintenance Registers are decoded and executed by the Maintenance Logic which is the major part of the maintenance hardcore.

The Maintenance Registers are loaded with commands from the Maintenance Panel (manual mode), the card reader (semi-automatic mode), a small stored program computer, or from an operative VP within the PPU (automatic mode). The source of the commands is dependent on the test mode which is controlled by a Maintenance Panel switch. The reaction of the Maintenance Logic to the commands is essentially identical for all command sources. The relationship of the PPU test facilities are shown in Figure 1.

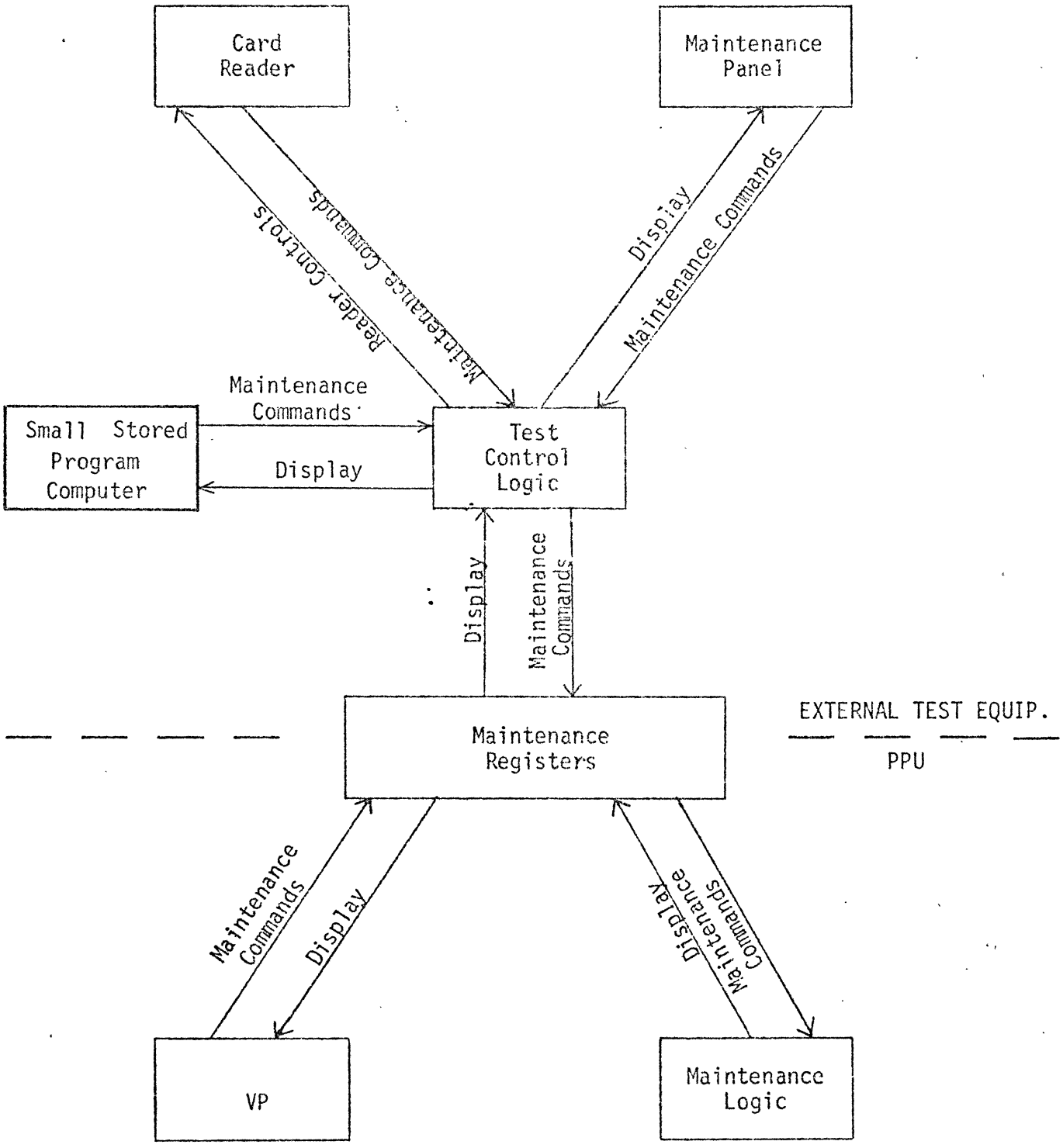
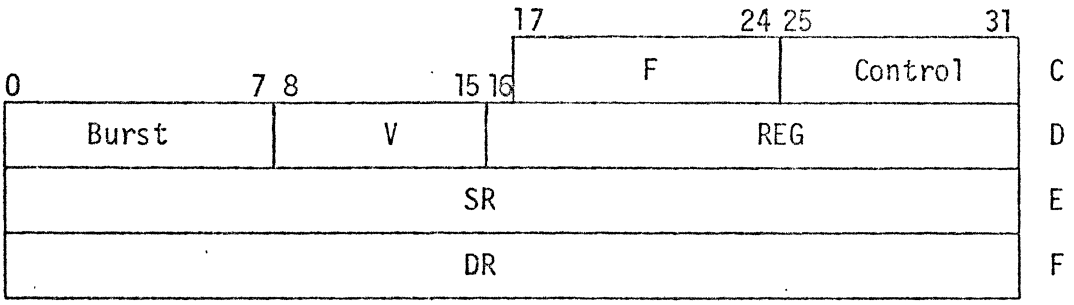


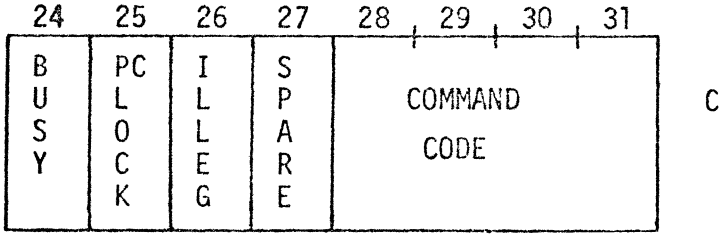
Figure 1. PPU Maintenance Hardware

DESCRIPTIONS

The Maintenance Registers are divided into seven fields as indicated below. The Control field contains the basic command to be executed by the Maintenance Logic, and the remaining fields contain addresses and parameters referenced by the commands or data resulting from command execution.



CONTROL FIELD



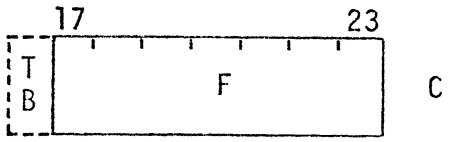
Busy - Set by Maintenance Logic on the bit period immediately following receipt of any non-zero command code in bits 28-31. Reset by Maintenance Logic on the bit period immediately following completion of the command. Resetting does not occur if the command is illegal.

PC Lock - Set or reset by the Maintenance Logic simultaneous with resetting of Busy bit. Indicates current status of program counter lock which is affected by a command code of 8 or 9 in bits 28-31. "1" denotes locked, "0" denotes normal.

Illegal - Set by Maintenance Logic if an illegal command is received.

Command Code - Set by the Test Control Logic or by a VP to initiate a maintenance operation. Reacted to and reset by the Maintenance Logic. Resetting occurs simultaneous with the resetting of the busy bit. Maintenance Logic reaction to commands varies slightly depending on whether the Maintenance Panel indicates manual, semi-automatic, or automatic test mode. Reactions to the 16 command codes are given in Table I.

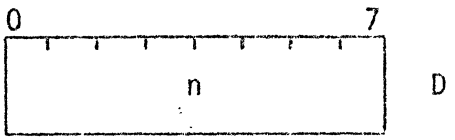
F FIELD



The F field specifies the portion of a VPR or CR to be affected by command codes 5 and 6. This field is not modified by the Maintenance Logic.

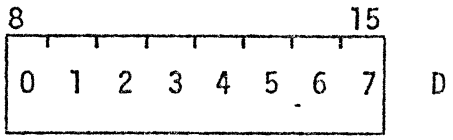
F	Register portion
XXXX 000	byte 0
XXXX 001	byte 1
XXXX 010	byte 2
XXXX 011	byte 3
XXXX 100	left half
XXXX 101	right half
XXXX 110	whole word
XXXX 111	

BURST FIELD



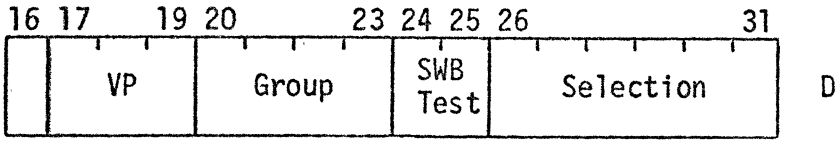
The Burst field specifies a number used in conjunction with command codes C and E. This field is never modified by the Maintenance Logic.

V FIELD



The V field specifies which VP's are under test as required for command codes C, D, E, and F. In the manual test mode, only one of the bits will be "1". This field is never modified by the Maintenance Logic.

REG FIELD



The VP, Group, and Selection portions of the REG field designate a register which is to be involved in the data transfer specified by command codes 2, 5, and 6.

VP	Group	Selection	Register Designated
n	0000	-	PC in VP_n
n	X001	-	NIR in VP_n
n	X010	i	IR in VP_n
n	X011	-	SWBA in VP_n
n	X100	-	SWBD in VP_n
n	X101	i	VPR_i in VP_n
-	X110	i	CR_i
-	X111	i	$SWBC_i$ } illegal for automatic mode
-	1000	i	MIR_i }

If Group and Selection designate $SWBC_8$, $SWBC_9$, $SWBC_A$ in conjunction with command code 5, then the command has a special meaning. These codes are used for controlling the asynchronous portions of the SWB as follows:

$SWBC_8$ - SWB stops on even state

$SWBC_9$ - SWB stops on odd state

$SWBC_A$ - SWB normal

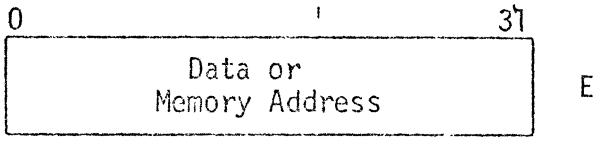
The Maintenance Logic maintains these control conditions until a change is commanded.

The VP portion of the REG field is used to specify a VP# in conjunction with command codes 3, 4, and 7.

The SWB Test portion of the REG field is used for placing the SWB under test, but only during the Semi-automatic mode. Whether or not the SWB has been placed under test affects the interpretation of command code C. The use of the SWB Test bits is independent of commands. The Maintenance Logic responds immediately to these bits as follows:

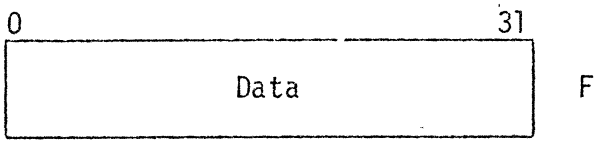
bit 24	bit 25	reaction
0	0	none
0	1	SWB placed in normal condition
1	0	SWB placed under test
1	1	none

SR FIELD



The SR field provides data for command codes 1 and 5, and provides a memory address for command codes 3, 4, and 7. This field is never modified by the Maintenance Logic.

DR FIELD



The DR field provides data to the Maintenance Logic for command codes 6 and 7. For command codes 1, 2, 3, and 4, the DR field is loaded by the Maintenance Logic.

Table 1. Command Codes

CODE	DESCRIPTION
0	This is the state of the command code at the completion of any command unless the command is illegal or cannot be completed. When such commands occur, the Command Code must be set to 0 by the tester in order to clear the Maintenance Logic. When the code is set to zero by the tester, the Maintenance Logic resets the Busy, PC Lock, and Illegal bits, and is ready to accept new commands.
1	(SR)→ DR
2	((REG))→ DR For automatic mode, illegal if REG specifies SWBC or MIR.

Table 1. Command Codes (Continued)

CODE	DESCRIPTION
3	$((SR))_{CM} \rightarrow DR$ The portion of the SWB designated by bits 17-19 of the REG field is used for this command.
4	$((SR))_{ROM} \rightarrow DR$ An NIR and part of the MIR are required. The NIR designated by bits 17-19 of the REG field is used.
5	$(SR) \rightarrow (REG)$ If REG specifies VPR or CR, then the portion of the register affected is controlled by the F field. For automatic mode, illegal if REG specifies SWBC or MIR. If REG specifies $SWBC_8$, $SWBC_9$, or $SWBC_A$, then command code 5 is interpreted differently, and controls the asynchronous portions of the SWB.
6	$(DR) \rightarrow (REG)$ If REG specifies VPR or CR, then the portion of the register affected is controlled by the F field. For automatic mode, illegal if REG specifies SWBC or MIR.
7	$(DR) \rightarrow (SR)_{CM}$ The portion of the SWB designated by bits 17-19 of the REG field is used for this command.
8	Lock PC's of all VP's designated by V field.
9	Unlock PC's of all VP's designated by V field.
A	Reset (asynchronous reset line) all F/F's unique to all VP's designated by V field.
B	Set (asynchronous set line) all F/F's unique to all VP's designated by V field.

Table 1. Command Codes (Continued)

CODE	DESCRIPTION
C	Automatic Mode - Advance all VP's designated by V field. Advancement occurs under the influence of the time slot table. Amount of advancement is determined by counting the number of time slots indicated by the Burst field. Counting begins at T. S. O. Semi-Automatic Mode - Reaction dependent on whether or not SWB has previously been placed under test as described under Register Field 1) SWB not under test - Advance the PPU the number of time slots indicated by the Burst field. This is equivalent to normal operation within the PPU except that it occurs for a limited time. 2) SWB under test - Advance the SWB the number of time slots indicated by the Burst field. Manual Mode - Reaction dependent on whether or not SWB has been placed under test by activation of the SWB LOCK switch on the Maintenance Panel. For each case, reaction is identical to respective Semi-Automatic case.
D	Automatic Mode - Advance all VP's designated by V field. Advancement occurs under the influence of the time slot table. Advancement of each VP continues until that VP completes its current instruction. Note that if more than one VP is designated by the V field, then termination of advancement of the designated VP's does not necessarily occur at the same time.

Table 1. Command Codes (Continued)

CODE	DESCRIPTION
	Semi-Automatic Mode - Start time slot counter. Proceed as for Automatic Mode. Stop time slot counter at time slot 0. Manual Mode - Advance the VP (only one) designated by the V field. Advancement occurs independent of the time slot table. Continue until the designated VP completes its current instruction.
E	Automatic Mode - illegal command Semi-automatic Mode - illegal command Manual Mode - Advance the VP (only one) designated by the V field. Advancement occurs independent of the time slot table. Amount of advancement is determined by the Burst field.
F	Automatic Mode - illegal command Semi-Automatic Mode - illegal command Manual Mode - Start advancement of the VP (only one) designated by the V field. Advancement occurs independent of the time slot table. Advancement continues after "completion" of the command by the Maintenance Logic. Advancement stops when any new command code is received including a second "F" command. If the second command is other than "F" then in addition to discontinuing advancement of the designated VP, perform the new command.

MAINTENANCE MODES

AUTOMATIC MODE

In the Automatic Mode a VP loads the Maintenance Registers through its normal addressing capability. A VP operating normally can place other VP's under test, control the advancement of the VP's under test, and interrogate and/or modify the results of the advancement.

SEMI-AUTOMATIC MODE

In the Semi-Automatic Mode, a card reader or small computer controls the advancement of the VP's under test, and can interrogate and/or modify the results of the advancement.

1) Card Reader Control

The card reader interprets the cards column by column and passes card commands to the Test Control Logic. A card command is a sequence of two or more of the hexadecimal digits 0 - F, terminated by a blank column or end of card. These digits are represented by the alphanumeric characters 0 - 9 (Single 0 to 9 punch) and A - F (a 12 punch and a single 1 to 6 punch).

A card column containing an eleven punch (alone or with other punches) is ignored. This allows overpunching errors with a minus sign without having to punch another card.

An illegal card command is treated the same as an end of card. An illegal command is defined to be:

1. A single digit field.
2. A field starting with a digit other than 0 to 6.
3. A character other than 0 - F, blank, or containing an 11 punch.

The card command size will be two, three, or five digits depending on the command code as shown in Table 2. If the card field has too

Table 2. Card Reader Commands

Card Code	Mnemonics		Test Control Logic Interpretation
OX	00	NOP	Load X into the least significant hex of the Control field in the CR file. This action issues a command to the Maintenance Logic. The Test Control Logic interprets the Busy bit (bit 0 of the CR located Control Field) to determine when to proceed. If X is "E" or "F", the Test Control Logic also does the following:
	01	XSD	
	02	XRD	
	03	XCD	
	04	XMD	
	05	XSR	
	06	XDR	
	07	XDC	
	08	LPC	"OE" - If (SR) = (DR), continue test. If (SR) ≠ (DR), stop test with Compare Error light on. Ignore remainder of card.
	09	UPC	
	0A	RST	
	0B	SET	"OF" - Stop Test. Ignore remainder of card.
	0C	CLK	
	0D	CYC	
	0E	CMP	
	0F	HLT	
1XXXX		LDL	Load XXXX into the most significant half of the SR field in the CR file.
2XXXX		LDR	Load XXXX into the least significant half of the SR field in the CR file.
3XXXX		LRG	Load XXXX into the REG field in the CR file.
40X		LRF	Load 0X into the F field in the CR file.
5XX		LVT	Load XX into the V field in the CR file.
6XX		LCB	Load XX into the Burst field in the CR file.

many digits, the extra digits will be ignored. If the card field has at least two digits but less than the required number, the action will take place leaving the hex digits in the register that correspond to the missing digits undisturbed.

2) Programmed Processor Control

A stored program processor may be connected to the Test Control Logic to expedite checkout and maintenance activities. Card commands are generated and the DR output is sampled by the processor to provide extended maintenance capabilities:

1) Maintenance command sequence looping to facilitate signal timing analysis (scope loops).
2) Printing contents of all or of selected registers as the PPU executes a series of instructions (microtrace).
3) Interactive hard copy display and modification of PPU register and CM location contents.

When the processor is connected to the Test Control Logic the card reader input is selectively disabled under control of the processor. When the processor is not connected the card reader input is enabled. The processor is detachable test equipment and is not an integral part of the PPU.

MANUAL MODE

In the Manual Mode the Maintenance Registers are loaded from the Maintenance Panel to control the advancement of the VP's and interrogate and/or modify results of the advancement. Manipulations of the Maintenance Panel are interpreted by the Test Control Logic, appropriately encoded, and loaded into the PPU Maintenance Registers.

The Maintenance Panel (Figure 2) includes some controls for purposes other than loading the Maintenance Registers. All functions of the panel are given in Table 3.

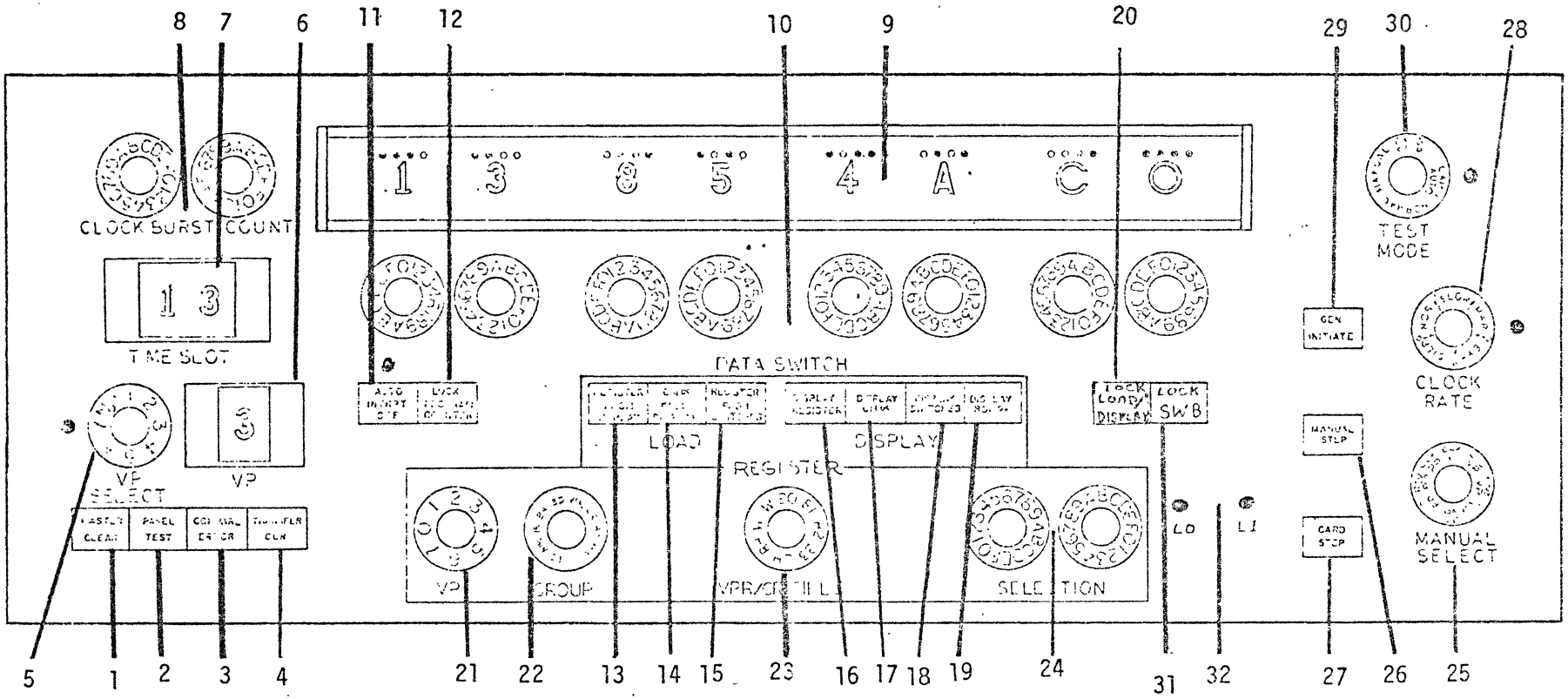


Figure 2. PPU Maintenance Panel

Table 3. PPU Maintenance Panel Controls and Indicators

No.	Control/Indicator	Function
1.	MASTER CLEAR Push button/Indicator	Enabled and lit only if the TEST MODE switch (30) is not in the NORMAL position. Switch forces all the cells in the PPU and in the interface controls of all of the I/O devices interfacing with the PPU to "0". This occurs via the asynchronous reset line provided for the cells, and is independent of the PPU clock system.
2.	PANEL TEST Push button/Indicator	All the panel lamps are illuminated.
3.	COMPARE ERROR Indicator	Illuminates if (SR) $\neq$ (DR).
4.	TRANSFER CCR	Enabled and lit in Manual Test Mode. Sets the TB bit in the Common Maintenance Registers to initiate a transfer of the CCR to a unit other than the PPU.
5.	VP SELECT Switch	If TEST MODE (30) = NOR: designates the selected VP for time slot zero. If TEST MODE (30) = MANUAL: designates the VP under test. If TEST MODE (30) = CARD STEP or CARD AUTO: inactive. The adjacent light extinguishes if the switch is in the NOR position.

Table 3. PPU Maintenance Panel Controls and Indicators (Continued)

No.	Control/Indicator	Function
6.	VP Display	Displays the current VP if; 1. the CLOCK RATE (28) = STEP or 2. the TEST MODE (30) = MAN and the MANUAL SELECT 25 = PP CLK, PP REV, or PP BST. Current VP is the VP currently at the execution level (MIR) in the PPU. This is the VP which will react to the next clock pulse and which corresponds to the time slot displayed at TIME SLOT (7).
7.	TIME SLOT Display	Displays a decimal readout of the current time slot if; 1. the CLOCK RATE (28) = STEP or 2. the TEST MODE (30) = MAN and the MANUAL SELECT (25) is in the PP CLK, PP REV, or PP BST. Time slot is the time slot which selected the displayed VP (6). Due to time slot lookahead, this is not the time slot controlling VP selection at the IR level or at the time slot table scan level.
8.	CLOCK BURST COUNT Switches	Provides input data to the Burst field when the MANUAL STEP (26) is depressed if the MANUAL SELECT (25) is in the PP BST or VP BST position.
9.	Data Display Display	Provides a hexadecimal readout of the DR. The small lights adjacent to the hex character displays provide a binary readout of the DR.

Table 3. PPU Maintenance Panel Controls and Indicators (Continued)

No.	Control/Indicator	Function
10.	Data Switch Switches	Provide input data to the SR. The value on the switches is inserted into the SR when any of the LOAD DISPLAY switches (13 thru 19) are activated.
11.	AUTO INTERRUPT OFF Push button/Indicator (2 color)	The selected VP does not receive the auto interrupt signal but its time slot is overriden the same as the other VPs. If the automatic interrupt logic is disabled (abnormal), then the indicator is illuminated and the light adjacent to the switch it illuminated.
12.	LOCK PROGRAM COUNTER Push button/Indicator (2 color)	Enabled in the Manual Test Mode. If illuminated, the program counter of the VP under test is locked. If not illuminated, no program counters are locked. Switch causes the following if Busy is "0"; 1. VP (21)→ V field, and 2. Command Code "8" → control field if indicator is off, or Command code "9" → control field if indicator is on.
13.	REGISTER FROM DISPLAY Push button/Indicator (2 color)	Enabled in Manual Test Mode unless LOCK LOAD/DISPLAY (20) is activated. Switch causes the following if the Busy bit is "0"; 1. REGISTER (21, 24)→ REG field, 2. VPR/CR (23)→ F field, and 3. Command code "6"→ Control field.

Table 3. PPU Maintenance Panel Controls and Indicators (Continued)

No.	Control/Indicator	Function
14.	CMα FROM DISPLAY Push button/Indicator (2 color)	Enabled in Manual Test Mode if both REGISTER VP (21) and VP indicated by (5) indicate a different VP. The switch causes the following if Busy is "0"; 1. DATA SWITCHES→ SR, 2. REGISTER →REG field, 3. Command Code "7"→ Control field.
15.	REGISTER FROM SWITCHES Switch/Indicator (2 color)	Enabled in Manual Test Mode. The switch causes the following if the Busy bit is "0"; 1. REGISTER→ REG field, 2. VPR/CR FIELD (23)→ F field, 3. DATA SWITCHES (10)→ SR, and 4. Command Code "5"→ Control field.
16.	DISPLAY REGISTER Push button/Indicator (2 color)	Enabled in Manual Test Mode. The switch causes the following if the Busy bit is "0"; 1. REGISTER→ REG field, 2. Command Code "2"→ Control field.
17.	DISPLAY CMα Push button/Indicator (2 color)	Enabled in Manual Test Mode if REGISTER VP (21) and VP SELECT (5) indicate a different VP. The switch causes the following if Busy bit is "0"; 1. DATA SWITCHES (10) → SR, 2. REGISTER→REG field, 3. Command Code "3"→ Control field.
18.	DISPLAY SWITCHES Push button/Indicator (2 color)	Enabled in the Manual Test Mode. The switch causes the following if Busy bit is "0"; 1. DATA SWITCHES (10)→ SR, 2. Command Code "1"→ Control field.

Table 3. PPU Maintenance Panel Controls and Indicators (Continued)

No.	Control/Indicator	Function
19.	DISPLAY ROMα Push button/Indicator (2 color)	Enabled in the Manual Test Mode. The switch causes the following if the Busy bit is "0"; 1. DATA SWITCHES (10) → SR, 2. REGISTER → REG field, 3. Command Code "4" → Control field.
20.	LOCK LOAD/DISPLAY Push button/Indicator (2 color)	Used in the Manual Test Mode to lock any of the LOAD/DISPLAY switches (13 thru 19) so that the function normally initiated by the locked switch occurs continuously. LOCK switch will lock the function switch simultaneously depressed if the function switch is enabled, and both indicators will illuminate; the other functions will be disabled and be extinguished. The locking is removed by 1. LOCK LOAD/DISPLAY switch, 2. changing any switch condition normally required for initiation of the locked function. If, while a LOAD/DISPLAY function is locked, another function, not subject to locking, requires the use of the Control field, then the locking logic temporarily relinquishes use of the Control field. When the second function is complete (as indicated by the Busy bit) then the locked function continues.
21.	REGISTER VP Switch	Provide input data to the REG field when one of the LOAD/DISPLAY switches is activated.
22.	REGISTER GROUP Switch	

Table 3. PPU Maintenance Panel Controls and Indicators (Continued)

No.	Control/Indicator	Function
23.	REGISTER VPR/CR FIELD Switch	Provides input data to the F field when one of the following LOAD DISPLAY switches is enabled; 1. REGISTER FROM DISPLAY (13), or 2. REGISTER FROM SWITCHES (15).
24.	REGISTER SELECTION Switch	Provides input data to the REG field when one of the LOAD/DISPLAY switches is activated.
25.	MANUAL SELECT Switch	Provides input data to Maintenance Registers as described under MANUAL STEP (26).
26.	MANUAL STEP Push button/Indicator	Enabled and lit in Manual Test Mode. Switch causes the following if the Busy bit is "0": a. CLOCK BURST COUNT (8) → Burst field if MANUAL SELECT (25) is in PP BST or VP CLK. b. BP SELECT (5) → V field if MANUAL SELECT (25) is in VP CLK, VP CYC, VP CNT, VP RST, or VP SET. c. Command code "C" → Control field if MANUAL SELECT (25) is in PP CLK, PP REV, or PP BST. d. Command code "D" → Control field if MANUAL SELECT (25) is in VP CYC. e. Command code "E" → Control field if MANUAL SELECT (25) is in VP CLK.

Table 3. PPU Maintenance Panel Controls and Indicators (Continued)

No.	Control/Indicator	Function
		f. Command code "F" → Control field if MANUAL SELECT (25) is in VP CNT. g. Command code "A" → Control field if MANUAL SELECT (25) is in VP BST. h. Command code "B" → Control field if MANUAL SELECT (25) is in VP SET. i. Binary 16 → Burst field if MANUAL SELECT (25) is in PP REV. j. Binary 1 → Burst field if MANUAL SELECT (25) is in PP CLK.
27.	CARD STEP Push button/Indicator	If TEST MODE (30) switch is in CARD STEP, the CARD STEP button/indicator is illuminated and enabled. Depressing the button initiates the sequence of tests punched on one card. If the card includes a command for a comparison, and if the comparison fails, then the remainder of the card is ignored by the Test Control Logic and COMPARE ERROR (3) is illuminated. If the TEST MODE (30) switch is in CARD AUTO, the CARD STEP button/indicator is initially illuminated and enabled. Depressing the button initiates the sequence of tests punched on the card deck. Card tests continue with the CARD STEP button/indicator extinguished and disabled. The tests continue until a comparison

Table 3. PPU Maintenance Panel Controls and Indicators (Continued)

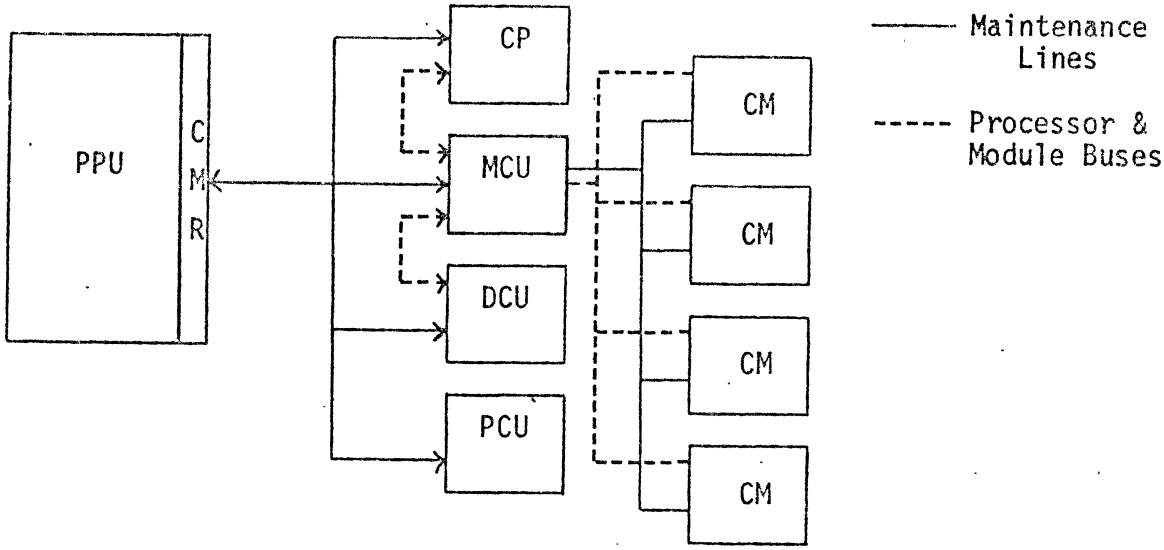
No.	Control/Indicator	Function
		command fails. When this occurs, the remainder of the card is ignored by the Test Control Logic, no additional cards are read by the reader, the CARD STEP button/indicator is illuminated and enabled, and COMPARE ERROR is illuminated.
28.	CLOCK RATE Switch	Controls clock source to the PPU as follows: a. NOR - Normal frequency b. MAR - Marginal frequency (normal + 5%) c. EXT - Clock source provided by external generator d. STEP - Clock source under control of MANUAL STEP (26) e. SLOW - Normal period + 100 ns The light adjacent to the switch is lit if the switch is not in the NOR position.
29.	GEN INITIATE Push button/Indicator	Enabled and lit under the following circumstances: a. CLOCK RATE switch in "NOR", "MAR", or "SLOW" position and clock source stopped. b. CLOCK RATE switch in "STEP" position.
30.	TEST MODE Switch	Activates the applicable maintenance equipment in Manual Test Mode (MAN), Semi-Automatic Test Mode (CARD STEP or CARD AUTO), or Normal Operating Mode or (Automatic Test Mode)

Table 3. PPU Maintenance Panel Controls and Indicators (Continued)

No.	Control/Indicator	Function
		(NOR). The light adjacent to the switch is illuminated when the switch is not in NOR.
31.	LOCK SWB Push button/Indicator	Enabled in Manual Test Mode. Places SWB under test or removes SWB from under test depending on present state of switch. Illuminated if SWB is under test.
32.	L0, L1	L0 - Illuminated if Busy bit is zero. L1 - Illuminated if an illegal maintenance command is sensed.

GENERAL

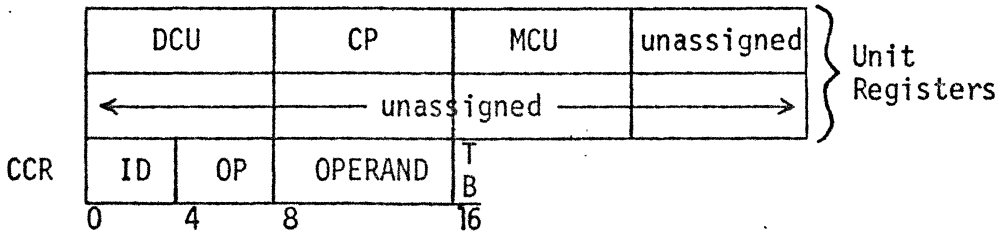
Maintenance is performed on the CP, MCU, and DCU using a program operating in the PPU which controls the designated unit via the Common Maintenance Register (CMR) as shown below.



The CMR commands action to be taken, transfer of data directly to or from the unit addressed, or between Central Memory and the addressed unit, using dedicated pointers contained in low order CM. The appropriate status registers, section control cells, decode circuitry, and hard core circuits are contained in each of the addressed units. These circuits are tailored to the needs of the unit in which they reside and interface in a standard way with the CMR.

This concept allows for system expansion and flexibility without having to reserve large blocks of CRs in the PPU for maintenance use. CM transfers to and from the addressed unit uses the unit's normal bus and addressing mechanism to pick up the pointer and request the required transfer of data.

The layout of the CMR in the CR byte is shown below.



Other CMR's may be used for operational communications where speed of response, interaction between the PPU and the other unit, or grouping of the CR for monitor response is dictated by the system requirements.

The communications between the MCU and the other unit is the normal address channel. If the operand specified by the CCR is in CM then the unit generates an address and requests the data from the MCU. Normally the data accessed is a specific word in the lowest portion of CM. This word in turn points to a data field where the actual data is located. The unit must provide for suppression of Map and Protect while the pointer and data are being accessed.

The maintenance panel may load CCR and the appropriate UR using a half word or byte load of the appropriate CR from the switch or display register. The corresponding words can be observed by transferring the appropriate CR to the Display Register. The "transfer CCR" push button switch is provided on the panel to set TB. TB is located at the 0th position of the 'F' byte in the Maintenance panel CR and may be set by the maintenance card reader by a LRF $8X_{16}$ instruction where X is the value of F that is desired in the right hex of the byte.

CCR is contained in the left half CR word that contains the F field and control field bytes. Since a zero value of these fields results in no action to the PPU, a whole word may be loaded into this CR by instruction that contains the desired command in the left half word and a 1 in bit sixteen and zeroes in bits 17-31. This causes the desired action to be set in CCR and TB triggered.

Signals which are contained in CCR code and which are also need for hardware action such as context switching are routed to the unit directly where they are "OR'ed" with the decoded value of the CCR for the same action. This prevents coding the signal or special decoding at the PPU interface.

DESCRIPTIONS

UNIT REGISTER

The eight one byte registers are used to transfer data from the addressed unit to a CR file and to load data directly when required. The loading of UR in the PPU is via software. Command Codes and supplementary addresses in CCR cause the transfer to or from the unit addressed and proper routing within the unit.

CCR

Common Command Register is shared by all the other units for maintenance control and infrequent communications. The register is divided into three sections.

ID(0-3) This field designates the unit that is to interpret the command and perform the necessary actions. Unit ID assignments are:

Bits 0-3

1 - MCU

2 - DCU

4 - CPU

OP(4-7) This is an operation code that is unique to the unit addressed by ID. The op codes are used to:

a) Direct loading or unloading of registers, control, or communication cells in the addressed unit from or to Central Memory. A dedicated communication word in CM contains a pointer to the CM region to be used.

b) Transfer a byte of data to or from the UR register in the CR file. This allows the unit addressed to be controlled or accessed by the maintenance panel, card reader, or small computer.

c) Initiate a control action based on the contents of the CCR.

d) Load communication or control cells directly from the CCR.

ADDR(8-15) The operand field

a) addresses specific registers or cells of the unit corresponding to ID.

b) contains immediate operand for the addressed unit.

c) contains supplementary Operation command.

d) a combination of a, b, c above.

TB

Transfer bit. When this bit is set to one, the unit addressed by the ID field takes the action indicated by the balance of the instruction in CCR. Two kinds of response result:

a) When the action is completed, TB is reset to zero by the addressed unit.

b) When the command is received, TB is reset to zero by the addressed unit. When the action is completed, bit 0 of the addressed unit's condition byte is set.

CENTRAL PROCESSOR MAINTENANCE

Maintenance features of the ASC require appropriate actions to take place in the Hard Core control of each of the three units in the CP. Figure 3 illustrates the type of control envisioned for the IPU. The AU and MBU have similar hardware.

The cells in each unit are divided into two types, the Hard Core cells and the functional cells. All cells receive clocks from the clock distribution logic centrally located in the Central Processor clock module. Section control bits in the Hard Core are controlled by CCR commands and are used to enable or disable the appropriate cells in each respective unit. Section controls are required to enable the cells to be loaded for any reason - this includes normal operation, maintenance loading, setting, or resetting. A cell can be stored, however, even if the section control would not allow loading. Certain CCR commands require that the functional cells not be allowed to change regardless of

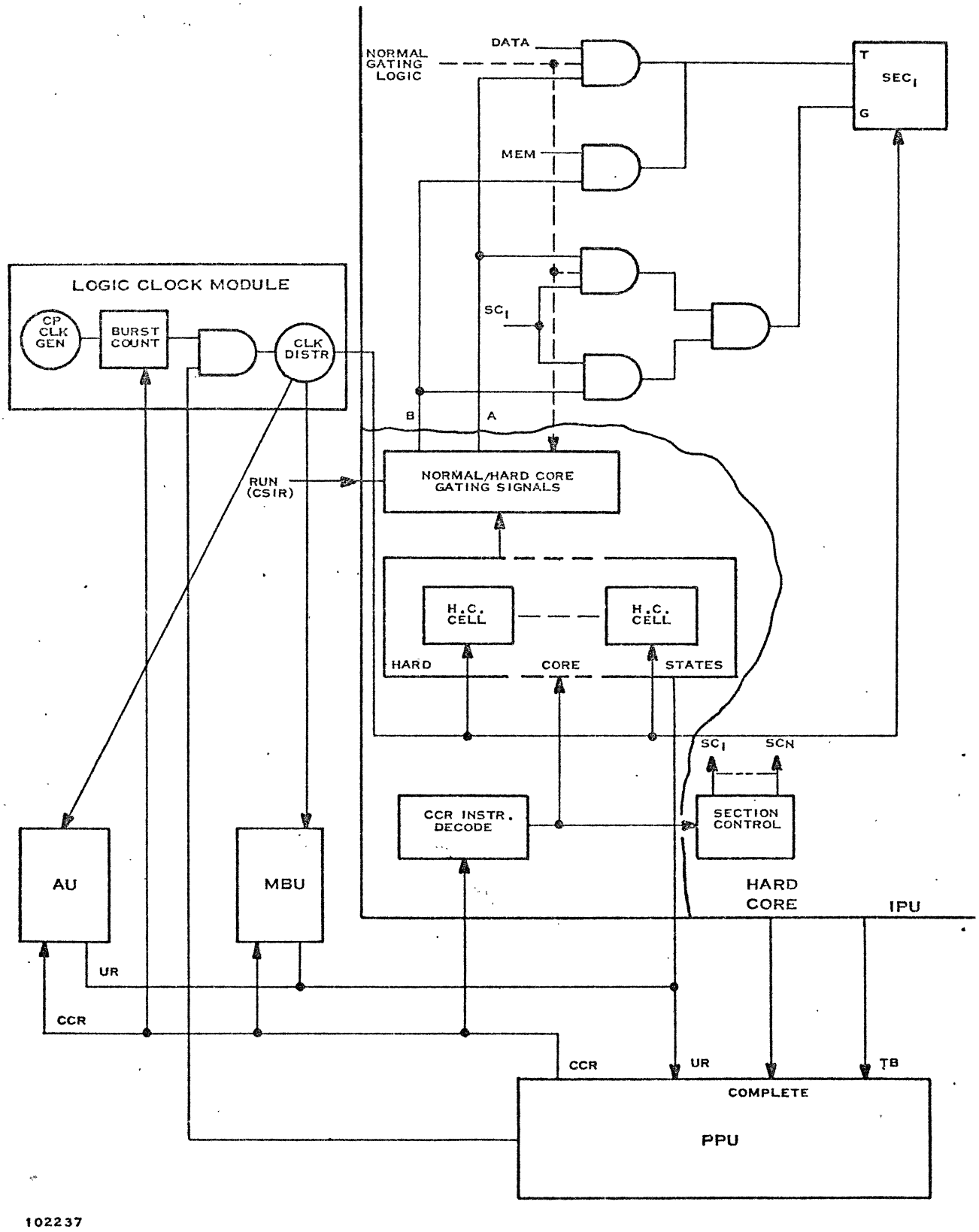


Figure 3. Central Processor Maintenance Logic

the section control status. During this time, the Hard Core will be required to operate thus necessitating clocks from the Logic Clock Module. In this case the Hard Core has the capability of turning off any or all of the gates to the functional cells. The lines that control the gating, labeled A and B, are functions of the state of the Hard Core and possibly a function of the line labeled "Normal Gating Logic". If the Normal Gating Logic is located in such a way that the Hard Core can include it in the determination of A and B the resulting hardware will be a minimum.

The primary control of the CP is through the CCR and the CSIR registers. The run bit in the CSIR controls whether the functional portion of the CP is in an execution or a wait state. In the later mode the Logic Clock Module is supplying clocks to the "hard core" of the CPU. The hard core of the CP can execute any of the CCR commands (except LCM Command which is executed independently by the LCM) in the Wait state and can execute any of the status, intermediate, or details type commands in either the run or wait state. In these cases the run bit controls whether execution proceeds after the command is complete.

The CP's LCM will respond to a 42XX command and reset the TB bit when it has taken the required action.

The command register in the CP's hard core asynchronously accepts the CCR input without requiring clock pulses. The TB is reset indicating reception of the command. When the command is completed, the PSC bit in the CP Condition byte is set. This bit must be reset by software. (Note this signal is used by the CSIR to indicate switching completed by the CP.)

Commands to the LCM can:	Stop the clock:	4280_{16}
	Start continuous clocks:	4200
	Burst of n clock pulses:	428n

The later command allows the CP or hard core to be stepped through their commands. If the CP is to execute a burst of n steps, the run bit must be on when the Burst command is given.

The CCR commands used by the CP are described below:

CCR code (hex)	
40XX	NOOP
4100	Stores all section control using pointer at location 10. The CP halts and transfers the state of its section control to CM. The AU stores its section control into the location specified by the pointer in location 10. The MBU stores its section control one word past the location specified by the pointer. The IPU stores its section control two words past the location specified by the pointer.
4101	Loads all section control using pointer at location 11. The CP acts in a manner similar to the Store section Control instruction except a new section control state is loaded.
4102	Unlock PC. Unlocks the PC and allows the normal instruction sequence to continue.
4103	Lock PC. The present address register is not allowed to be incremented or set to a new value. This causes the corresponding instruction to flow down the pipe; once all old instructions have been completed a state will be reached where all levels of the pipe are executing the same instruction.
4104	Reset. Resets all CP storage cells (to 0) in every CP section not inhibited by section control. This command may only be sent when the Run bit is off.
4105	Set. Sets all CP storage cells (to 1) in every CP section not inhibited by section control. This command may only be sent when the Run bit is off.

4108 Store Status using pointer at location 14. The CP permits all instructions which are currently in process to go to completion. No new instructions are fetched by the instruction fetch unit after receiving this signal. After all instructions have been completed, the program status doubleword and all register files are stored according to the address given by memory location 14.

4109 Load Status using pointer at location 15. The CP is reset and then a new program status doubleword and all register files are loaded according to the address given by memory location 15 and then proceeds with execution, provided that the CP is in the "run" state as defined by bit R of the Context Switch Interlock Register (CSIR).

410A Exchange Status using pointers at locations 14 and 15, and load map and protect bounds using pointer at 28. The CP first performs as for 4108; it then performs as for 4109. Simultaneously, the memory map is loaded from the address given by memory location 28. The protection bounds registers loaded are the ones for the IPU and MBU buses. The CP execution does not proceed until the new map and protect bounds from location 28 have been loaded. This action is independent of the status of the CSIR register except as stated in 4109.

410B Store intermediate using pointer at location 16. If a vector instruction is being processed within the arithmetic unit, the vectors are abnormally terminated and the intermediate results within the arithmetic unit are permitted to complete. Then, status information is stored for the entire CP such that, when reinstated, the program that was executing would be resumed at the point in the vector computation at which the CP status was recorded provided that the vector instruction is not one of the types that has to be restarted from the beginning.

If a scalar instruction is being processed, any intermediate results within the arithmetic unit are permitted to complete. Then, status information is stored for the entire CP such that, when reinstated, the program that was executing would be resumed at the point where the CP status was recorded.

410C Load Intermediate using pointer at location 17. The CP immediately loads the status information, as described under 410B, beginning with the address given by memory location 17 and then proceeds with execution, provided that the CP is in the "run" state as described by bit R of the Context Switch Interlock Register (CSIR).

410D Exchange Intermediate using pointers at locations 16 and 17, and load map and protected bounds using pointer at location 28. The CP first performs as described under 410B; it then performs according to 410C.

410E Store details using pointer at location 18. The CP immediately stores its internal status (all flip-flops) beginning with the address given by location 18 and resumes execution if Run bit is 1, otherwise it halts.

410F Load details using pointer at location 19. The CP immediately loads its total internal status (all flip-flops not inhibited by section control) beginning with the address given by location 19 and proceeds with execution if the run bit is 1.

42mn CP Logic Clock Module Command. This command is detected by logic in the LCM not by the CP's hard core. The format is:

*	Continuous	Normal Clock	0 X 0 0
	"	Margin Clock	0 X 0 1
	"	Slow Clock	0 X 1 0
	"	Normal Clock	0 X 1 1

Note: n is ignored in above commands.

Burst of n Normal clock pulses	1 X 0 0
Burst of n Margin clock pulses	1 X 0 1
Burst of n Slow clock pulses	1 X 1 0
Burst of n Normal clock pulses	1 X 1 1

44nn Store byte nn of hard core status into UR. The CP stores the specified syte into the CP's UR. The CP clock must be off during this command.

DATA CHANNEL MAINTENANCE

A CMR interface is provided to allow setting and testing of the internal functions of each DC and DCC pair. Using the CCR commands the DCC control and buffer registers may be loaded or read one byte at a time, the command in the control register executed, or the indicator bits from the DIU set.

MEMORY CONTROL UNIT MAINTENANCE

Control of the MCU maintenance (Section A) is through the CCR and the unit registers. Paths are also provided via the MCU to interface with the CM modules. The CCR commands used by the MCU are:

CODE (Hex)	
10 XX	NOOP
11 bn	Fetch byte n (0-F) of Processor Interface Subunit b(0-7) status into UR
11 (b+8)n	Fetch byte n(0-F) of Memory Interface Subunit b's (0-7) status into UR
12 bn	Store byte n(0-F) of Processor Interface Subunit b(0-7) status from UR
12 (b+8)n	Store byte n(0-F) of Memory Interface Subunit b(0-7) status from UR

130n	Fetch byte n(0-F) of the MCU Control Interface Subunit into UR
131n	Store byte n(0-F) of the MCU Control Interface Subunit from UR
1488	Store all Map and Protect Registers into CM using pointer at location 38
1489	Load all Map and Protect Registers from CM using pointer at location 39
15b8	Store Protect Registers for Processor Interface Subunit b(0-7) into CM using pointer at location 38
15 b9	Load Protect Registers for Processor Interface Subunit b(0-7) from CM using pointer at location 39
16 b0	Reset Processor Interface Subunit b(0-7) registers
16 (b+8)0	Reset Memory Interface subunit b(0-7) registers
16 X2	Reset all MCU registers
17 80	Shutdown Central Memory
17 m8	Store Map segment m(0-3) from CM using pointer at location 38
17 m9	Load Map segment m(0-3) from CM using pointer at location 39