Programmer's Reference Series

ECLIPSE® MV/8000 Principles of Operation

ECLIPSE® MV/8000 Principles of Operation

Notice

Data General Corporation (DGC) has prepared this manual for use by DGC personnel, licensees, and customers. The information contained herein is the property of DGC and shall not be reproduced in whole or in part without DGC's prior written approval.

Users are cautioned that DGC reserves the right to make changes without notice in the specifications and materials contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the materials presented, including but not limited to typographical, arithmetic, or listing errors.

NOVA, **INFOS** and **ECLIPSE** are registered trademarks of Data General Corporation, Westboro, Massachusetts. **DASHER** and **microNOVA** are trademarks of Data General Corporation, Westboro, Massachusetts.

Ordering No. 014-000648
©Data General Corporation, 1980
All Rights Reserved
Printed in the United States of America
Rev. 00 April, 1980

ECLIPSE MV/8000 Principles of Operation

Introduction 3
Definition of Terms 3
The Addressing Scheme 3
Protection 4
The Instruction Set 4
The Organization of this Manual 4
Related Manuals 7

Chapter 1
An Introduction to the MV/8000 Hardware 9
Memory System 10
System Cache 11
Memory Modules 12
Bank Controller 13
Address Translation Unit 13
Processing Unit 14
Instruction Processor 14
Microsequencer 15
ALU 15
I/O System 16

Chapter 2
Logical Addressing 19
Program Counter 19
Instruction Addressing 20
Data Addressing 20
Data Lengths 20
Displacements 21
Addressing Modes 22
Direct and Indirect Addressing 23
Word Addressing 23
Byte Addressing 25
Bit Addressing 27
Summary 29

Chapter 3
Logical to Physical Address Translation 31
Introduction 31
The Paging Mechanism 32
Referenced and Modified Flags 32
Segment Base Registers 32
Page Tables 33
The Translation Process 34
3.5.1 One-level Page Table Translation 35
Two-level Page Table Translation 36
Initialization 38
Power Up 38
Summary 38

Chapter 4
The Protection System 39
Process-wide Protection 39
Source-Destination Verification 40
Referencing Other Segments for Data 40
Ring Crossing 41
Indirection Protection 44
Page Protection 45
Page Table Entry Validation 45
Access Validation 45
Protection Faults 45
Priority of Protection Faults 46
Servicing a Protection Fault 46
Page Faults 47

Chapter 5
Reserved Memory Locations and Faults 49
Page Zero 49
Page Zero Locations for Segment 0 49
Page Zero Locations for Segments 1-7 51

Chapter 6
Data Types 55
Program Counter 55
Processor Status Register 55
Fixed Point Data 56
Fixed Point Accumulators 56
Fixed Point Format 56
Carry 57
C/350 Compatability 57
Floating Point Data 57
Floating Point Registers 57
Floating Point Format 58
C/350 Compatability 59
Commercial Data 60
Commercial Registers 60
Data Type Indicator 60
Commercial Formats 60
Character Strings 63
Data Manipulation 63

Chapter 7
Fixed Point Instructions 65
Conventions 65
Fixed Point Indexed Address Instructions 66
Fixed Point Single-Word Indexed Arithmetic Instructions 67
Fixed Point Double-Word Indexed Arithmetic Instructions 67
Memory to Accumulator Instructions 68
C/350 Word Memory to Accumulator Instructions 68
Bit Manipulation Instructions 68
Byte Manipulation Instructions 70
Fixed Point Single-Word Arithmetics 71
Fixed Point Double-Word Arithmetics 72
Fixed Point Single-Word Logicals 73
Fixed Point Double-Word Logicals 74
Single-Word Compare Accumulators 75
Double-Word Compare Accumulators 76
Other Instructions 76
Programming Notes 77
C/350 ALC Manipulation 77
ALC Instructions 77
C/350 ALC Instruction Execution 78

Chapter 8
Floating Point Instructions 83
True and Impure Zero 83
Normalized Format 83
Magnitude 83
Guard Digits 84
Floating Point Operation 84
Maintaining Accuracy 84
Finishing Up 85
Floating Point Instructions 85
Floating Point Faults 88

Chapter 9
Commercial Instructions 91
Decimal Instructions 91
Commercial Faults 92
Return Blocks 93
CF0, CF2 and CF3 93
CF1 94
CF4 and CF7 95
CF6 96

Chapter 10
Character String Instructions 97
Character Manipulation 97
Character Instructions 97
C/350 Character Instructions 97

Chapter 11
Stacks and Fault Handling 99
Introduction 99
The Wide Stack 99
Stack Registers 99
Wide Stack Operation 100
Wide Stack Instructions 101
The Narrow Stack 102
Narrow Stack Pointer 102
Narrow Frame Pointer 103
The Narrow Stack Limit 103
Narrow Stack Fault Address 103
Return Block Format 103
Narrow Stack Instructions 103
Stack Faults 104
Wide Stack Faults 104
Narrow Stack Faults 105
Examples 107
Stack Usage 108
The Protection Mechanism 108
Fixed Point Overflow 109
Floating Point Fault 110
Program Flow 111

Chapter 12
Program Flow Instructions 113
Direct Alteration 113
Stack Changes 116

Chapter 13
System Control Instructions 121
Privileged Instructions 121
Queues 122
Building a Queue 122
Queue Descriptor 123
Setting Up and Modifying a Queue 123
Examples 123
Queue Instructions 126

Chapter 14
Input/Output 131
The I/O System 131
Programmed I/O 131
Data Channel I/O 131
Burst Multiplexor I/O 132
Busy and Done Flags 132
Interrupt On Flag 133
Priority Mask 133
I/O Instructions 133
Interrupts 134
ATU Enabled/Disabled 134
Address Resolution 135
Handler Identification 135
C/350 Interrupt 135
Immediate Interrupt 135
Vectored Interrupt 137
Interrupting An Instruction 140
Standard I/O Devices 141
Programmable Interval Timer 142
Real-Time Clock 142
Asynchronous Line Controller 143

Chapter 15
The I/O Processor 145
Forms of Host-IOP Communication 145
The MAP 145
Communication Instructions 145
Elements of the IOP 146
IOP Memories 146
MAP 146
User and Data Channel Maps 146
Parity Generator 147
Host-IOP Interface 147
Interface Elements 147

Cross Interrupts 148
Setting the Interrupt Request Flags 148
Host-IOP Communications Instructions 149
Programming Examples 150
Example 1 150
Example 2 151
Changing the Host Data Channel Map from the IOP 154

Chapter 16
The MV/8000 Instruction Dictionary 157
General Programming Notes 157
Instruction 158

Chapter 17
I/O Instruction Dictionary 323
General I/O Instructions 323
Burst Multiplexor Channel 328
Device Flag Commands 328
Map Load Formats 328
Map Dump Formats 328
Central Processor 332
Device Flag Commands 332
Common Process 338
Mode A 338
Modes B Through E 338
Host/IOP Communication 339
Device Flag Commands 339
Memory Allocation and Protection 341
Device Flag Commands 342
Programmable Interval Timer 347
Device Flag Commands 347
Real Time Clock 348
Device Flag Commands 348
Primary Asynchronous Line Input 349
Device Flag Commands 349
Primary Asynchronous Line Output 350
Device Flag Commands 350

Chapter 18
IOP Communication Instruction Dictionary 351
IOP/Host Communication 351
Device Flag Commands 351

Appendix A
The ASCII Character Set 355

Appendix B
Context Block Format 357

Appendix C
MV/8000—C/350 Program Combinations 361
Using C/350 Instructions 361
Expanding a C/350 Program to Run on the MV/8000 Computer 362
Calling a C/350 Subroutine From an MV/8000 Program 363

Appendix D
Anomolies 365
MV/8000 Instruction Opcodes 365
Program Counter Wraparound 365
Float/Fixed Conversions 365
Address Wraparound 365
C/350 Signed Divide Instructions 366
C/350 Vector and NIO Instructions 366
Floating Point Trap 366
Floating Point Numerical Algorithms 366
C/350 Commercial Faults 367
C/350 MAP Instructions 367

Appendix E
C/350 Memory Allocation and Protection 369
MAP Functions 369
Address Translation 369
Sharing Physical Memory 370
MAP Modes 370
Mapped Mode 370
Unmapped Mode 371
MAP Protection Capabilities 371
Validity Protection 371
Write Protection 371
Indirect Protection 371
I/O Protection 371
MAP Protection Faults 372
Load Effective Address Mode 372
Initial Conditions 372
MAP Instructions 373

Appendix F
Instruction Execution Times 375

Appendix G
Floating Point Operations 377
Floating Point Addition 377
Floating Point Subtraction 377
Floating Point Multiplication 377
Floating Point Division 378

Appendix H
Standard I/O Device Codes 379

Introduction

The ECLIPSE® MV/8000 system represents an extension of minicomputer technology which provides computational characteristics normally associated with mainframe machines. It is a sophisticated, state of the art, 32-bit processing system that retains substantial hardware and software compatability with previous 16-bit ECLIPSE systems.

Some readers may be unfamiliar with the terms used to describe the features of the ECLIPSE MV/8000 system, so the following section provides a brief definition of terms. This is followed by a short description of the organization of this manual, and a list of associated manuals.

Definition of Terms

The following glossary relates particularly to the new addressing scheme and instruction set of the MV/8000. (Note that throughout this manual the ECLIPSE MV/8000 system is usually referred to simply as the MV/8000.)

The Addressing Scheme

Logical and Physical Address Spaces
The MV/8000 main memory makes up the *physical address space*. Physical addresses range from 0 to 2 megabytes. This space is much smaller than the *logical address space*, which is 4 gigabytes.

Logical Addresses
The MV/8000 uses 31-bit word addresses which can reference all 4 gigabytes of the logical address space.

Segmentation
The MV/8000's large logical address space is divided, or segmented, into eight smaller logical address spaces. Each of these eight *segments* is a complete address space of 512 megabytes.

Mapping and Demand Paging
The size of the MV/8000 logical address space means that not all logical locations can be represented in physical memory at the same time. The *demand paging* system moves pages between physical memory and a storage device upon demand, and also keeps track of pages currently in memory. The *address translation unit,* or *ATU*, translates the specified logical address to its physical equivalent.

Page

A page is a 2 Kbyte block of contiguous logical addresses. The demand paging system uses the page as the smallest unit of logical memory that can be moved between physical memory and storage devices.

Page Table

A page table is made up of *page table entries*, or *PTEs*. Each PTE contains information about one page. The processor uses this information when translating a logical address to a physical one. A page table contains up to 512 PTEs.

Protection

The MV/8000 system uses a hardware-implemented hierarchical protection system that allows programs different levels of privilege. Each segment has a different level, or *ring*, of protection associated with it. This means that each ring governs the associated segment with a different degree of privilege. Ring 0 has the highest degree of protection, so the kernel of the operating system would normally reside in Segment 0.

The Instruction Set

The ECLIPSE MV/8000 instruction set is a superset of the previous (16-bit) ECLIPSE instruction set. In this manual, the new 32-bit instructions are referred to as *MV/8000-specific* instructions. The 16-bit instructions supported by the MV/8000, but which are also supported by previous (16-bit) ECLIPSEs (such as the ECLIPSE C/350), are referred to as C/350 instructions.

MV/8000-Specific Instructions

These instructions manipulate data with lengths of 8, 16, or 32 bits. The mnemonics of the instructions indicate the size of the data fields referenced. The mnemonic preceded by the letter **N** manipulate 16-bit (narrow) data; **W**, 32-bit (wide) data. There is no special mnemonic prefix for those instructions that manipulate 8-bit data.

There are also mnemonic prefixes that indicate the addressing range of the instruction. **X** indicates that the instruction has a 64-Kbyte (extended) offset addressing range; **L**, a 4-gigabyte (long) addressing range.

MV/8000 - C/350 Compatibility

The MV/8000 supports the instruction mnemonics and binary opcodes of most instructions implemented on the ECLIPSE C/350 . This means that most programs that execute on the C/350 computer will also execute on the MV/8000 without recompiling or reassembling.

Note that the C/350 instructions maintain their limitations of a 64-Kbyte addressing range.

The Organization of this Manual

The contents of each chapter and appendix of this manual are as follows.

Chapter 1 An Introduction to the MV/8000 discusses the hardware implemented on the MV/8000. A general system overview introduces many of the system's capabilities and how they interact, as well as many of the terms that will be discussed in later chapters.

Chapter 2 Logical Addressing illustrates how the MV/8000 uses information contained in registers and instructions to form logical addresses. These addresses can reference data and other instructios.

Chapter 3 Logical to Physical Address Translation takes the logical addresses described in Chapter 2 and shows how they are converted into physical addresses in main memory. The demand paging system that implements this translation mechanism is also explained.

Chapter 4 The Protection System details the types of checks that are made to ensure valid addresses. System-wide protection shows how *rings*, the instruments of the MV/8000's hierarchical protection system, check memory references for valid accesses and control transfers. To ensure page protection, rings check for valid accesses to pages of physical memory. The consequences of invalid references, page and protection faults, are also described.

Chapter 5 Reserved Memory Locations and Faults lists the memory locations that contain information pertinent to system maintainance. Such locations contain the starting addresses of fault handlers, the contents of system registers, and other types of information.

Chapter 6 Data Types describes the various data types supported on the MV/8000, their formats, and the registers used to manipulate them.

Chapter 7 Fixed Point Instructions introduces the instructions used to manipulate fixed point data. The instructions are grouped in tables according to function.

Chapter 8 Floating Point Instructions summarizes how the MV/8000 performs floating point operations. The instructions used to perform these operations are summarized in tables.

Chapter 9 Commercial Instructions illustrates the decimal data formats used by MV/8000 commercial data. The instructions that manipulate decimal data and a brief discussion of comercial faults ends the chapter.

Chapter 10 Character String Instructions lists the instructions that manipulate character strings.

Chapter 11 Stacks and Fault Handling discusses two types of stacks and their variety of uses in the MV/8000 system as well as in applications programs. Stack instructions and examples round out the discussion.

Chapter 12 Program Flow Instructions details the two types of program flow: direct alteration of sequential flow, and changes made to the stack. Both categories use MV/8000-specific and C/350 instructions to perform the change.

Chapter 13 System Control Instructions describes the MV/8000 *privileged* instructions and the conditions which must be met in order to use them. *Queues* and the instructions that manipulate them make up the second half of the chapter.

Chapter 14 Input/Output describes the three types of I/O used on the MV/8000: programmed I/O, data channel I/O, and burst multiplexor channel I/O. Closely connected to the I/O system is the interrupt mechanism, which supports three types of interrupts: C/350 interrupts, MV/8000-specific immediate interrupts, and vectored interrupts. A brief description of the programmable interval timer, the real-time clock, and the asynchronous line controller completes the chapter.

Chapter 15 The I/O Processor introduces the I/O Processor (IOP), a complete ECLIPSE processor contained within the MV/8000 cabinet. The IOP interfaces with the host processor, but operates independently. Examples of loading the IOP are included in the discussion.

Chapter 16 The MV/8000 Instruction Dictionary contains a detailed explanation of all instructions supported by the MV/8000 (except I/O and IOP instructions). The instruction entries are presented alphabetically by mnemonic.

Chapter 17 I/O Instruction Dictionary contains instruction entries for all MV/8000 I/O instructions. The instructions are grouped according to the device that uses them: general I/O, followed by those used by the BMC, CPU, IOP, MAP, PIT, RTC, TTI, and TTO. Within each category, the instructions are arranged alphabetically, by device mnemonic.

Chapter 18 IOP Instruction Dictionary contains instruction entries for the four instructions that manipulate the IOP. The instructions are presented alphabetically by mnemonic.

Appendix A ASCII Character Set is a table of the ASCII character set, with the related decimal, octal, and hexadecimal codes.

Appendix B Context Block Format shows the format of the context block.

Appendix C MV/8000—C/350 Program Combinations describes how to run C/350 programs on the MV/8000 computer, how to use C/350 instructions on an MV/8000, how to expand a C/350 program to run on an MV/8000, and how to call a C/350 subroutine from an MV/8000 program.

Appendix D Anomolies discusses restrictions in using C/350 instructions on an MV/8000 computer.

Appendix E C/350 Memory Allocation and Protection describes the C/350 MAP, which is supported on the MV/8000.

Appendix F Instruction Execution Times lists typical execution times for all the instructions supported by the MV/8000.

Appendix G Floating Point Operations describes the four floating point operations, addition, subtraction, multiplication, and division.

Appendix H Standard Device Codes is a table of the Data General standard device codes.

Instruction Indexes are aids to locating entries in the Instruction Dictionaries. These indexes are: instruction index by instruction name; instruction index by mnemonic.

Related Manuals

Other manuals describing aspects of the ECLIPSE MV/8000 system are:

The System Control Processor, Operator's Reference Series (DGC No. 014-000649)

The ECLIPSE MV/8000, Product Summary Series (DGC No. 014-000650)

The ECLIPSE MV/8000, Field Engineer's Maintenance Series (DGC No. 015-000106)

Chapter 1
An Introduction to the MV/8000 Hardware

The ECLIPSE MV/8000 is a sophisticated 32-bit system equipped with advanced data processing features. The majority of these features are implemented in hardware to provide fast service with a minimum of system overhead. This chapter gives a brief description of the hardware that implements these features and how the hardware interacts.

As shown in Figure 1.1, the MV/8000 computer contains four main systems:

- A block-oriented MOS memory system,
- A microprogrammed processing unit,
- An independent I/O system,
- An operator-controlled System Control Processor (SCP) for system interface and diagnostic functions.

The four main systems are connected by three 32-bit data busses. **IPM** connects memory, the I/O system, and the SCP. **CPM** allows data to flow between memory and the processing unit. The third data bus, **CPD**, connects the processing unit, the SCP, and the I/O system. In addition, there are two physical address busses: **IPA**, which connects memory, the SCP, and the I/O system, and **CPA**, which connects memory and the processing unit.

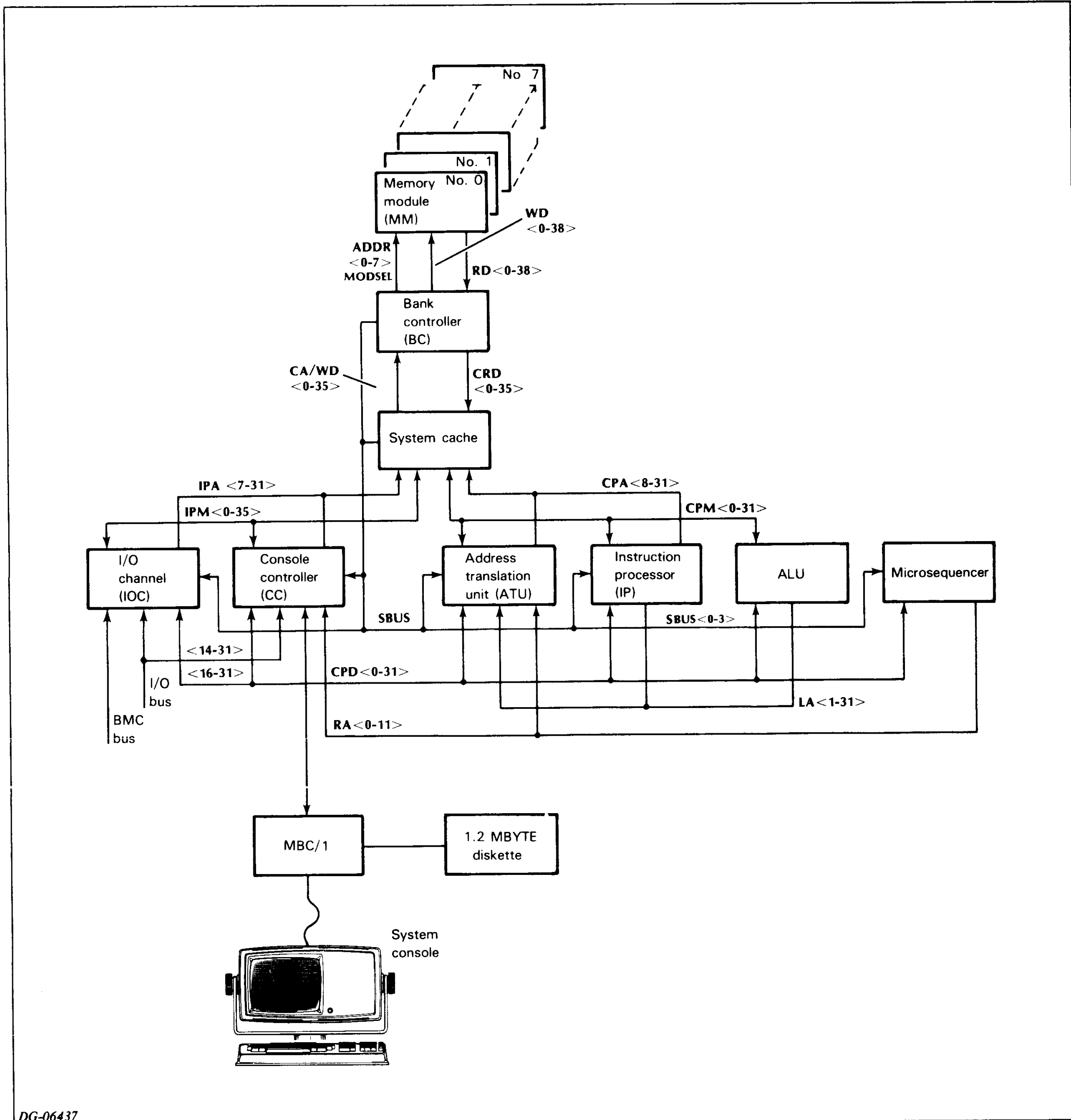


Figure 1.1

Memory System

The MV/8000 memory system is block-oriented. This means that the elements that make up the system expect and manipulate uniform data sizes and formats. The elements transfer data in 16-byte blocks (4 successive double-words) to one another.

The major elements of the memory system are:

- The system cache,
- The memory modules,
- The bank controller,
- The address translation unit (ATU).

System Cache

The system cache functions as both a look-ahead and look-back buffer for the system. This reduces the time needed by both the CPU and the I/O system to access main memory.

The system cache contains 1024 blocks of 16 bytes each. These blocks are directly mapped to main memory locations (see Figure 1.2). This means that any block in the system cache contains 16 contiguous bytes from main memory. Note that the system cache blocks cannot contain arbitrary locations from memory. As shown in Figure 1.2, memory can be divided into up to 128 units; each unit contains 1024 16-byte blocks. Block 0 in the system cache can contain Block 0 of any unit in main memory; Block 1 in the system cache can contain Block 1 of any unit in main memory, and so on.

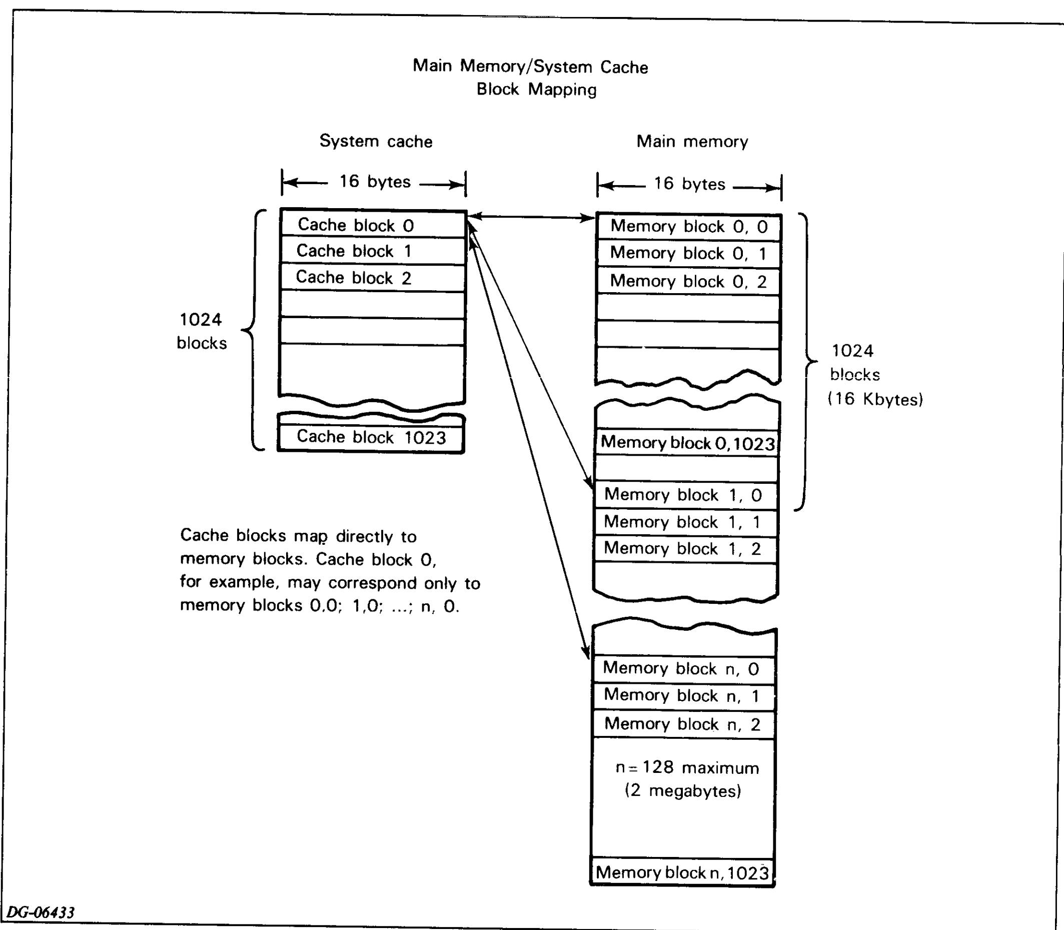


Figure 1.2 Block mapping from main memory to system cache

When a process makes a memory reference to Block *n* of Unit *m* in main memory, the system cache loads the appropriate 16-bit memory block containing the referenced data into system cache Block *n*. This memory block will remain in the system cache until a new memory reference is made to Block *n* of some other Unit *j* in main memory.

When this happens, the system cache examines the *cache block modified bit* of Block *n* of Unit *m* (the block currently in the cache). If the cache block modified bit is 1, the system cache writes Block *n* of Unit *m* back into main memory, then loads the new Block *n* of Unit *j* into system cache Block *n*. If the cache block modified bit is 0, the system cache simply overwrites the current contents of system cache Block *n* with those of Block

n of Unit *j*.

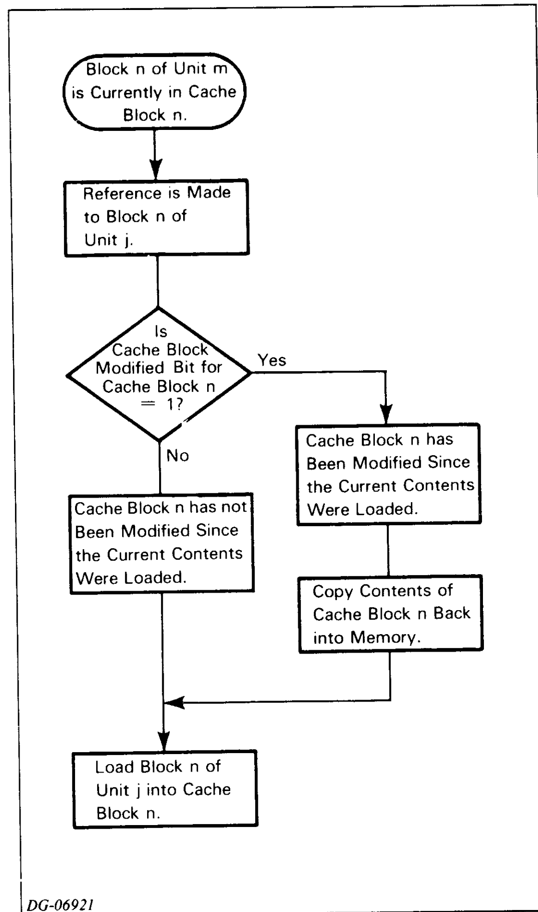


Figure 1.3

The system cache contains two ports: one for the CPU and the other for direct transfers between memory and the I/O system.

The interface between the system cache and main memory transfers 16 bytes of information in 550 nanoseconds for write operations, and 16 bytes in 440 nanoseconds for read operations.

Memory Modules

The MV/8000 can support up to eight dynamic RAM memory modules of 256 Kbytes each. Each module is 64K double-words, where each double-word is 4 bytes long. Each double-word has seven error checking and correction bits associated with it.

Each memory module contains four independent planes, each containing 16K double-words. The planes are arranged so that each plane contains every fourth double-word. This means that plane 0 contains location 0, plane 1 contains location 1, plane 2 contains location 2, and so on. This arrangement allows memory operations to overlap so that 4 double-words (the standard unit of data for the memory system) can be accessed with one address.

The address sent to the module by the bank controller references a position in all four planes of the module. The module checks if the address is valid; if so, the module starts the specified operation in plane 0 for the first double-word. It then starts the indicated operation in plane 1 for the second double-word, then in planes 2 and 3 in the same way.

Overlapping memory operations mean that transfer rates between memory modules and the system cache are very high. The MV/8000 transfers data at a rate of 36.4 megabytes per second.

Bank Controller

The bank controller:

- Performs error checking and correction on transfers between itself and memory;
- Performs the refresh operations required by the memory modules;
- Selects a memory module upon receipt of an address from the system cache;
- Checks for byte parity between the system cache and itself.

Error Detection

Error detection within the memory system helps ensure system reliability and makes a "fail soft" capability possible. The error detection mechanism operates as two independent systems: one that checks data passing between the memory modules and the bank controller, and one that checks data passing between the system cache and the bank controller.

The error checking and correction feature (ERCC) operates on all data passing between the memory modules and the bank controller. It detects and automatically corrects all single-bit errors and detects all double-bit errors.

The second error detection system operates on all data block transfers between the system cache and the bank controller. It generates and checks parity on every byte of data passed between the two elements.

Refresh Operations

As described above, each memory module contains four planes. The planes consist of 16K by one bit chips. To specify the position of any bit on the plane requires a column address and a row address. Because the memory modules of the MV/8000 are dynamic MOS, a refresh operation must occur every 15 microseconds. The bank controller does this by sending a row address to the modules; everything in that row on all planes in all modules will refresh.

Each time a refresh operation begins, the bank controller reads one word from the memory row being refreshed. This word goes through a complete ERCC check and correction and is written back to the memory module. This operation occurs on a different word during each refresh cycle; the entire contents of memory are checked and, if need be, corrected every 4 seconds. This mechanism reduces the possibility of encountering memory errors.

Address Translation Unit

The MV/8000 has a logical memory size of 4 Gbytes and a physical memory size of up to 2 Mbytes. Because the logical address space is so much larger than the physical address space, the MV/8000 uses a demand paged system where units of logical memory called *pages* are stored on disk until needed. When a page on disk is referenced, it is moved to physical memory for manipulation. In addition to the page swapping mechanism, this system also requires a translator that will convert the logical address of a piece of data into a physical address in memory. This translator is called the *address translation unit*, or ATU.

To perform the translation, the ATU uses a series of *page tables* that contain information about the pages of logical memory. These tables contain entries, one for each page, and they indicate if the page is currently in physical memory, if the page is valid and can be accessed, and other information. To avoid referencing a page table for every memory reference, the ATU maintains a table of address translations and access privileges for 256 recently referenced pages. The hardware checks the ATU's table for entries before referencing a page table in memory.

Because the memory references for a procedure tend to cluster in several pages, a needed page translation is likely to be in the ATU's table of address translations. The ATU updates the entries in this table as execution continues.

Referenced and Modified Bits
The ATU also controls two memory management bits for each page: the *modified bit* and the *referenced bit*. The operating system uses these bits during *page faults*.

A page fault occurs when a reference is made to a page that is not currently in physical memory. Each time a page fault occurs, a new page must be transferred from disk to physical memory. This may mean that a page in physical memory must be removed from physical memory to make room for the new page. The modified bit indicates whether the old page has been changed since it was brought into physical memory. If the modified bit for the old page is 1, then some change has been made to the old page and a new copy must be sent to the disk before the new page can be brought in. If the modified bit is 0, the copy of the old page on disk is still valid and the new page can be moved into memory immediately.

The referenced bit helps determine which page in memory should be replaced by a new page being brought in from disk. In general, the page least frequently referenced is the page to be replaced. The referenced bit allows the operating system to determine the frequency of references to individual pages.

Protection Validation
The ATU performs all hardware checks required by the protection system. These checks include access validation, page validation, ring crossing validation, and others. If any of the checks fails, the ATU initiates a protection fault to the operating system. For more information about the types of protection checks, refer to Chapter 4.

Processing Unit

The processing unit of the MV/8000 is composed of:

- A pipelined instruction processor,
- A RAM-based microsequencer,
- A high-speed arithmetic and logical unit (ALU).

Instruction Processor

The instruction processor decodes instructions for execution. Its main component is the *instruction cache*, which provides input to the instruction decoder. The instruction cache is 1 Kbyte in size, 64 blocks of 16 bytes per block, and maps directly to the system cache.

Note that the 16 bytes in the instruction cache blocks correspond to the 16 bytes in the system cache blocks. Like the system cache blocks, the instruction cache blocks cannot contain arbitrary information; Block 0 of the instruction cache can contain any Block 0 in the system cache; Block 1 in the instruction cache can contain any Block 1 in the system cache; and so on.

During program execution, the instruction cache provides a speed increase because of its look-ahead and look-back potential. Program loops or backward jumps, in particular, profit from this feature.

Instruction Execution

The instruction processor executes instructions in four steps:

- An instruction is fetched from instruction cache.
- The instruction opcode is parsed to obtain the the starting address of the microcode routine, and operand information is collected.
- Parsed information waits to be executed (while a new instruction is parsed.
- The microinstructions are executed.

This four-stage sequence allows four instructions to be in the pipeline at any one time: one instruction being executed, the next being parsed and readied, the next being decoded, and the fourth being fetched.

Microsequencer

The microsequencer contains the RAM memories that form the control store for the MV/8000's microcode. The microcode generates the control signals required by the other elements of the system to perform their operations. In particular, the microcode controls all ALU operations.

Because the control store is constructed from RAMs, the microcode must be reloaded each time the system is powered up. This is done by the System Control Processor (see next section).

ALU

Built around advanced bit slices, the 32-bit ALU performs complex operations in a minimum number of cycles. Two separate sections exist; one manipulates the exponents of floating point numbers, and the other manipulates floating point mantissas, fixed point quantities, and addresses.

ALU Accumulators

The MV/8000 contains four 32-bit fixed point accumulators. The ECLIPSE C/350 16-bit fixed point accumulators correspond to bits 16-31 of the MV/8000 accumulators. The program counter (PC) is 31 bits wide; bits 1-3 specify the current segment of execution, and bits 4-31 specify an address in the segment. The C/350 15-bit PC corresponds to bits 16-31 of the MV/8000 PC.

Four floating point accumulators, each 64 bits wide, contain both the exponent and mantissa of any single- or double-precision floating point operand. These four registers are identical to the C/350 floating point registers. The MV/8000 floating point status register (FPSR) is 64 bits wide.

Four 32-bit registers govern the MV/8000 wide stack: the wide stack pointer (**WSP**), the wide frame pointer (**WFP**), the wide stack limit (**WSL**), and the wide stack base (**WSB**). Maintaining the stack in hardware speeds up stack management operations.

I/O System

The MV/8000 I/O system is both electrically compatible and program compatible with the ECLIPSE C/350. This means that MV/8000 supports the full family of standard DGC peripherals with high-speed burst multiplexor channel (BMC) I/O, data channel I/O, and programmed I/O. All three types of I/O are under the control of the I/O channel board.

Both the BMC and the data channel transfer data to and from the system cache directly; data does not pass through the processor. The BMC transfers blocks of data to and from memory at a rate up to 14.54 Mbytes per second on output and up to 16.16 Mbytes per second on input. The data channel operates at rates up to 1.3 Mbytes per second on output and 2.27 Mbytes per second on input. Information can move between the system cache and the I/O channel board at a maximum of 18.2 million bytes per second. Even at this rate, the central processing unit can continue unabated.

The programmed I/O system operates with words or parts of words being transferred between accumulators in the processor and I/O devices. These transfers can be used to set up the parameters of the transfers for the higher speed channels.

System Control Processor

The System Control Processor (SCP) is a system within the MV/8000 that has its own microcomputer. That is, the SCP has its own CPU and its own operating system. The SCP:

- Is a soft system console,
- Performs diagnostic functions,
- Loads MV/8000 microcode into the microsequencer.

As a soft console, the SCP performs system control functions under operator control. It permits the operator to load or examine and modify MV/8000 main memory and microcode, verify either one against a reference file, and single-step his way through a program, instruction by instruction.

As a diagnostic tool, the SCP runs programs designed to help isolate hardware problems. It also maintains an error log. When an error occurs, the log records the type of error, its location, and the time it occurred.

As a microcode loader, the SCP loads and verifies the MV/8000 microcode when the system is powered up, or under operator control.

The SCP consists of four major parts:

- The console control (CC) board,
- The MCB microcomputer with its own memory,
- A diskette drive,
- An operator's terminal.

The CC board provides all the system timing for the MV/8000. It also connects to other MV/8000 components via several busses to allow internal registers to be examined and modified.

The MBC is the interface between the operator's terminal and the CC board. It operates under the control of its own operating system (i.e., the SCP operating system), which supports all traditional operator functions and also allows the operator to run diagnostic programs.

The diskette drive holds the diskette containing the SCP operating system, diagnostic programs, error logs, and copies of the MV/8000's microcode. The MBC controls the drive.

The operator's terminal gives the operator complete control over the MV/8000 system by transmitting commands to the system and providing direct responses and reports.

Chapter 2
Logical Addressing

As mentioned in the Introduction, the ECLIPSE MV/8000 has a logical address space of 4.3 billion bytes. This address space is partitioned into eight 512 Mbyte sections called *segments* to make memory management easier. The segments are numbered from 0 to 7; Segment 0 is in the lowest order part of logical memory. Figure 2.1 shows the segments of memory.

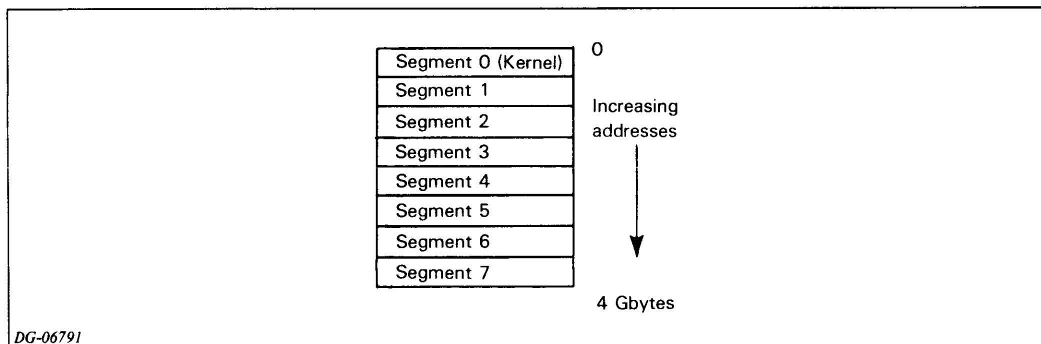


Figure 2.1

To reference information in any segment, use a *logical address*. Each logical address specifies a segment number and a location within that segment. With a logical address, two types of information can be referenced: data, or instructions. To reference data, the processor uses information coded in the referencing command to construct the logical address of the desired data. To reference instructions, the processor uses the address contained in a register called the *program counter.*

Program Counter

The program counter (PC) is 31 bits long. It specifies the logical address of the currently executing instruction. Bits 1–3 of the PC specify the current segment of execution; bits 4–31 specify an address within the segment, as shown in Figure 2.2.

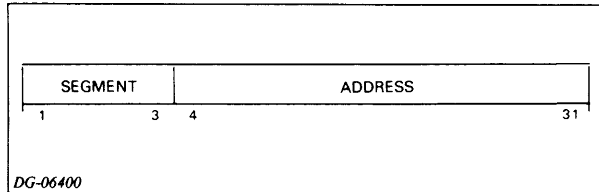


Figure 2.2 Format of the MV/8000 program counter (PC)

When the processor increments the PC to reference the next sequential instruction, only bits 4–31 take part in the increment. This means that every address formed by incrementing the PC will remain within the current segment. The MV/8000 has instructions that change the segment of execution (see Chapter 4, *The Protection System*).

C/350 program flow instructions leave bits 1–3 of the PC unchanged, and set bits 4–16 to 0. Bits 17–31 are loaded with the address provided by the program flow instruction. This means that all C/350 programs are constrained to be located in the first 64 Kbytes of each segment. For more information about specific instructions, refer to the individual instruction descriptions found in Chapter 16, *The MV/8000 Instruction Dictionary.*

The PC can be used as a base register for PC-relative addressing, as described later in this chapter.

Instruction Addressing

The processor addresses all instructions, regardless of their type, in the same way; it fetches the contents of the location specified by the PC.

Data Addressing

As shown above, the PC is used to address an instruction. This type of addressing does not depend on the instruction's contents. To reference data, however, the structure of the data must be known before a reference can be made. Data comes in different types and lengths to cover a variety of requirements. Data types are:

- Fixed point numbers
- Floating point numbers
- Decimal numbers
- Alphanumeric character strings

Chapters 6 through 10 contain detailed discussions of all of these data types.

Data Lengths

The four basic units of information are bits, bytes, words, and double-words. A byte contains eight bits, a word contains sixteen bits, and a double-word contains thirty-two bits.

The first physical word in memory has the address 0; the next, address 1; the next, address 2; and so on.

MV/8000 instructions can specify data of varying precision. The first one or two letters of an instruction mnemonic indicate the sizes used in that instruction. Table 2.1 lists the various mnemonic prefixes and their meanings.

Prefix	Meaning
N	Narrow data - 16 bits wide
W	Wide data - 32 bits wide
X	Extended displacement - 15 or 16 bits wide
L	Long displacement - 31 or 32 bits wide

Table 2.1 Prefixes for MV/8000-specific instructions

Displacements

A displacement is a value contained in an instruction that the processor uses to calculate a logical address. The processor uses it as an offset from some predetermined base value to produce a logical address.

There are three types of displacements used in MV/8000 instructions. *Word displacements* are 8, 15, or 31 bits long and are used to form a *word address.* When manipulating 8- or 15-bit word displacements, the processor extends them to 28 (Mode 0, see next section) or 31 (Modes 1-3) bits before using them to form a word address.

Byte displacements are 16 or 32 bits long; the processor uses them to form *byte addresses.* When manipulating 16-bit byte displacements, the processor extends them to 29 (Mode 0) or 32 (Modes 1-3) bits before using them to form a byte address.

Figure 2.3 shows the various types of displacements.

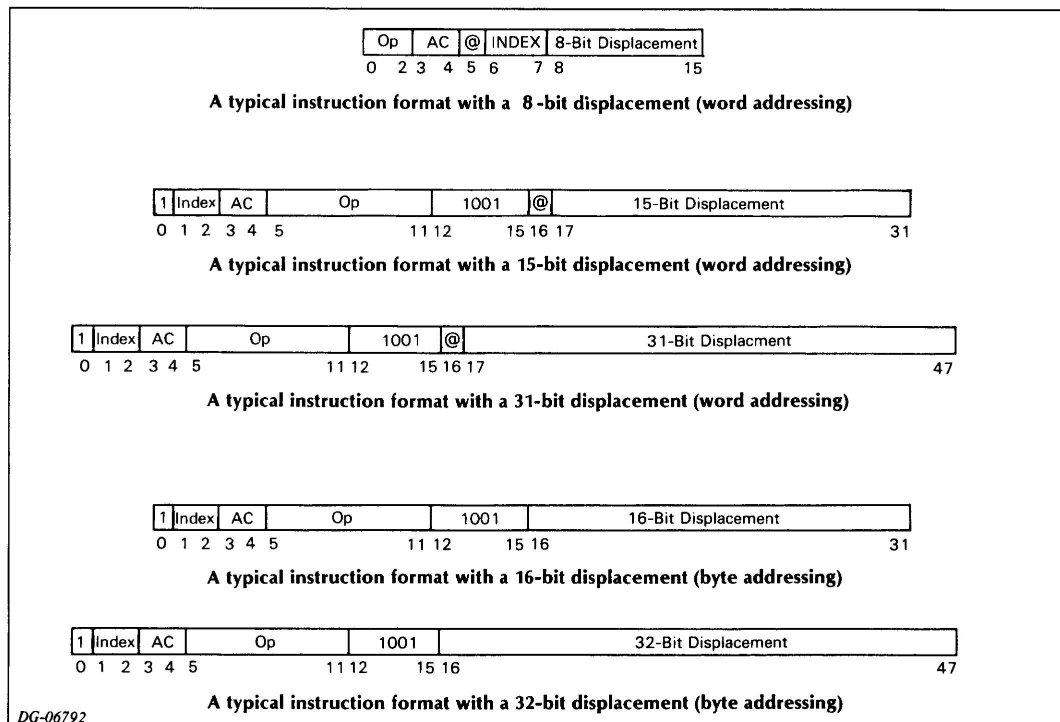


Figure 2.3

As with data, MV/8000 instruction mnemonics specify the size of the displacement used in the address calculation. The first or second letters of the mnemonic describe the displacement size. Table 2.1 lists the mnemonic prefixes and their meanings.

Addressing Modes

To reference a unit of information in logical memory, specify the address of the information. Usually, the instructions used will specify various pieces of information that the processor needs to construct a *logical* address. This information is located in the the *displacement*, *index bits*, and the *indirect (@) bit.*

The two index bits specify an *addressing mode*. The modes specify a base address to be used with the displacement to produce a logical address. These modes are:

- Absolute mode (Mode 0) - index bits are 00
- PC-relative mode (Mode 1) - index bits are 01
- AC2-relative mode (Mode 2) - index bits are 10
- AC3-relative mode (Mode 3) - index bits are 11

Addressing Modes — 31- and 32-bit Displacements
For all modes, the processor adds the base address specified by the mode to the offset specified by the displacement. The sum of the two values is the *intermediate* logical address.

Addressing Modes — 8-, 15-, and 16-bit Displacements
For Mode 0, the processor zero extends any 8-, 15-, or 16-bit displacements. Word displacements (8 or 15 bits long) are extended to 28 bits; byte displacements (16 bits long), to 29 bits. The processor adds the base address specified by the mode to the offset specified by the displacement. The processor appends the value of PC bits 1-3 to the leftmost bit of the 28- or 29-bit value; the result is the *intermediate* logical address.

For Modes 1-3, the processor sign extends any 8-, 15-, or 16-bit displacements. Word displacements (8 or 15 bits long) are extended to 31 bits; byte displacements (16 bits long), to 32 bits. The processor adds the base address specified by the mode to the offset specified by the displacement. The sum of the two values is the *intermediate* logical address.

Absolute Mode
In absolute mode, the processor uses zero as the base value added to the displacement value. With a 31-bit displacement, any location in the logical address space can be directly specified. With a zero-extended 15-bit displacment, any location in the first 64 Kbytes of the current segment of execution can be specified. With a zero-extended 8-bit displacment, any location in the first 377 words of the current segment of execution can be specified.

PC-Relative Mode
The PC-relative mode uses the value contained in the program counter as the base value added to the displacement. This mode allows a program to specify addresses greater than 64 Kbytes with a sign-extended 15-bit displacement.

When an 8- or 15-bit displacement is specified in PC-relative mode, the PC contains the address of the word containing the displacement. When a 31-bit displacement is specified, the PC contains the address of the word containing bits 1-15 of the displacement.

Accumulator-Relative Modes

There are two accumulator-relative modes. One uses the value contained in AC2 as the base value; the other uses the value contained in AC3. As with the PC-relative mode, these two modes allow a program to specify addresses greater than 64 Kbytes with a sign-extended 15-bit displacement.

Direct and Indirect Addressing

After producing the intermediate address from the displacement and index bits, the address translation unit (ATU) translates it from a logical address to a physical one. The processor then uses the *indirect bit* (bit 0 of the intermediate address) to determine the final address.

An indirect bit of 0 specifies direct addressing. This means that the physical address becomes the final address without change.

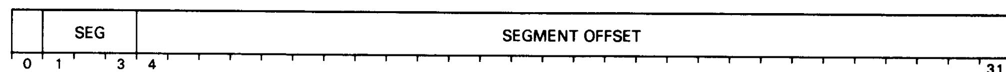
An indirect bit of 1 specifies indirect addressing. This means that the contents of the logical address are to be treated as a double-word pointer to another address. The processor fetches the contents of the double-word location specified by the intermediate address and examines bit 0. If bit 0 is 1, the ATU fetches the contents of the fetched location, translates the contents to a physical address, and fetches the contents of the addressed location. This *indirection chain* is followed until bit 0 of some double-word location is 0. The contents of that double-word location form the last logical address in the chain; the ATU translates the address into a physical one, and uses it as an address to a piece of data.

> NOTE: *The indirection chain may include a maximum of 15 levels of indirection. If more levels are specified, a protection fault occurs; AC2 contains the code 5.*

When an instruction that calculates an effective address is used, the processor checks all source and destination addresses for access validity. This means that when indirect addressing is used, the processor checks each intermediate address obtained in the indirection chain, even if the instruction will not use that intermediate address as the final address (see the **LEF** instruction). When no indirection is specified, the processor compares the final address to the current segment to check for valid access. For more information about validity checks, refer to Chapter 4.

Word Addressing

To form the logical address of a word in memory, the processor uses the information coded in the index and displacement bits of an instruction. As described in previous sections, the processor determines the correct addressing mode and adds the displacement to the specified base value:



where

SEG contains the segment number found in bits 1-3 of the PC, and

SEGMENT OFFSET specifies the displacement in words from the start of the segment.

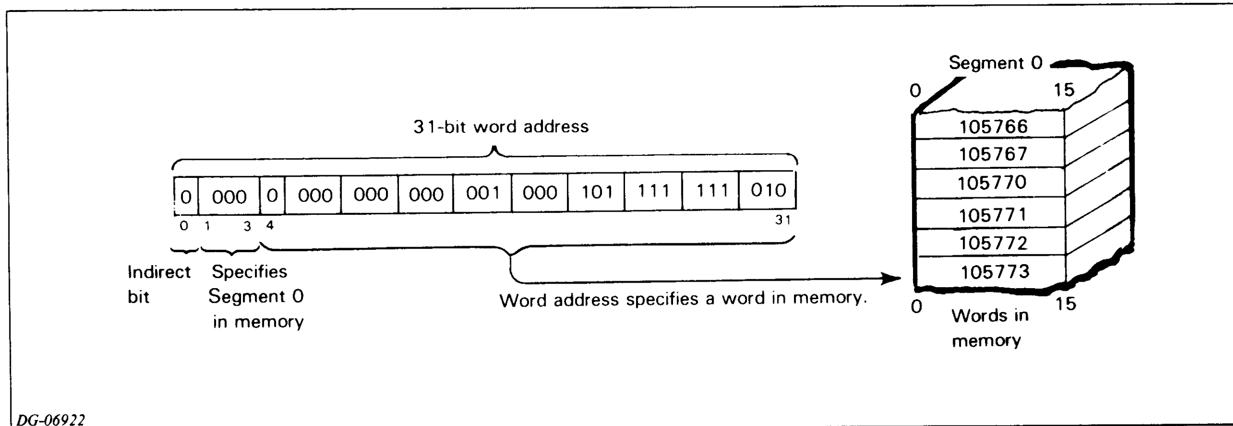
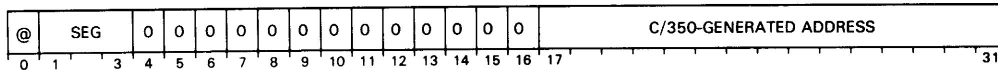


Figure 2.4

C/350 Word Addressing

To reference a word with a C/350 instruction when the ATU is enabled, the processor forms an address with the following format:



where

SEG contains the segment number found in bits 1-3 of the PC, and

C/350-GENERATED ADDRESS is the 15-bit address formed from the information in the C/350 instruction.

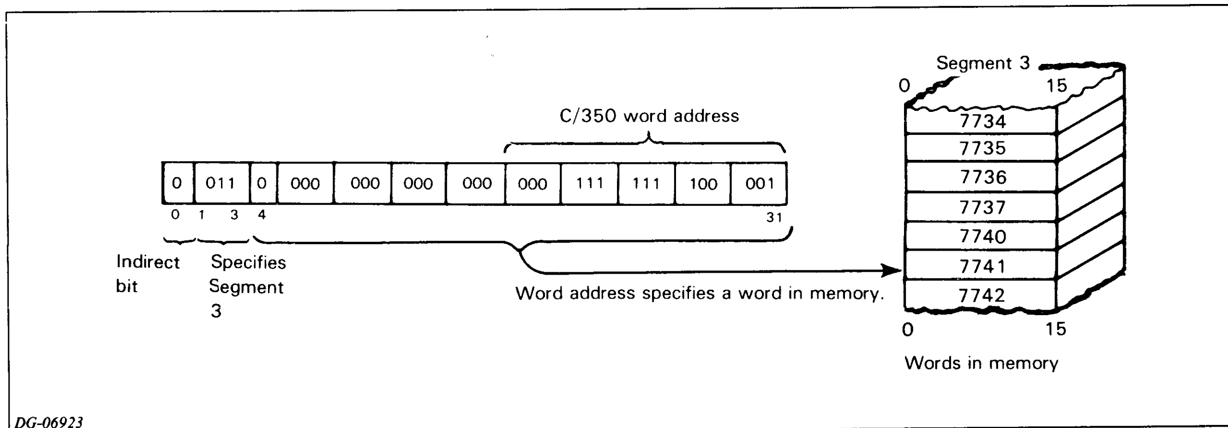
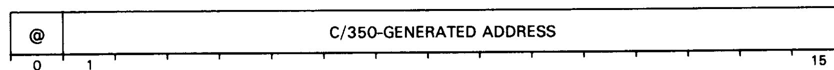


Figure 2.5

When the C/350 MAP is enabled, the word address is 15 bits long and is formed from the index bits and displacement in the C/350 instruction:



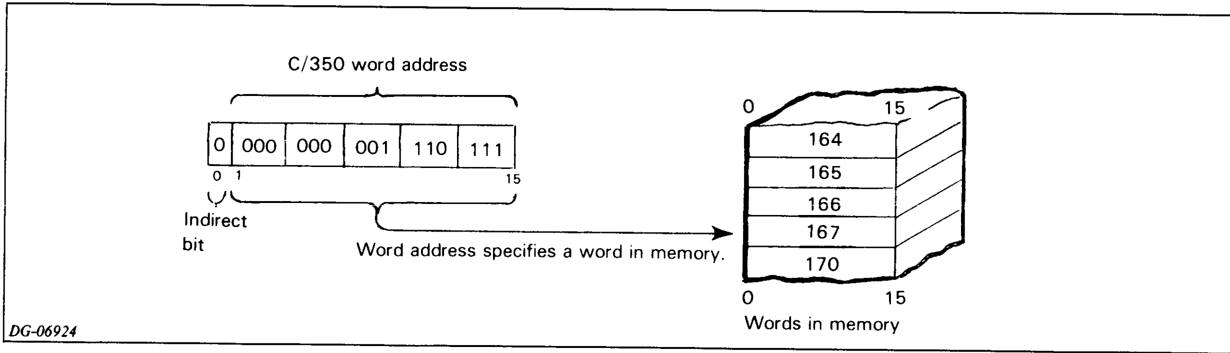


Figure 2.6

Byte Addressing

Calculating a byte address is much like calculating a word address. As in word addressing, the index bits and displacement determine an intermediate address.

NOTE: *Byte addresses cannot specify indirection.*

After all calculation is complete, the processor interprets the byte pointer according to the following format:

where

SEG contains the segment number specified in bits 1-3 of the PC,

SEGMENT OFFSET specifies the displacement in words from the start of the segment, and

BI is the byte indicator. A 0 in this bit specifies the left byte; a 1 specifies the right byte.

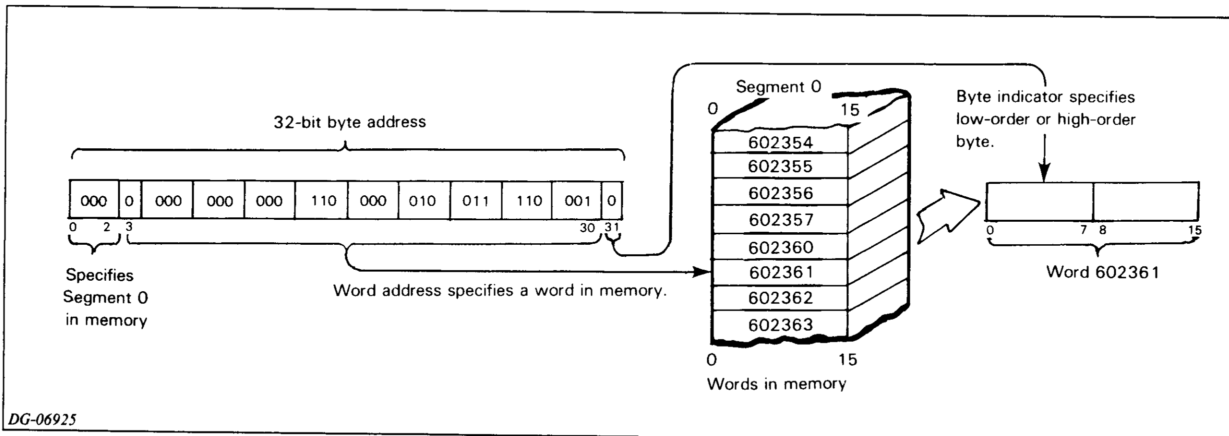
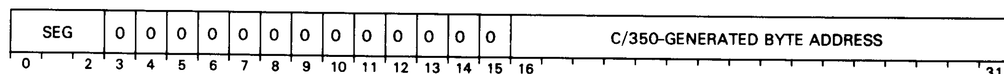


Figure 2.7 Byte addressing: MV/8000 instructions; ATU enabled

C/350 Byte Addressing

To reference a byte with a C/350 instruction when the ATU is enabled, the processor forms an address with the following format:



where

SEG contains the segment number specified in bits 1-3 of the PC, and

C/350-GENERATED BYTE ADDRESS is the byte address formed from the information in the C/350 instruction. This byte address is 16 bits long; bits 0-14 specify a word address and bit 15 is the byte indicator.

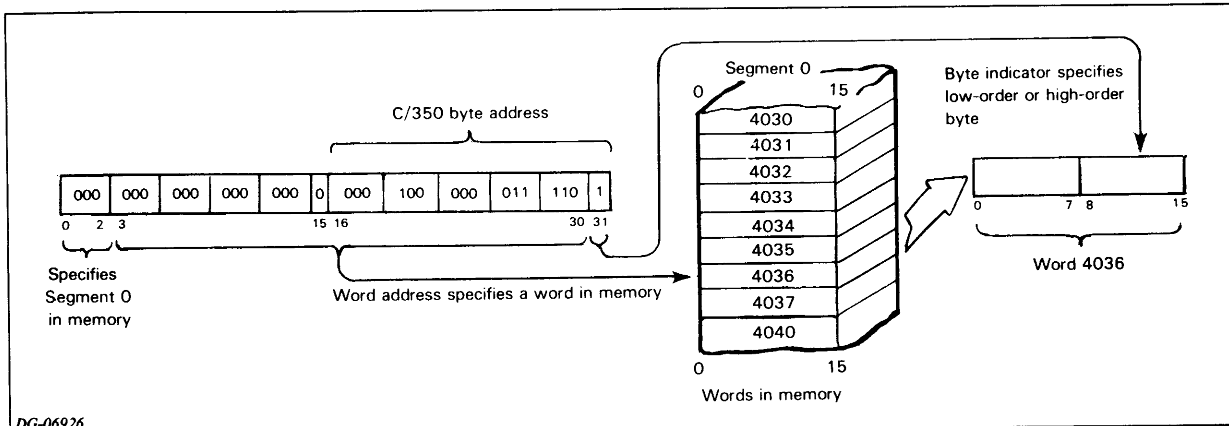


Figure 2.8 Byte addressing: C/350 instruction; ATU enabled

When the C/350 MAP is enabled, the processor uses a 16-bit byte pointer wi*1 the format:

where

C/350 WORD POINTER is the address of a 2-byte word, and

BI indicates one of the two bytes in the addressed word. If bit 15 is 0, the high-order byte (bits 0-7) will be used. If bit 15 is 1, the low-order byte (bits 8-15) will be used. See Figure 2.9.

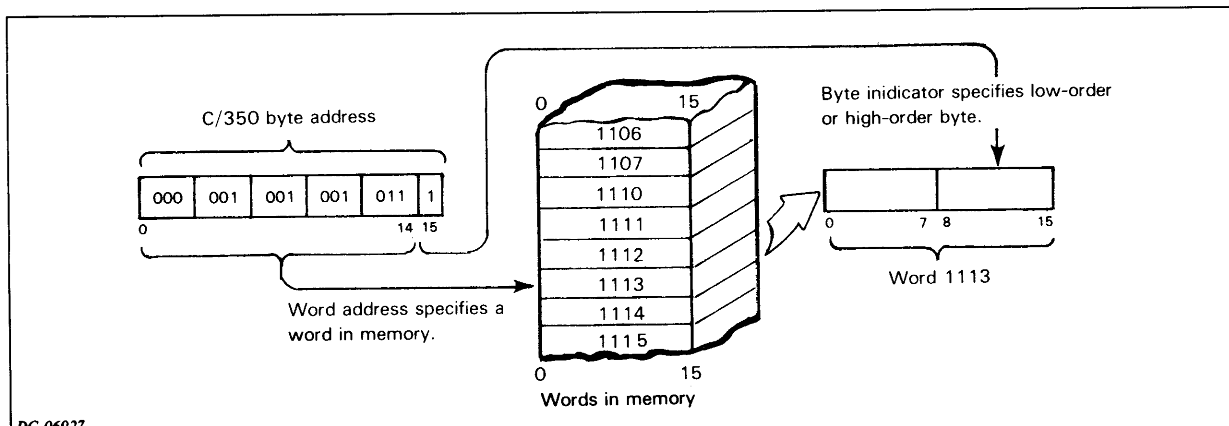
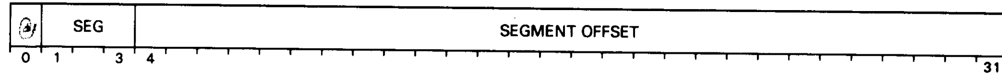


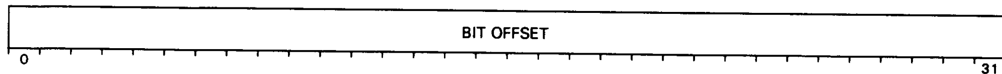
Figure 2.9 Byte addressing: C/350 instruction; MAP enabled

Bit Addressing

Bit addressing uses a word pointer and a bit offset to calculate an address. The word pointer can be indirectable. The format of the word pointer is:



The bit offset can reference any bit in the current segment. It has the format:



The word pointer acts as a base address; it specifies a word in memory. The bit offset specifies a number, *n*. The *n*th bit past the word addressed by the base address is the desired bit.

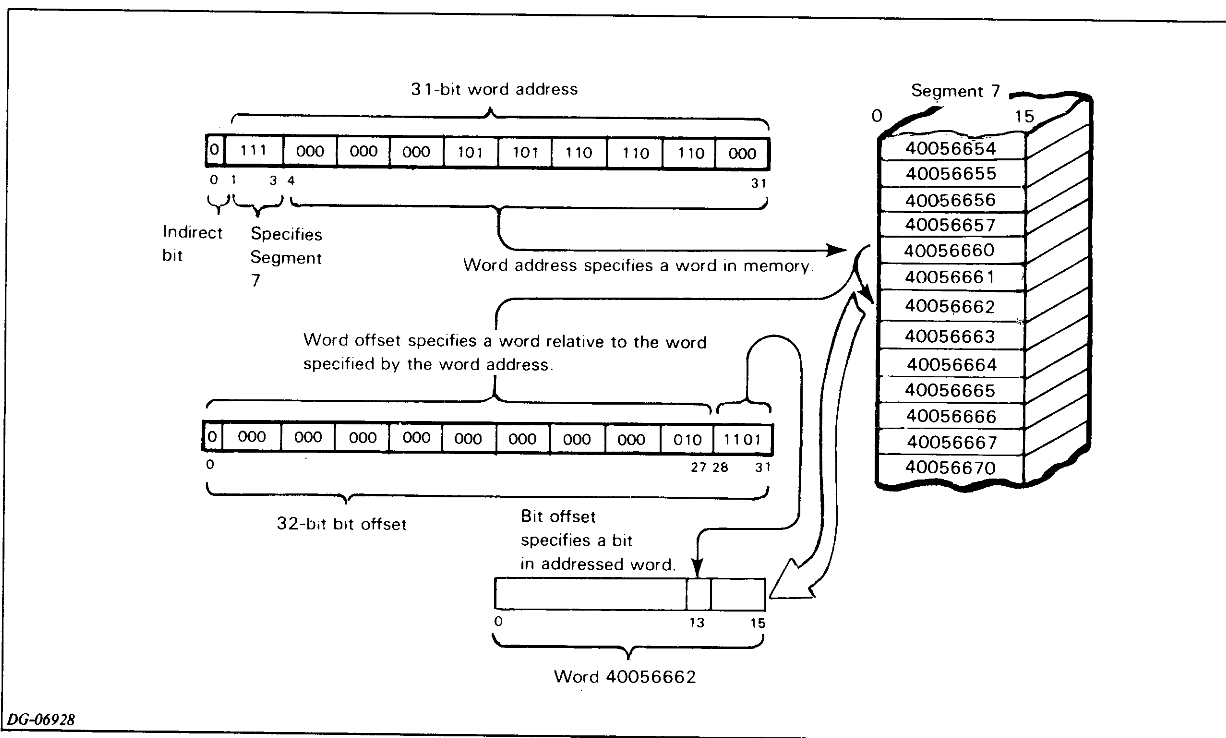
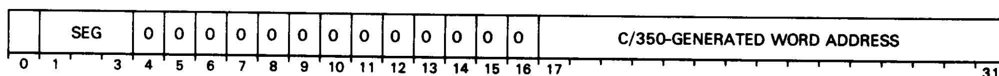


Figure 2.10

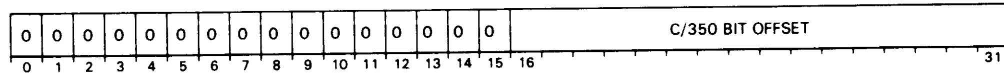
To specify the word pointer and bit offset, load the appropriate values into two fixed point, 32-bit accumulaters. These accumulators are then specified in a bit instruction. If the two accumulators are the same, then the word pointer is assumed to be zero in the current segment.

C/350 Bit Addressing

To reference a bit with a C/350 instruction when the ATU is enabled, the processor forms a word pointer and bit offset from the information contained in the C/350 instruction. The format of the word pointer is:



The format of the bit offset is:



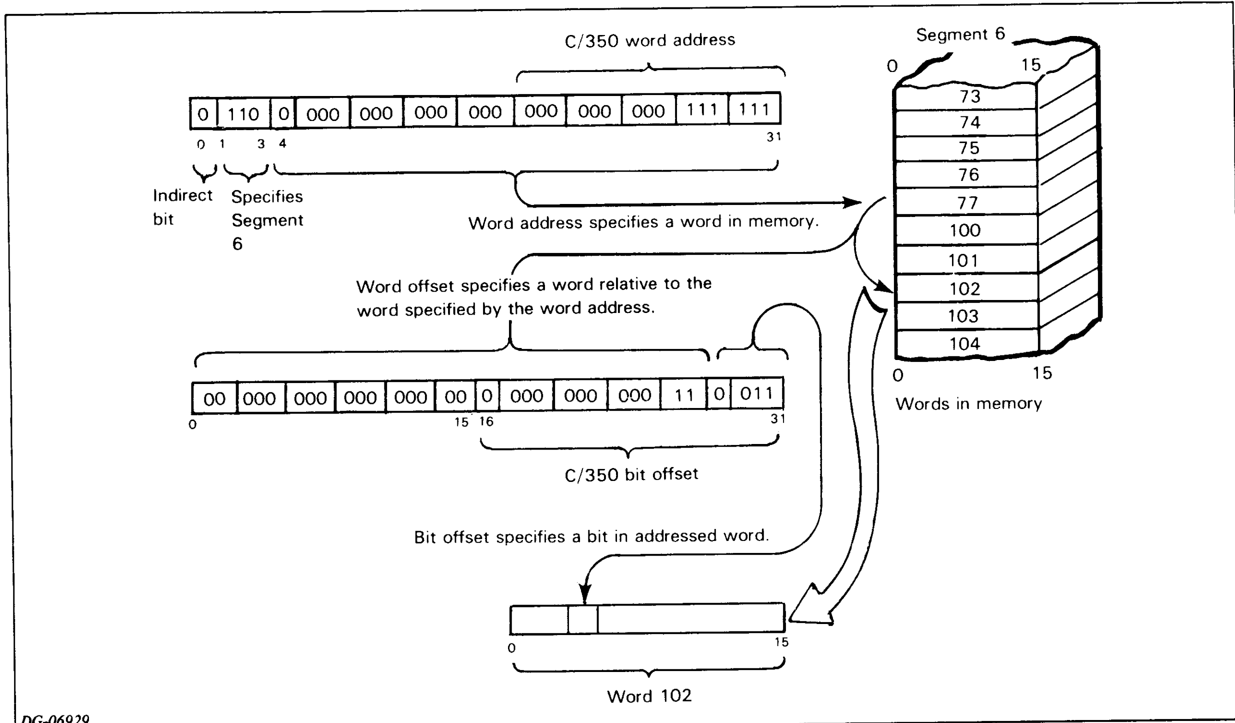
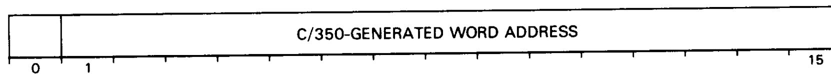
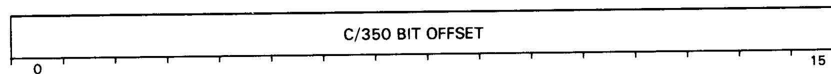


Figure 2.11

When the C/350 MAP is enabled, the processor forms a 15-bit word address and a 16-bit bit offset. The word address has the format:



The bit offset has the format:



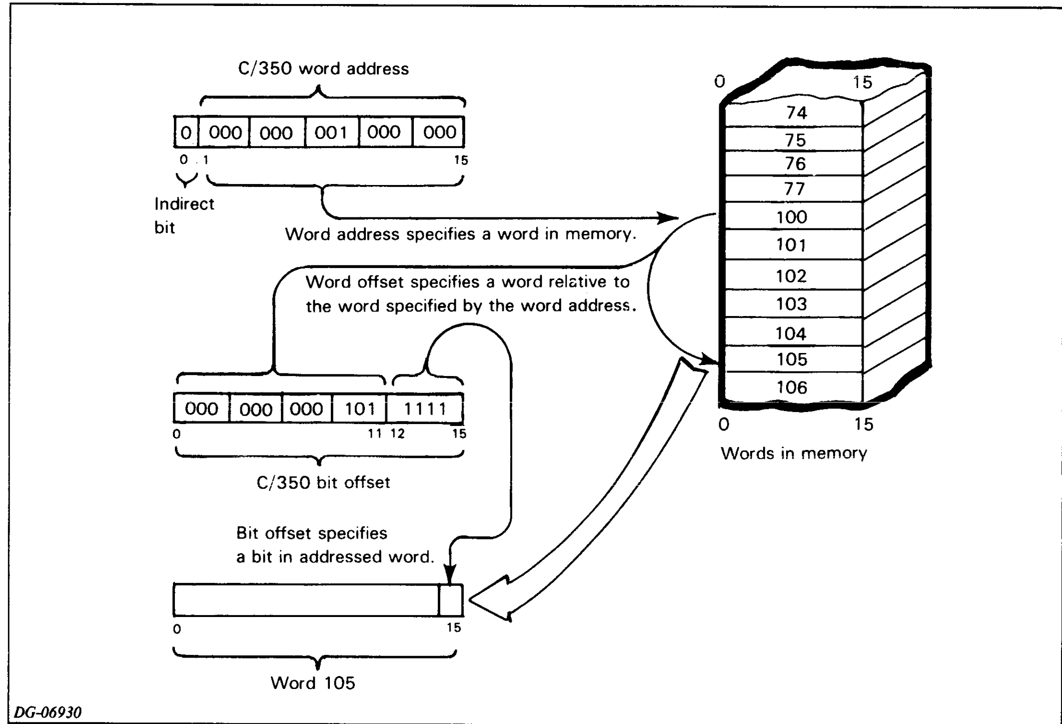


Figure 2.12

Summary

This chapter has explained how the processor uses information provided in an instruction to produce a logical address. This logical address, however, is not sufficient to allow the processor to reference a physical memory location. To do this, the processor must translate the logical address into a physical address. It must also be able to manage the physical locations specified by the translated addresses. The next chapter will describe how the processor translates logical addresses into physical addresses, and how physical memory is managed.

Chapter 3
Logical to Physical Address Translation

This chapter describes the steps used to translate a logical address to a physical address, and the information provided for the management of physical memory.

Introduction

Each of the eight segments in the MV/8000 has its own logical address space. This means that for each segment the processor must translate all logical addresses to physical addresses before it can reference the desired location. To perform the translations, the processor uses a series of tables.

This translation algorithm is based on the following items:

- A standard unit of memory allocation called a *page*. A page contains 2048 bytes of storage which can contain instructions, data, or both.
- A table of entries (referred to as a *page table*) which contains information used to translate a logical address to a physical address. Each *page table entry*, or PTE, contains information relevant to one page of storage.
- A *segment base register* (SBR) for each segment. An SBR contains the physical base address of a page table and an indication of the number of page tables that have to be traversed to produce a physical address.

This translation mechanism is used to implement a *demand paged* storage allocation approach. Demand paging means that a page can be referenced without it necessarily being located in main memory: it may be located in secondary storage. The operating system controls the pages that are in main memory, and when necessary it moves pages from secondary storage to main memory, and from main memory to secondary storage.

The translation process and page management is performed by the *address translation unit*, or *ATU*, of the MV/8000.

As part of the production of a physical address, the translation mechanism performs protection checks and provides information necessary for the management of physical memory. The protection system that preserves system integrity is discussed in the next chapter. In this chapter, the translation process is described as if no protection violations

occur.

The Paging Mechanism

Because its logical address space is larger than the physical memory space, all pages cannot be in physical memory at the same time. The MV/8000's paging mechanism (under the control of the operating system) moves referenced pages in and out of memory whenever necessary. The paging mechanism uses page tables and registers to keep track of the pages and their status.

The paging mechanism also includes a page fault handler. When a reference is made to a page that is not currently in memory, a page fault occurs, signalling the operating system to load the referenced page into memory. The page fault handler loads the appropriate page, and the processor completes the reference. For more information about page faults, refer to the next chapter.

Referenced and Modified Flags

Associated with each physical page in memory are two flags: *referenced* and *modified.* When a successful read is initiated, the referenced flag associated with the physical page of the translated address is set to 1. When a successful write is initiated, the referenced and modified flags associated with the physical page of the translated address are *both* set to 1.

For the purposes of memory management, a successful action is defined as a memory reference which does not result in a protection fault on a resident page. I/O memory references do not effect the state of these flags.

Privileged instructions exist to manipulate the referenced and modified flags. They are shown in Table 3.1.

Mnem	Name	Meaning
LMRF	Load modified and referenced bits	Loads the modified and referenced bits of a specified page into AC0.
ORFB	OR referenced bit	Inclusively ORs the specified referenced bits with a bit string in memory and resets the referenced bits to the ORed value.
RRFB	Reset referenced bit	Resets the modified bits of the specified pages to zero.
SMRF	Store modified and referenced bit	Stores bits 30–31 of AC0 in the modified and referenced bits of the specified page.

Table 3.1 MV/8000 instructions that manipulate the referenced and modified bits

Segment Base Registers

There is a segment base register (SBR) for each of the eight segments. An SBR contains information used for the logical address translation mechanism and for I/O protection. An SBR indicates whether a segment is currently defined, the number of page table levels necessary for logical address translation, the address of translation information, and other system data. The processor checks these registers not only for the data contained there, but to ensure that a specified logical address is valid. If it is not, the

processor invokes the protection system. (For more information about the protection system, see the next chapter.)

Each 32-bit SBR has the following format:

The meanings of the SBR fields are shown in Table 3.2.

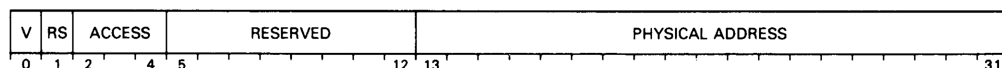
Bit field	Symbol	Description
0	V	Segment validity bit — indicates whether the segment can be referenced. 0 indicates an invalid SBR 1 indicates a valid SBR
1	L	Length bit — indicates the maximum range of the physical memory address. 0 indicates a 1-level page table (maximum range is 1 Mb). 1 indicates a 2-level page table (maximum range is 512 Mb).
2	LEF	LEF bit — indicates whether the processor will operate in LEF or I/O mode. 0 indicates I/O mode. 1 indicates LEF mode.
3	I/O	I/O enable bit — indicates if an I/O protection violation will occur when you try to execute an I/O instruction in your program. 1 indicates the I/O instruction will execute. 0 indicates the protection violation will occur (code= 10 in AC1).
4	RESERVED	Hardware reserved.
5–12	RESERVED	Software reserved.
13–31	PHYSICAL PAGE ADDRESS	Identifies the physical page address in memory of the indicated page table.

Table 3.2 SBR fields

Page Tables

The processor uses page tables to keep track of data pages and page table pages. These tables contain entries, one for each page, that contain the information necessary to reference locations in a particular page. A page table entry (PTE) contains flags that indicate if the page is in physical memory and if it can be referenced. It also contains information needed to translate a logical address to a physical address.

The format of a PTE is shown below.



The meanings of the PTE fields are shown in Table 3.3.

Bit field	Symbol	Description
0	V	Valid page bit — indicates whether a page is currently defined (valid). 0 indicates page is currently invalid. 1 indicates page is currently valid.
1	RS	Resident bit — indicates presence of page in physical memory. 0 indicates page is not in physical memory. 1 indicates page is in physical memory.
2	R	Read access bit — indicates whether a read reference to this page is valid. 0 indicates the read is invalid. 1 indicates the read is valid.
3	W	Write access bit — indicates whether a write reference to this page is valid. 0 indicates the write is invalid. 1 indicates the write is valid.
4	E	Execute access bit — indicates whether an execute reference to this page is valid. 0 indicates the execute is invalid. 1 indicates the execute is valid.
5	---	Reserved for hardware use.
6-12	---	Reserved for software use.
13-31	PHYSICAL ADDRESS	Physical Address field — specifies the 19 high order bits of a physical address.

Table 3.3 PTE fields

The physical address contained in the PTE references one of two things: a page containing an instruction and/or data, or the base of another page table. If the ATU uses only one page table for address translation, the PTE contains the physical address of a data page. In this case, the ATU uses a *one-level page table*. If the ATU uses two page tables for the translation process, then the first PTE referenced contains the address of the second page table. In this case, the ATU uses a *two-level page table.*

The option of specifying either a one- or a two-level page table allows the address space to be tailored to the program. For a small program (less than 1 Mbyte), a one-level mechanism would be used, since a one-level scheme can access the lowest one five-hundred-and-twelfth of memory. For larger programs, a two-level mechanism would be used.

Bit 1 of the referenced SBR determines whether the ATU uses a one-level page table or a two-level page table. When bit 1 of the SBR contains a 0, the ATU uses a one-level page table. When bit 1 contains a 1, the ATU uses a two-level page table.

The Translation Process

The following paragraphs describe how the ATU uses one-level and two-level page tables to translate logical to physical addresses.

3.5.1 One-level Page Table Translation

An earlier section described how the processor generates a logical address. Once the logical address is in the proper form the translation process takes place.

In the following discussion, refer to Figure 3.1. The numbers labeling the text correspond to the numbers displayed on the diagram.

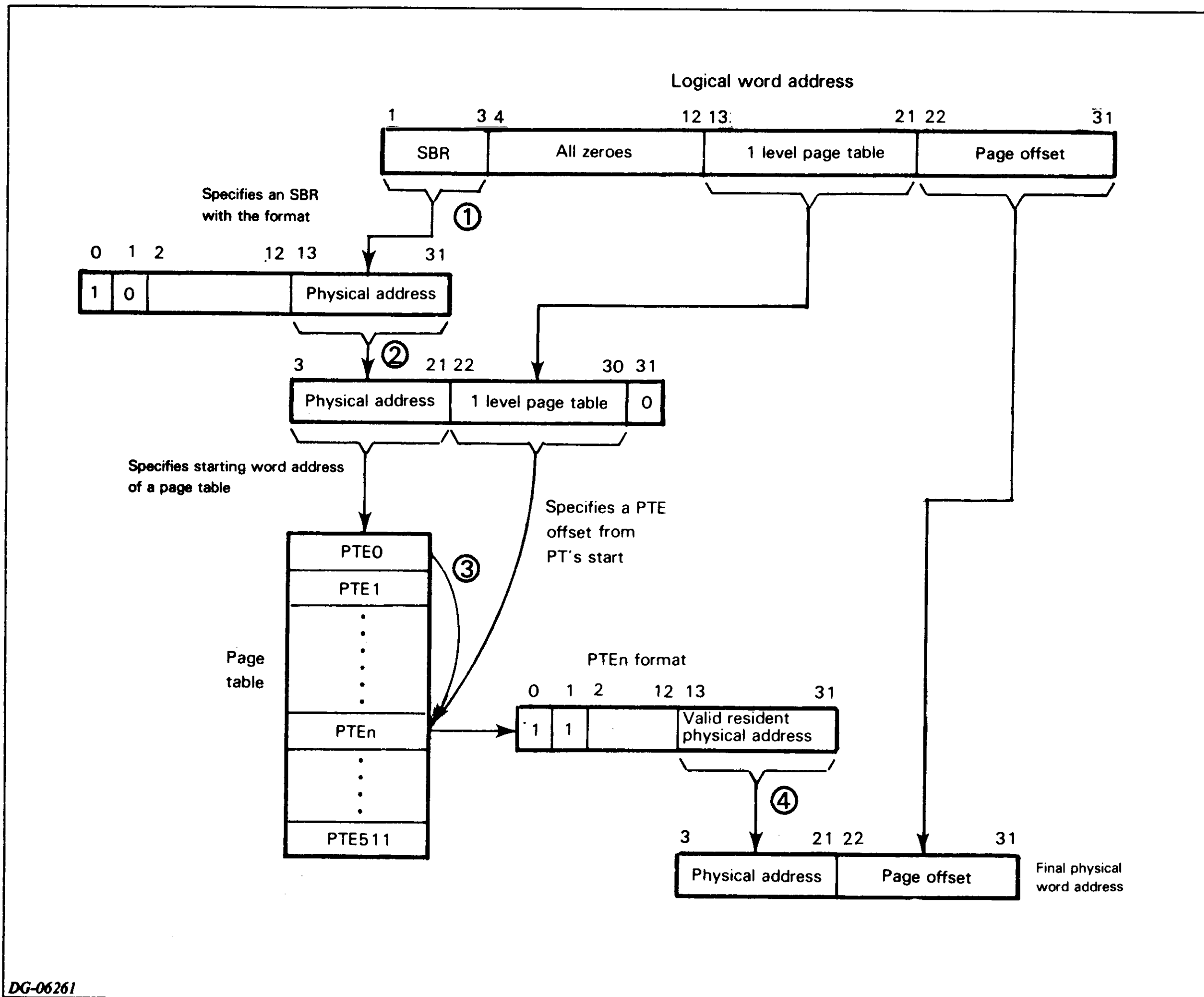


Figure 3.1 One-level page table translation

1. The logical word address to be translated has the format shown in the diagram. Bits 1–3 of the word address specify one of the eight segment base registers (SBRs). The ATU will use the contents of this valid SBR to form the physical address of a PTE.

2. To form this physical page address, the ATU begins with the physical address specified in bits 13–31 of the SBR discussed above. This address becomes bits 3–21 of the PTE address. Bits 13–21 of the logical word address become bits 22–30 of the PTE address. The ATU appends a zero to the right of the PTE address, making a 29-bit word address.

3. Bits 3–21 of the PTE address (unchanged in 2 above) specify the starting address of a page table. Bits 22–31 of the PTE address specify an offset from the start of the table to some PTE (labelled PTE*n* in Figure 3.1). This PTE specifies the starting address of a page of memory.

4. PTE*n* bits 13–31, the page address, become bits 3–21 of the physical address. The page offset field specified in bits 22–31 of the logical word address become bits 22–31 of the physical address. This is the physical word address translated from the original word address.

NOTE: *When using a one-level page table, bits 4–12 of the logical word address must be zero. If they are not zero, a page fault occurs. For more information, see the next chapter.*

Two-level Page Table Translation

Just as in the one-level page table translation process, the processor produces a logical address. In the paragraphs that follow, refer to Figure 3.2. **1.** The logical word address to be translated has the format shown in the diagram. Bits 1–3 of the word address specify one of the eight segment base registers (SBRs). The ATU will use the contents of this valid SBR to form the address of a PTE.

2. To form this address, the ATU begins with the physical address specified in bits 13–31 of the SBR discussed above. This address becomes bits 3–21 of the PTE address. Bits 4–12 of the logical word address become bits 22–30 of the PTE address. The ATU appends a zero to the right of the PTE address, making a 29-bit word address.

3. Bits 3–21 of the PTE address specify the starting address of a page table. Bits 22–31 of the PTE address specify an offset from the start of the table to some PTE (labelled PTE*n* in Figure 3.2). The PTE specifies the starting address of a page table.

4. The ATU now constructs the address of a second PTE. The physical address specified in bits 13–31 of the first (PTE*n*) become bits 3–21 of the address of the second PTE*m*. Bits 13–21 of the logical word address become bits 22–30 of the second PTE's address. The ATU appends a zero to the right of the second PTE address to make a 29-bit word address.

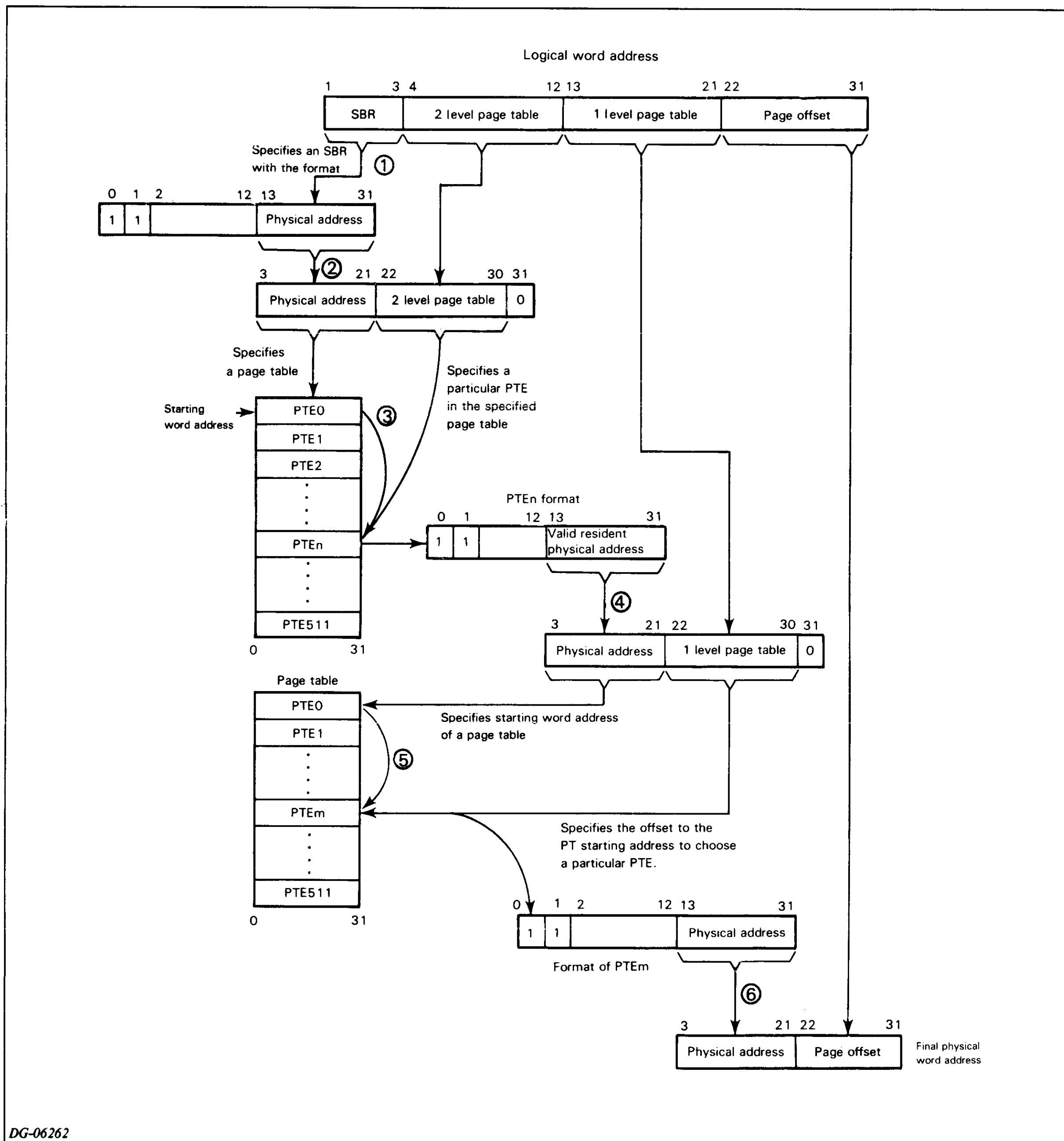


Figure 3.2 Two-level page table translation

5. Bits 3–21 of the second PTE address specify the starting address of a second page table. Bits 22–31 of the second PTE address specify an offset from the start of the second table to some PTE (labelled PTE*m* in Figure 3.2). The second PTE specifies the starting address of a page.

6. The second PTE*m*'s bits 13–31, the page address, become bits 3–21 of the physical address. The page offset specified in bits 22–31 of the logical word address becomes bits 22–31 of the physical address. This last value is the physical word address.

Initialization

Power Up

When the power is first turned on, or after a system reset, the processor is in the following state:

- Logical address translation is disabled (this means that logical and physical addresses are the same),
- The protection system functions as if logical address translation is still enabled. That is, ring maximization is still performed.
- The values of the referenced and modified flags are indeterminate.
- The processor status register (PSR) is cleared to 0, and the processor is halted.

Executing the *I/O Reset* instruction disables logical address translation, just as a system restart or power up does. In addition, after the I/O reset occurs, the processor sets the PSR and bits 0-9 of the floating point status register (FPSR) to 0.

The MV/8000 effective address calculation works the same way when in physical mode as it does when the ATU is enabled. However, because the logical address space exceeds the physical address space, the processor discards a number of the logical address' most significant bits (to give a 20-bit word address) before referencing memory.

When in physical mode, the processor executes a C/350 **LMP**, **SYC**, or a MAP Enable instruction exactly as if the processor were an ECLIPSE C/350 .

Summary

This chapter described the logical-to-physical address translation mechanism. The next chapter describes the protection system and how it ensures that all references are valid.

Chapter 4
The Protection System

The previous chapter described the address translation mechanism. This chapter will describe how the protection system works when the ATU is enabled to validate the translation mechanism. Protection takes two forms: process-wide protection, and page protection. Process-wide protection checks ring protection, data references to other segments, control transfers, and I/O protection. Page protection checks for valid pages and valid page accesses.

Process-wide Protection

The MV/8000 uses a hierarchical protection mechanism. Under such a scheme, the operating system can assign varied degrees of privilege to programs. This scheme provides protection for sensitive data as well as for the operating system itself, and also allows common routines to be used by all programs.

In a system using a hierarchical protection mechanism, the address space of each user includes the operating system. This greatly reduces the number of calls to the operating system that must be included in a program. A program invokes the operating system with a subroutine call rather than specific system calls. In fact, a program sees the operating system as a system-provided subroutine.

To make this hierarchical arrangement work, the MV/8000 uses a separate protection system for each segment. These protection systems (*rings*) allow varying degrees of access to locations in the corresponding segment. Ring 0 corresponds to Segment 0 and provides the most rigorous protection. (Privileged instructions can only be executed in Ring 0.) The kernel of the operating system is located in Segment 0. Ring 1 corresponds to Segment 1, and so on. Ring 7 is the least privileged protection system. User programs are usually located in Segment 7 to minimize the chance of damage to the operating system or other facilities.

Process-wide protection governs the first part of the address translation process. When a reference from one segment to another is made, the ring of the destination segment determines whether the source segment can legally make the reference to the destination segment. This holds true whether the reference is a simple data reference, or a transfer of control from one segment to another.

Source-Destination Verification

When the processor translates a logical address to a physical address, the protection system first checks the SBR specified by the logical address. As described in the previous chapter, the SBR contains information about the validity of the referenced segment. Bit 0 determines if the segment can be referenced. If it cannot, a protection fault occurs and AC1 contains the code 3.

Bit 1 of the SBR specifies the number of page table levels this segment uses in the translation mechanism. The value of this bit places a restriction on the segment's logical address to be translated. If SBR bit 1 contains a zero, indicating that this segment can perform only one- level page table translations, then zeroes must be specified in bits 3–11 of the logical byte address. If a value other than zero is specified in this field, a page table fault occurs. For information about page faults, see the end of this chapter.

Bit 2 of the SBR is as an indicator of **LEF** mode. The C/350 *Load Effective Address* instruction uses the same format as C/350 I/O instructions. The processor must have some means to identify when these instructions should be interpreted as I/O instructions and when as **LEF** instructions. When bit 2 of the SBR contains a 0, then the processor will execute all **LEF** and I/O instructions as I/O instructions. A 1 in this bit means that the processor will execute all **LEF** and I/O instructions as **LEF** instructions.

The SBR also determines whether I/O instructions can be used. A 1 in bit 3 of the SBR means that any I/O instruction will be executed. If this bit contains a 0, a protection fault will occur each time the processor attempts to execute an I/O instruction. When this fault occurs, AC1 contains the code 10.

Referencing Other Segments for Data

A program's ability to reference data in another segment depends upon:

- The program's logical location in memory;
- The protocols defined by the ring protecting the referenced segment.

When a reference to another segment is made, the processor notes the segment from which the reference is being made (the source segment). It also notes the destination segment, and determines if the source segment's reference is valid according to the destination segment's protection mechanism. Table 4.1 shows which accesses are valid or invalid. A *V* shows that the access is valid; an *F* shows that the attempted access will cause a protection fault. If a fault occurs, AC1 contains the code 4.

Source Segment	Destination Segment							
	0	1	2	3	4	5	6	7
0	V	V	V	V	V	V	V	V
1	F	V	V	V	V	V	V	V
2	F	F	V	V	V	V	V	V
3	F	F	F	V	V	V	V	V
4	F	F	F	F	V	V	V	V
5	F	F	F	F	F	V	V	V
6	F	F	F	F	F	F	V	V
7	F	F	F	F	F	F	F	V

Table 4.1 Valid and invalid segment accesses

Access Brackets

The MV/8000 protection system controls cross-ring accesses with *access brackets*. Each ring has an access bracket which specifies a set of segments that can make legal references to data protected by this ring.

When performing a read or a write, the access bracket indicates the highest order segment that can make legal references to data protected by this ring. In the example used above (Table 4.1), the read and write access brackets for Ring 4 would specify 4, which means that Segments 0 through 4 can access data protected by Ring 4, but Segments 5 through 7 cannot.

When performing an execute, the access bracket indicates a set that contains only the current segment. This means that if the current segment is 6, only programs located within Ring 6 can be executed.

When the processor determines that the reference to another segment is valid, the rest of the translation process and protection checks take place. If the reference is invalid, a protection fault occurs and AC1 contains the code 4.

Ring Crossing

When the processor increments the PC to reference the next instruction, it increments only the 28 least significant bits. This is to keep all standard program references in the current segment.

To transfer control to another segment, a particular type of program control instruction must be used. The processor will execute this type of program control instruction only if two conditions are met:

- The instruction must be a subroutine call or return, such as **LCALL** or **WRTN**. The processor ignores the segment field of the logical address generated by any other type of program flow instruction.
- A subroutine call must be inward (towards Ring 0); a subroutine return must be outward (towards Ring 7). Outward calls and inward returns will cause protection faults and AC1 will contain either 7 (outward calls) or 8 (inward returns).

Gates and Gate Arrays

A valid inward subroutine call causes the processor to transfer control to the specified segment. The transfer is strictly controlled by a *gate array*, which is part of the protection system. A gate array is a series of locations in the destination segment that specify legal entrypoints, or *gates*, into that segment. The gate array has the format shown in Figure 4.1.

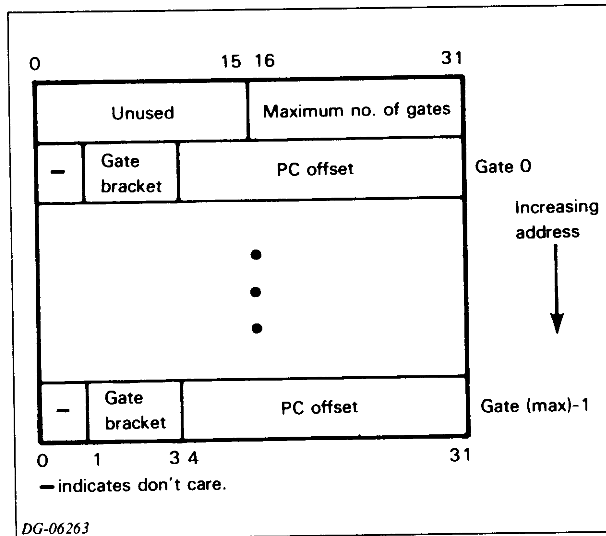


Figure 4.1

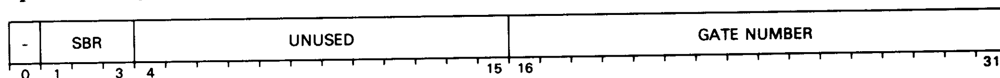
The first-double word of the gate array specifies the total number of gates in the gate array. The rest of the locations contain the gates.

Bits 1–3 of each gate specify a *gate bracket*. A gate bracket is similar to the access brackets used in acessing another ring. The gate bracket specifies an unsigned integer in the range of 0 to 7. This integer specifies the rings that can legally use this gate. For example, a gate contains a 3 in its gate bracket field. This means that only Rings 0 through 3 can use this gate. If the gate bracket field contains a 6, then Rings 0 through 6 can use this gate.

Specifying an invalid gate causes a protection fault to occur. AC1 contains the code 6.

Logical locations $34–35_8$ in the called segment contain a pointer to the first word of the gate array.

When a valid inward cross-ring call is specified, the processor interprets the address specified by the calling instruction according to the following format:



Bits 16–31 specify one of the gates in the gate array.

Performing the Crossing

When an inward cross-ring call is specified, the processor compares the gate number specified in the call to the maximum number of gates specified in the gate array (bits 16–31 of the first double-word). If the gate number in the call is greater than or equal to the maximum, then the call fails and a protection fault occurs. AC1 will contain 6. Note that the protection fault occurs in the original segment, not the called one.

If the gate number in the call is less than the maximum, then the processor uses the gate number to index one of the gates. The processor reads the contents of the appropriate gate, and compares the segment number specified by the PC to the segment number specified by the gate. If the number specified by the gate is less than or equal to that specified by the PC, then the cross-ring call is not made and a protection fault occurs. AC1 will contain a 6. Again, the fault occurs in the source segment, not the destination segment.

If the number specified by the gate is greater than that specified by the PC, then the processor changes the contents of the PC. The processor loads PC bits 4–31 with the 28-bit PC offset specified in the gate. Bits 1–3 of the PC specify the number of the segment that contains the gate array.

> **NOTE:** *If the maximum number of gates is 0, then the segment containing the gate array can not be the target of an inward ring crossing.*

The **WRTN** or the **WPOPB** instructions are used for outward returns. The processor interprets the address specified by either of these instructions as a normal word address. This means that the processor does not need gates or gate arrays to return control to the outer segment.

Transferring Control to the New Segment

When a valid ring crossing is specified, the processor references a gate and reloads the PC. However, before control can transfer from the old segment to the new, the processor must save the source segment's state, create a new wide stack for the destination segment, and copy any parameters onto the new wide stack. For more information about MV/8000 wide stacks and how the processor manipulates them during a ring crossing, see Chapter 10.

Copying Parameters

Once the new wide stack in the destination segment has been created, the processor checks for potential stack overflow. The number of parameters to be copied is specified by the **LCALL** or **XCALL** instruction that initiated the ring crossing. The processor uses this information to determine if the number of parameters to copy exceeds the size of the wide stack.

If copying the parameters would cause a stack overflow, the processor does not copy the parameters and a stack fault occurs. AC1 will contain 2. Note that since the ring crossing is valid, the stack fault occurs in the destination segment, not the source segment. The PC contains the address specified in the gate array entry; this is the address of the first instruction to be executed in the destination segment.

If copying the parameters would not cause a stack overflow, the processor copies the parameters from the source wide stack to the destination wide stack. The order of the arguments in the destination wide stack matches the order of the arguments in the source wide stack. This means that the parameters are referenced exactly as if no ring crossing had occurred.

Finishing Up

After copying all parameters, the processor pushes a double-word onto the top of the destination wide stack. This double-word contains the PSR and the number of arguments pushed.

The processor completes the ring crossing process by executing the instruction specified by the PC. As described above, this address is specified by the gate array entry. Normally, the instruction specified by the PC is a **WSAVR** or **WSAVS**. Either of these pushes a return block onto the newly defined stack, and loads WFP with the updated value of WSP.

> **NOTE:** *All the actions necessary for ring crossing are the result of the execution of an* **LCALL** *or an* **XCALL** *instruction. The processor performs these functions without software assistance.*

Trojan Horse Pointers

Suppose a program in Segment 6 calls a subroutine in Segment 2, and suppose one of the arguments passed to Segment 2 is a pointer to information in Segment 2. When execution in Segment 2 uses this pointer to make a reference, the processor will use Segment 2's access privileges to determine the validity of the reference. *These access privileges may allow references to data that would otherwise be restricted.* A pointer that attempts such access is called a *Trojan Horse* pointer.

Argument Validation/Physical Address Instructions

The *Validate Word Pointer* and *Validate Byte Pointer* instructions check for Trojan Horse pointers. Table 4.2 describes these instructions; also included is an instruction that translates a logical address to a physical address.

MNEM	Name	Action
VWP	Validate Word Pointer	Checks a word pointer for validity and skips depending on the outcome of the check.
VBP	Validate Byte Pointer	Checks a byte pointer for validity and skips depending on the outcome of the check.
LPHY	Load Physical	Translates a logical address to a physical address.

Table 4.2 Argument validation/physical address instructions

Note that instructions such as **WCMT** and **WCMV** can move data in descending order. When this occurs, the instructions check to make sure that the segment field of the source data does not change during the move.

Indirection Protection

Each time an indirect address is specified, the processor must check for a valid ring crossing. When a memory reference is made for the first time, the processor treats the segment specified in the PC as the source segment, and the segment specified in the intermediate address as the destination segment. The processor compares these two fields. If the access is invalid, then a protection fault occurs, and AC1 will contain a 4.

If the access is valid, then the processor fetches the new address specified by the intermediate address. If further indirection is to occur, the segment specified in the intermediate address becomes the source segment, and the segment specified in the new address becomes the destination segment.

Up a maximum of 15 levels of indirection can be specified. If more than 15 are specified, a protection fault occurs and AC1 will contain the code 5. Indirection protection is always enabled whenever the processor is operating with the ATU enabled.

If an instruction can resolve two indirection chains (such as **WBLM**) then the *total* number of indirect references between the two chains can not be more than 16.

Page Protection

The previous section described process-wide protection between segments. Page protection checks to see if the process-wide reference can legally access a particular page in the segment. There are two types of page protection: page table entry validation, and access validation.

Page Table Entry Validation

As mentioned earlier, bit 0 of the PTE (the validity bit) indicates whether the page is defined. A 1 in bit 0 specifies a defined (and therefore valid) page; a 0 specifies an invalid (non-defined) page. Bit 1 of the PTE specifies if the page referenced by the PTE is currently in the physical memory allocated to the program. The page is in memory if bit 1 contains 1. If this bit contains a 0, the page is located in a secondary storage device (e.g. on a disc) and must be fetched into memory. Referencing a non-resident page causes a page fault. For more information about this fault, see the section entitled *Page Faults* at the end of this chapter.

Access Validation

When a referenced page is in physical memory, the processor determines whether the page is restricted to a particular access. Bits 2–4 of the referenced PTE contain the access bits that specify any restriction.

When the reference to memory is a read, the processor checks bit 2. A 1 in this bit indicates a valid read; a zero indicates an invalid read. If the reference is invalid, a protection fault occurs and AC1 contains the code 0.

> **NOTE:** *When executing an instruction that specifies an immediate field, the processor will not interpret the read access bit of the page containing the immediate field.*

Bit 3 determines if any write reference to the page is valid. A 1 in this bit indicates a valid write; a zero indicates an invalid write. An invalid reference causes a protection fault, and AC1 contains the code 1.

Bit 4 governs the transfer of control from one location to another (execute). A 1 in this bit indicates validity; a zero indicates invalidity. When the reference is invalid a protection fault occurs and AC1 contains the code 2.

Protection Faults

As mentioned throughout this chapter, the address translation process and subsequent references or control transfers can initiate several kinds of protection faults. When a fault occurs, the processor takes suitable actions to correct whatever caused the fault, if possible, or to transfer control to a protection fault handler.

Priority of Protection Faults

Because a reference could produce multiple protection violations, the MV/8000 imposes priorities on the protection faults. This means that only the highest priority fault is acted upon. Lower priority faults are suppressed, and thus they do not result in additional protection faults. Table 4.3 shows the priority of the protection faults. A level of 0 means that the fault has the highest priority.

Level of Priority	Type of Protection Fault
0	Privileged or I/O instruction violation
1	Defer (indirection) violation
2	Inward reference violation
3	Segment validity violation
4	Page table validity violation
5	Read, write, or execute access violation
6	Ring crossing violation

Table 4.3 Priority of MV/8000 protection faults

Servicing a Protection Fault

When a protection fault occurs, the processor first stores the WSP and the WFP in the appropriate page zero locations of the current segment. The processor then performs a ring crossing to Segment 0 (if Segment 0 is not the current segment).

After the ring crossing, the processor pushes a wide return block onto the Segment 0 stack. The PC in the return block points to the instruction that would have been executed next had the protection fault not occurred. AC0 contains the address of the instruction causing the fault, and AC1 contains a code indicating the kind of protection fault. The processor then checks for stack overflow. If overflow has occurred, the processor pushes an additional return block onto the stack and a stack fault occurs. (Note that this second return block contains the values of the PC, carry, and accumulators as they were during the initial processing of the first fault.) AC1 will contain the value 4. Processing continues with the stack fault handler.

If overflow has not occurred, the processor sets *OVK*, *OVR*, and *IRES* to 0, and loads AC1 with the correct fault code. A jump via page zero location 36_8 transfers control to the protection fault handler. Location 36_8 contains a 16-bit pointer to the fault handler.

Note that interrupts are not handled before the first instruction of the protection fault handler has executed.

If a protection fault occurs while any other type of fault is being serviced, the processor aborts service of the first fault. The return block pushed onto the stack for the protection fault is undefined, as are the contents of AC0 and AC1. After servicing the protection fault, the processor loads AC0 and AC1 with the contents appropriate to the fault.

The protection fault codes placed in AC1 and their meanings are summarized in Table 4.4. For more information, refer to the preceding sections of this chapter.

Code	Meaning	Explanation
0	Read violation	Bit 2 of the specified PTE contains a 0.
1	Write violation	Bit 3 of the specified PTE contains a 0.
2	Execute violation	Bit 4 of the specified PTE contains a 0.
3	Validity violation (SBR or PTE)	Bit 0 of the specified SBR or PTE contains a 0.
4	Inward address reference	Attempted access to a location in an inner ring.
5	Defer (indirect) violation	More than 15 levels of indirection specified.
6	Illegal gate	Gate number specified in an inward call is greater than or equal to the maximum number of gates; or the gate's PC offset is -1.
7	Outward call	Attempted transfer of control from the current ring to another ring with an outward subroutine call.
8	Inward return	Attempted transfer of control from the current ring to another ring with an inward return from a subroutine.
9	Privileged instruction violation	Attempted use of a privileged instruction in a segment other than Segment 0.
10	I/O protection violation	Attempted use of an I/O instruction when bit 3 of the current segment's SBR is set to 0.
11	—	Reserved for future use.
12	—	Reserved for future use.

Table 4.4 MV/8000 protection fault codes

Page Faults

A page fault occurs when:

- A reference is made to a page that is part of the logical address space, but not part of the system's physical address space;
- An attempt is made to translate a logical address using a two level page table when only a one level page table has been specified.

When a page fault occurs, the processor stores a copy of the *context block* into memory. The context block contains information about the status of the machine. Locations $32–33_8$ of Segment 0 specify the address in memory where the first double-word of the context block will be stored. The processor stores the remainder of the context block in the double-word locations that follow.

The context block contains a code that specifies the type of page fault that occurred. The code is located in words 18–19 of the context block and specifies one of three types of page faults:

- Page table depth,
- Page fault when referencing a page table,
- Page fault when referencing an object page.

For more information about the format of the context block, refer to Appendix B.

After storing the context block, the processor performs a ring crossing to Segment 0 if the current segment is not Segment 0, then jumps indirectly through locations $30\text{–}31_8$ of Segment 0 to the MV/8000 page fault handler.

If a page fault occurs while the processor is storing the context block, crossing to Segment 0, or transferring to the page fault handler, the processor halts and a message is displayed at the operator's terminal. For more information, refer to *The ECLIPSE MV/8000 System Control Processor*, Operator's Manual, DGC. No. 014-000649.

Chapter 5
Reserved Memory Locations and Faults

As described in the previous chapter, control transfers to an appropriate fault routine when a protection violation occurs. To transfer to these fault routines, the processor determines the starting address of the desired handler then and transfers control to that location. To do this, the processor references a *reserved storage location*, which contains the starting address of the fault handler.

Each segment has a set of reserved storage locations. This chapter describes these locations for all segments, and also lists the locations that apply when the C/350 MAP is enabled.

Page Zero

Each segment has a *page zero* that occupies the first 377_8 words.

Page Zero Locations for Segment 0

When an MV/8000-specific interrupt occurs, locations 0–47_8 have the meanings shown in Table 5.1. For information on MV/8000-specific interrupts, refer to Chapter 14.

When the MV/8000 ATU is enabled, the processor interprets all locations as logical.

When the C/350 MAP is enabled, the processor interprets locations 0,1,2, and 3 as physical locations (see Table 5.2). Locations 4,5,6, and 7 may be interpreted as either physical or logical as determined by the state of the C/350 MAP. Locations 40–47_8 also have new meanings.

Note that some of the pointers desribed in Tables 5.1 and 5.2 are 16 bits long. This means that they can reference locations in the first 64 Kbytes only of the segment that contains the pointer. If the pointer is indirect, all pointers in the indirect chain can also reference only the first 64 Kbytes of the segment.

Word	Name	Function
0	Interrupt level	Level of interrupt processing. Zero indicates base level processing; Non-zero indicates intermediate level processing.
1	I/O handler	Address of the I/O interrupt handler. Indirectable.
2	I/O return address MV/8000, higher bits	Higher order bits of the MV/8000 I/O interrupt return.
3	I/O return address MV/8000, lower order bits	Lower order bits of the MV/8000 I/O interrupt return.
4	Vector stack pointer	This is a 16-bit word offset. Lower order 16 bits of vector stack pointer, base and frame. High order bits are zero.
5	---	
6	Vector stack limit	This is a 16-bit word. Lower order 16 bits of vector stack limit.
7	Vector stack fault address	Address of the vector stack fault handler. Indirectable.
10-11	MV/8000 breakpoint address	Address of the breakpoint handler. Indirectable
12-13	MV/8000 XOP origin address	Address of the beginning of the MV/8000 extended operations table. Non-indirectable.
14	MV/8000 stack fault address	Address of the MV/8000 stack fault handler. Indirectable.
15-17	Reserved	Reserved.
20-21	WFP	MV/8000 frame pointer. Non-indirectable.
22-23	WSP	MV/8000 stack pointer. Non-indirectable.
24-25	WSL	MV/8000 stack limit. Non-indirectable.
26-27	WSB	MV/8000 stack base. Non-indirectable.
30-31	MV/8000 page fault handler	Address of the MV/8000 page fault handler. Indirectable.
32-33	MV/8000 context block pointer	Address of the base of context block save area. Indirectable.
34-35	WGP	MV/8000 gate pointer. Address of the gate array. Non-indirectable.
36	Protection fault handler address	Address of the MV/8000 protection fault handler. Indirectable.
37	Fixed point fault handler address	Address of MV/8000 fixed point fault handler. Indirectable. Indirectable.
40	Stack pointer	Address of the top of the C/350 stack. Non-indirectable.
41	Frame pointer	Address of the start of the current C/350 frame minus 1. Non-indirectable.
42	Stack limit	Address of the last normally usable location in the C/350 stack.
43	C/350 stack fault address	Address of the C/350 stack fault handler. Indirectable.
44	XOP origin address	Address of the beginning of the C/350 extended operations table.
45	Floating-point fault address	Address of the floating point fault handler. Indirectable.
46	Commercial fault address	Address of the commercial fault handler. Indirectable.
47	Reserved	

Table 5.1 Page zero locations for Segment 0, ATU enabled

Word	Name	Function
0	I/O return address	Return address used by C/350 programs during I/O interrupts.
1	I/O handler address	Address of the I/O interrupt handler. Indirectable.
2	SC handler address	Address of the C/350 *System Call* instruction. Indirectable.
3	Protection fault handler address	Address of the C/350 protection fault handler. Indirectable.
5	Current C/350 mask	Current interrupt priority mask.
40	Stack pointer	Address of the top of the C/350 stack. Non-indirectable.
41	Frame pointer	Address of the start of the current C/350 frame minus 1. Non-indirectable.
42	Stack limit	Address of the last normally usable location in the C/350 stack.
43	C/350 stack fault address	Address of the C/350 stack fault handler. Indirectable.
44	XOP origin address	Address of the beginning of the C/350 extended operations table.
45	Floating-point fault address	Address of the floating point fault handler. Indirectable.
46	Commercial fault address	Address of the commercial fault handler. Indirectable.
47	Reserved	

Table 5.2 Page zero locations for Segmet 0, C/350 MAP enabled

Page Zero Locations for Segments 1–7

Page zero locations $0\text{–}47_8$ for Segments 1–7 are shown in Table 5.3.

When the C/350 MAP is enabled, the processor interprets the page zero locations as shown in Table 5.4.

Location	Name	Function
0–7	Reserved	—
10–11	MV/8000 breakpoint address	Address of the breakpoint handler. Indirectable.
12–13	MV/8000 XOP origin address	Address of the beginning of the MV/8000 extended operations table. Non-indirectable.
14	MV/8000 stack fault address	Address of the MV/8000 stack fault handler. Indirectable.
15–17	Reserved	
20–21	WFP	MV/8000 frame pointer. Non-indirectable.
22–23	WSP	MV/8000 stack pointer. Non-indirectable.
24–25	WSL	MV/8000 stack limit. Non-indirectable.
26–27	WSB	MV/8000 stack base. Non-indirectable.
30–31	Reserved	—
32–33	Reserved	—
34–35	EGP	MV/8000 gate pointer. Address of gate array. Non-indirectable.
36	Reserved	—
37	Fixed point fault handler address	Address of MV/8000 fixed point fault handler. Indirectable.
40	Stack pointer	Address of the top of the C/350 stack. Non-indirectable.
41	Frame pointer	Address of the start of the current C/350 frame minus 1. Non-indirectable.
42	Stack limit	Address of the last normally usable location in the C/350 stack.
43	C/350 stack fault address	Address of the C/350 stack fault handler. Indirectable.
44	XOP origin address	Address of the beginning of the C/350 extended operations table.
45	Floating-point fault address	Address of the faulting point handler. Indirectable.
46	Commercial fault address	Address of the commercial fault handler. Indirectable.
47	Reserved	Address of DERR handler

Table 5.3 Page zero locations for Segments 1-7, ATU enabled

Location	Name	Function
40	Stack pointer	Address of the top of the C/350 stack. Non-indirectable.
41	Frame pointer	Address of the start of the current C/350 frame minus 1. Non-indirectable.
42	Stack limit	Address of the last normally usable location in the C/350 stack.
43	C/350 stack fault address	Address of the C/350 stack fault handler. Indirectable.
44	XOP origin address	Address of the beginning of the C/350 extended operations table.
45	Floating-point fault address	Address of the faulting point handler. Indirectable.
46	Commercial fault address	Address of the commercial fault handler. Indirectable.
47	Reserved	—

Table 5.4 Page zero locations for Segments 1-7, C/350 MAP enabled

Chapter 6
Data Types

As mentioned in Chapter 2, the MV/8000 manipulates several types of data:

- Fixed point data,
- Floating point data,
- Character strings,
- Bits,
- Commercial data.

This chapter describes the registers used to manipulate each group of data, the formats associated with each group, and the range of magnitudes possible with each group. First, however, two registers that are used to store system information, the *program counter* and the *processor status register*, are discussed.

Program Counter

The program counter is 31 bits long. Bits 1–3 specify the current segment of execution. Bits 4–31 specify an instruction address. The 16-bit PC used by C/350 programs is mapped into bits 17–31 of the PC.

When the processor increments the PC to reference the next sequential instruction, only bits 4–31 take part in the increment; this means that every address will remain within the current segment. Instructions exist to change the ring of execution; for more information, refer to Chapter 4.

Processor Status Register

The processor status register (PSR) contains information about the state of the MV/8000. The table shown below describes the information contained in this register.

BITS	NAME	CONTENTS or FUNCTION
0	OVK	Overflow Mask. If this bit is set, a 1 in bit 1 will result in a fixed-point trap.
1	OVR	Fixed-point Overflow Indicator.
2	IRES	Interrupt Resume.
3-15	—	Reserved.

Format of the PSR

Table 6.1 lists the MV/8000-specific instructions that affect the contents of the PSR.

MNEM	Name	Action
LPSR	Load processor status register	Loads the *OVK*, *OVR*, and *IRES* bits into AC0.
SPSR	Store processor status register	Loads bits 0-2 of AC0 into the *OVK*, *OVR*, and *IRES* flags in the PSR.
SNOVR	Skip on *OVR* reset	Tests the *OVR* flag and skips the next 16-bit word if *OVR* is 0.

Table 6.1 PSR instructions

Fixed Point Data

Fixed point values are signed integers, unsigned integers or logical (bit string) quantities. The processor uses four fixed point accumulators to manipulate these values.

Fixed Point Accumulators

Each fixed point accumulator is 32 bits wide and can contain either a fixed point integer, a logical quantity, or an address.

When executing an MV/8000-specific instruction, the processor sign-extends all 16-bit integers located in memory to 32 bits before loading them into a 32-bit accumulator. The processor zero-extends all 8-bit bytes located in memory to 32 bits before loading them into a 32-bit accumulator.

Fixed Point Format

Fixed point numbers are signed and unsigned integers. A signed integer uses two's complement representation to distinguish between positive and negative values. A positive integer contains a 0 in its bit 0; a negative integer contains a 1 in its bit 0.

The MV/8000 manipulates fixed point integers in units of 16 and 32 bits. Table 6.2 shows the possible range of 16-bit and 32-bit integers represented by this format.

	16-Bit Integers	32-Bit Integers
Unsigned	0 to 65,535	0 to 4,294,967,295
Signed	-32,768 to +32,767	-2,147,483,648 to +2,147,483,647

Table 6.2 Range of 16-bit and 32-bit integers

Carry

Sometimes the arithmetic logic unit (ALU) produces a result that is larger than the data size specified in the executed instruction. When this occurs, the processor sets the value of *carry* to indicate that the result just calculated was too large. Carry can have a value of 0 or 1. A value of 1 usually indicates that the ALU result is too large. Note, however, that instructions exist to set carry to the appropriate value. This allows carry to be used as an indicator.

The instructions used to explicitly set carry are shown in Table 6.3.

MNEM	Name	Action
CRYTC	Complement carry	Complements the value of carry.
CRYTO	Set carry to one	Sets the value of carry to one.
CRYTZ	Set carry to zero	Sets the value of carry to zero.

Table 6.3 Carry instructions

C/350 Compatability

The 16-bit C/350 accumulators correspond to bits 16–31 of the wide accumulators. That is, C/350 instructions load and store bits 16–31 of the wide accumulators, generally leaving bits 0–15 undefined.

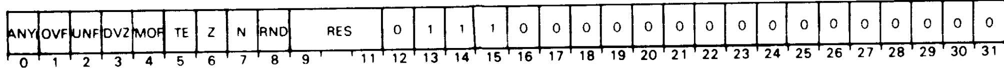
C/350 program flow instructions leave bits 1–3 of the wide PC unchanged, and set bits 4–16 to 0. They load bits 17–31 with the appropriate address. For more information about specific instructions, refer to the individual instruction descriptions found in the MV/8000 Instruction Dictionary, Chapter 16.

Floating Point Data

Floating point numbers allow the use of very large numbers and fractions. Floating point numbers can be as large as a 16-word multiple precision integer, and as small as 1/16 x 16^{-64}.

Floating Point Registers

The floating point processor provides five registers accessible to the programmer. The four floating point registers are called *FPACs*and are numbered FPAC0, FPAC1, FPAC2, and FPAC3. Each FPAC is 64 bits wide. The fifth register is called the *floating point status register* (FPSR). The FPSR is 64 bits wide and contains information about the current status of the floating point processor. The format of the FPSR is shown below.



BITS	NAME	CONTENTS or FUNCTION
0	ANY	The value of this bit is the logical OR of FPSR bits 1–4.
1	OVF	Overflow indicator. While processing a floating point number, an exponent overflow occurred; the result is correct except the exponent is 128 too small.
2	UNF	Underflow indicator. While processing a floating point number, an exponent underflow occurred; the result is correct except the exponent is 128 too large.
3	DVZ	Divide by zero. While processing a floating point number, a zero divisor was detected; division was aborted and the operands remain unchanged.
4	MOF	Mantissa overflow. During a **FSCAL** instruction a significant bit was shifted out of the high order end of the mantissa. During a **FFAS, FFMD**, or **WFFAD** instruction the result did not fit into the destination location.
5	TE	Trap enable. If this bit is 1, a 1 in any of bits 1–4 will result in a floating point trap, except where noted.
6	Z	Zero bit. The result of the last floating-point operation was true zero.
7	N	Negative bit. The result of the last floating-point operation was less than zero.
8	RND	Rounding bit. Indicates whether unbiased rounding or truncation is used during floating point arithmetics. 0 indicates truncation; 1 indicates unbiased rounding.
9	RES	Indicates an interrupt has occurred during execution of a resumable instruction.
10–11	—	Reserved for future use.
12–15	FPMOD	MV/8000 floating point ID code. Must be 0111.
16–32	—	Reserved.
33–63	FPPC	Floating point program counter. This is the logical address of the first floating point instruction that caused a fault.

Format of the floating point status register

Note that all reserved fields in the FPSR must be zero.

Floating Point Format

Floating point numbers occupy either four bytes (single precision) or eight bytes (double precision). The format of single and double precision floating point numbers is shown in Figure 6.1.

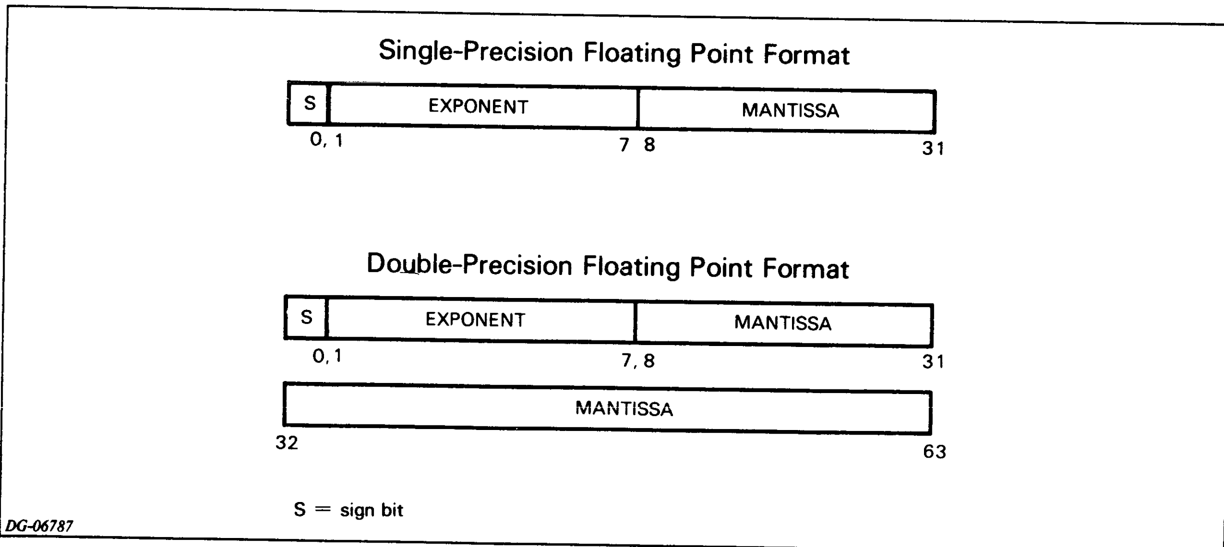


Figure 6.1 Floating point format

Sign

Bit 0 of the first byte is the sign bit. If the sign bit is 0, the number is positive. If the sign bit is 1, the number is negative. Numbers using this form for sign representation are referred to as "sign and magnitude".

Exponent

Bits 1–7 of the first byte contain the exponent. All exponents are represented in *excess 64* representation. This means that the value represented in bits 1–7 of the number is 64 greater than the true value of the exponent. Table 6.4 illustrates this.

Exponent Field	True Value of Exponent
0	-64
64	0
127	63

Table 6.4 Excess 64 Representation

Mantissa

The mantissa is an unsigned fraction. The mantissa of a single precision number occupies bytes 2–4; the mantissa of a double precision number occupies bytes 2–8. The binary point lies *to the left of* the first bit of the mantissa.

All single precision operations that specify an accumulator fetch the most significant 32 bits of the FPAC and ignore the least significant 32 bits. Upon completion of the specified operation, the processor returns the result to the most significant portion of the FPAC. The processor loads the least significant 32 bits of the FPAC with zeros.

C/350 Compatability

The C/350 floating point registers are identical to those of the MV/8000. The formats of MV/8000 and C/350 floating point numbers are also identical.

Commercial Data

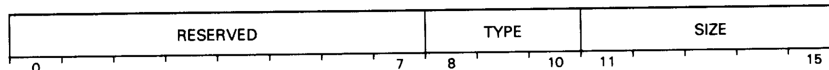
Commercial Registers

The MV/8000 uses the fixed point and floating point registers when manipulating commercial data.

Data Type Indicator

Most MV/8000 instructions explicitly specify the format of data in memory. For example, the *Load Byte* instruction loads a single byte of information, and the *Block Move* instruction moves several bytes of information. Likewise, the *Load Floating Point Double* instruction loads an eight-byte aggregate of data.

The commercial instructions, however, do not make such explicit specifications. A parameter must be passed to these instructions that defines the type and precision of the number to be manipulated. This parameter, the *data-type indicator*, defines the data representation the instruction is to use and the size of the number. The data-type indicator (which is passed to the instruction in an accumulator) has the following format:



BITS	NAME	CONTENTS or FUNCTION
0-7	---	Reserved for future use.
8-10	TYPE	Data type: 0 Unpacked decimal, low-order sign. 1 Unpacked decimal, high-order sign. 2 Unpacked decimal, trailing sign. 3 Unpacked decimal, leading sign. 4 Unpacked decimal, unsigned. 5 Packed decimal. 6 Two's complement integer, byte aligned. 7 Floating point, byte aligned.
11-15	SIZE	Data length: For all except data type 5, count of bytes in number *minus 1* (including sign); For data type 5, the count of *digits* in the number.

Format of the data type indicator

Commercial Formats

The two basic formats used to represent commercial data are *packed decimal* and *unpacked decimal.*

Unpacked Decimal

In this format, one byte contains the code for one ASCII character. Each ASCII character represents one decimal digit. Table 6.5 shows the ASCII characters used to represent the combination of a digit and sign.

Digit	Digit With + Sign ASCII Character	Octal Code	Digit With - Sign ASCII Character	Octal Code
0	{	173	}	175
1	A	101	J	112
2	B	102	K	113
3	C	103	L	114
4	D	104	M	115
5	E	105	N	116
6	F	106	O	117
7	G	107	P	120
8	H	110	Q	121
9	I	111	R	122

Table 6.5 ASCII character sign/digit combinations

There are four ways to represent the sign of an unpacked decimal number, as illustrated in Figure 6.2. The first two columns name and describe the sign representation. The rest of the columns show an example. In each example, the first column shows the normal representation of the number. The second column shows the ASCII characters placed in each byte. The third column shows the octal code of each byte.

Type	Characteristic	Example: Normal representation	Example: ASCII characters in each byte	Example: Octal code in each byte
Leading sign	Sign appears in separate byte before number.	+2048	+ 2 0 4 8	053 062 060 064 070
Trailing sign	Sign appears in separate byte after number.	-1756	1 7 5 6 —	061 067 065 060 055
High-order sign	First byte indicates sign and high-order byte.	+1850	A 8 5 0	101 070 065 060
Low-order sign	Last byte indicates sign and low-order byte.	-3972	3 9 7 K	063 071 067 113

DG-06935

Figure 6.2

The tables below show the ASCII characters that can be used to represent legal unpacked decimal numbers. Table 6.6 lists the characters that can represent a digit-sign combination of data types 0 and 1. Table 6.7 shows the equivalent information for data types 2 and 3. Table 6.8 shows the characters that can represent decimal digits of data types 0 through 4.

Character	Octal Code	Numerical Value	Sign	Character	Octal Code	Numerical Value	Sign
blank	040	0	+	E	105	5	+
+	053	0	+	F	106	6	+
{	173	0	+	G	107	7	+
0	060	0	+	H	110	8	+
1	061	1	+	I	111	9	+
2	062	2	+	-	055	0	-
3	063	3	+	}	175	0	-
4	064	4	+	J	112	1	-
5	065	5	+	K	113	2	-
6	066	6	+	L	114	3	-
7	067	7	+	M	115	4	-
8	070	8	+	N	116	5	-
9	071	9	+	O	117	6	-
A	101	1	+	P	120	7	-
B	102	2	+	Q	121	8	-
C	103	3	+	R	122	9	-
D	104	4	+				

Table 6.6 Valid characters for data types 0 and 1; sign position

Character	Octal Code	Numerical Value	Sign
+	053	None	+
-	055	None	-

Table 6.7 Valid characters for data types 2 and 3; sign position

Character	Octal Code	Numerical Value
Blank	040	0
0	060	0
1	061	1
2	062	2
3	063	3
4	064	4
5	065	5
6	066	6
7	067	7
8	070	8
9	071	9

Table 6.8 Valid characters for data types 0, 1, 2, 3, and 4; non-sign position

Packed Decimal
In this format, each decimal digit occupies one half byte (four bits) in memory. The sign appears in a separate trailing half byte. The number must start and end on a byte boundary.

Note that a packed decimal number always consists of an odd number of digits followed by the sign. A zero is placed in front of numbers with an even number of digits.

Numbers represented in packed decimal format use the octal codes 14 or 17 to represent a plus sign; 15, a negative sign. To represent a decimal digit, packed decimal numbers use the octal codes 000 through 011 to represent the decimal digits 0 through 9.

Several examples of packed decimal numbers are shown below.

Example	Decimal digit in each half byte						Binary code in each half byte					
+2048	0	2	0	4	8	+	0000	0010	0000	0100	1000	1100
+32,456	3	2	4	5	6	+	0011	0010	0100	0101	0110	1100
-1756	0	1	7	5	6	-	0000	0001	0111	0101	0110	1101
-25,989	2	5	9	8	9	-	0010	0101	1001	1000	1001	1101

Figure 6.3

Character Strings

Character strings use the same format as decimal data (see the preceding section). The MV/8000 uses the fixed point registers when manipulating character strings.

Data Manipulation

The MV/8000 has many sets of instructions that manipulate all of the data types described in the preceding sections. These instructions are described in detail in the next four chapters.

Chapter 7
Fixed Point Instructions

The last chapter described the format of fixed point numbers and the registers used in manipulating them. This chapter introduces the instructions used to perform operations on fixed point data. Chapter 16, the MV/8000 Instruction Dictionary, discusses each instruction in detail. Appendix C describes the generic properties of MV/8000-specific and C/350 instructions and discusses programs using both types of instructions.

Because the MV/8000 has many fixed point instructions, they are divided into categories based on the variety of functions and data lengths. These categories are:

- Indexed address instructions
- Memory to accumulator instructions
- Bit manipulating instructions
- Byte manipulating instructions
- Single-word arithmetics
- Double-word arithmetics
- Single-word logicals
- Double-word logicals
- Single-word compares
- Double-word compares
- ALC instructions

Conventions

The following mnemonic conventions are used to distinguish between instructions with similar semantics:

X Extended displacement (one word)

L Long displacement (two words)

N Narrow data (16 bits)

W Wide data (32 bits)

Fixed Point Indexed Address Instructions

Indexed address instructions either load, store, or modify data. The displacements specified by these instructions can be 15 or 31 bits long (word displacements), or 16 or 32 bits long (byte displacements). These instructions can reference data that is 1, 2, or 4 bytes long.

MNEM	Action
LLDB	Calculates a byte pointer. Loads the addressed byte into bits 24–31 of the specified accumulator.
LLEF	Calculates an effective address and loads it into the specified accumulator.
LLEFB	Calculates an effective byte address and loads it into the specified accumulator.
LNDSZ	Calculates an effective address. Decrements the 16-bit contents of the specified memory location by one and skips the next 16-bit word if the decremented value is zero.
LNISZ	Calculates an effective address. Increments the 16-bit contents of the addressed memory location by one and skips the next 16-bit word if the incremented value is zero.
LNLDA	Calculates an effective address. Loads the 16-bit contents of the addressed location into bits 16–31 of the specified accumulator, and sign extends bits 0–15.
LNSTA	Calculates an effective address. Stores bits 16–31 of the specified accumulator into the addressed memory location.
LPEF	Calculates an effective address and pushes it onto the wide stack.
LPEFB	Calculates an effective byte address and pushes it onto the wide stack.
LSTB	Calculates a byte pointer. Stores the byte contained in bits 24–31 of the specified accumulator into the addressed byte in memory.
LWDSZ	Calculates an effective address. Decrements the 32-bit contents of the addressed location and skips the next 16-bit word if the decremented value is zero.
LWISZ	Calculates an effective address. Increments the 32-bit contents of the addressed location and skips the next 16-bit word if the incremented value is zero.
LWLDA	Calculates an effective address. Loads the 32-bit contents of the addressed location into the specified accumulator.
LWSTA	Calculates an effective address. Stores the 32-bit contents of the specified accumulator into the location addressed by the effective address.
XLDB	Calculates a byte pointer. Loads the addressed byte into bits 24–31 of the specified accumulator.
XLEF	Calculates an effective address and loads it into bits 16–31 of the specified accumulator.
XLEFB	Calculates an effective byte address and loads it into the specified accumulator.
XNDSZ	Calculates an effective address. Decrements the 16-bit contents of the addressed location by one and skips the next 16-bit word if the decremented value is zero.
XNISZ	Calculates an effective address. Increments the 16-bit contents of the addressed location by one and skips the next 16-bit word if the incremented value is zero.
XNLDA	Calculates an effective address. Loads the 16-bit contents of the addressed location into the specified accumulator. Sign Extend
XNSTA	Calculates an effective address. Stores the contents of the specified accumulator into the location addressed by the effective address.
XPEF	Calculates an effective address and pushes it onto the wide stack.
XSTB	Calculates a byte pointer. Stores the byte contained in bits 24–31 of the specified accumulator into the addressed byte in memory.
XWDSZ	Calculates an effective address. Decrements the 32-bit contents of the addressed location by one and skips the next 16-bit word if the decremented value is zero.
XWISZ	Calculates an effective address. Increments the 32-bit contents of the specified location by one and skips the next 16-bit word if the incremented value is zero.
XWLDA	Calculates an effective address. Loads the 32-bit contents of the location into the specified accumulator.
XWSTA	Calculates an effective address. Stores the 32-bit contents of the specified accumulator into the location addressed by the effective address.

Table 7.1 Fixed point indexed address instructions

Fixed Point Single-Word Indexed Arithmetic Instructions

These instructions perform arithmetic operations between 16-bit integers located in memory and 16-bit integers located in an accumulator. Each instruction specifies a displacement and index bits that are used to calculate the address of the integer in memory. The integer in memory is fetched and used in the specified operation. The accumulator specified in the instruction supplies the second operand and stores the result of the operation.

Table 7.2 lists the instructions of this type. Those with **X** in the mnemonic use 15-bit displacements to calculate the address of the integer in memory. Those with **L** in the mnemonic use 31-bit displacements.

MNEM	Action
LNADD, XNADD	Narrow add with memory word
LNSUB, XNSUB	Narrow subtract with memory word
LNMUL, XNMUL	Narrow multiply with memory word
LNDIV, XNDIV	Narrow divide with memory word

Table 7.2 Fixed point single-word indexed address arithmetic instructions

Fixed Point Double-Word Indexed Arithmetic Instructions

These instructions perform arithmetic operations between 32-bit integers located in memory and 32-bit integers located in an accumulator. Each instruction specifies a displacement and index bits that are used to calculate the address of the integer in memory. The integer in memory is fetched and used in the specified operation. The accumulator specified in the instructrion supplies the second operand and stores the result of the operation.

Table 7.3 lists the instructions of this type. Those with **X** in the mnemonic use 15-bit displacements to calculate the address of the integer in memory. Those with **L** in the mnemonic use 31-bit displacements.

MNEM	Action
LWADD, XWADD	Wide add with memory word
LWSUB, XWSUB	Wide subtract with memory word
LWMUL, XWMUL	Wide multiply with memory word
LWDIV, XWDIV	Wide divide with memory word

Table 7.3 Fixed point double-word indexed address arithmetic instructions

Memory to Accumulator Instructions

Instructions of this type move data between memory and an accumulator.

MNEM	Name	Action
LDATS	Load accumulator with contents pointed to by WSP	Loads the contents of the word addressed by WSP into the specified accumulator.
LNLDA, LWLDA	Long load Accumulator	Loads the specified accumulator with the contents of a location in memory.
LNSTA, LWSTA	Long store Accumulator	Stores the contents of the specified accumulator into memory.
NLDAI, WLDAI	Load accumulator immediate	Loads the specified accumulator with the contents of an immediate value.
STATS	Store accumulator at location pointed to by WSP	Stores the contents of the specified accumulator into the location addressed by WSP.
XNLDA, XWLDA	Extended Load accumulator	Loads the specified accumulator with the contents of a location in memory.
XNSTA, XWSTA	Extended Store accumulator	Stores the contents of the specified accumulator into memory.

Table 7.4 Memory to accumulator instructions

C/350 Word Memory to Accumulator Instructions

Instructions of this type move data between memory and an accumulator. Note that bits 0–15 of the 32-bit accumulator will be indeterminate after the word of data is loaded into bits 16–31.

MNEM	Name	Action
LDA, ELDA	Load Accumulator	Loads the specified accumulator with the contents of a (16-bit address) location in memory.
STA, ESTA	Store accumulator	Stores the contents of the specified accumulator into memory.

Table 7.5 C/350 Memory word to accumulator instructions

Bit Manipulation Instructions

As described in Chapter 2, a bit pointer is necessary to address a bit in memory. This bit pointer is made up of a word pointer and a bit offset. The word pointer is 32 bits long and points to a word in memory. The bit offset is also a 32-bit value; it specifies the bit location of the referenced bit relative to the base location specified by the word pointer.

Table 7.6 lists the MV/8000-specific bit manipulation instructions that require bit pointers. Table 7.7 lists the MV/8000-specific instructions that test or affect bits in an accumulator. Table 7.8 lists the privileged instructions that affect the referenced and modified bits.

MNEM	Name	Action
WBTO	Set bit to one	Sets the bit addressed by ACS and ACD to 1.
WBTZ	Set bit to zero	Sets the bit addressed by ACS and ACD to 0.

Table 7.6 MV/8000 bit manipulation instructions

MNEM	Name	Action
NSALA WSALA	Skip on all bits set in accumulator	Performs a logical AND on the immediate field and a specified accumulator and skips if the result of the AND is zero.
NSALM WSALM	Skip on all bits set in memory location	Performs a logical AND on the immediate field and a specified memory location and skips if the result of the AND is zero.
NSANA	Skip on any	Performs a logical AND on the
WSANA	bit set in accumulator	immediate field and a specified accumulator and skips if the result of the AND is nonzero.
NSANM WSANM	Skip on any bit set in memory location	Performs a logical AND on the immediate field and a specified memory location and skips if the result of the AND is nonzero.
WCOB	Count bits	Counts the number of ones in one accumulator and adds that number to the second accumulator.
WLOB	Locate lead bit	Counts the number of high-order zeroes in ACS and adds that number to ACD.
WSKBO	Wide skip on bit set to one	Tests a specified bit in AC0 and skips of the bit is 1.
WSKBZ	Wide skip on bit set to zero	Tests a specified bit in AC0 and skips of the bit is 0.
WSNB	Skip on non-zero bit	Skips the next sequential word if the bit addressed by ACS and ACD is 1.
WSZB	Skip on zero bit	Skips the next sequential word if the bit addressed by ACS and ACD is 0.
WSZBO	Skip on zero bit and set to one	Sets the bit addressed by ACS and ACD to 1 and skips the next sequential word if the bit was originally 0.

Table 7.7 MV/8000 bit rest instructions

MNEM	Name	Action
LMRF	Load modified and referenced bits	Loads the modified and referenced bits of a specified page into AC0.
ORFB	OR the referenced bits	Inclusively ORs a number of the referenced bits with a word string and stores the result in a second word string.
RRFB	Reset referenced bit	Resets the modified bits of the specified pages to zero.
SMRF	Store modified and referenced bit	Stores bits 30–31 of AC0 in the modified and referenced bits of the specified page.

Table 7.8 MV/8000 privileged bit field manipulation instructions

The MV/8000 supports the C/350 bit manipulation instructions shown in Table 7.9. Some of the bit instructions use a bit pointer to locate a bit in memory; others only affect bits within the specified accumulators. Note that the bit pointers used by these instructions have a different format than that used by MV/8000-specific bit manipulation instructions. For more information, refer to Chapter 2.

MNEM	Name	Action
BTO	Set bit to one	Sets the bit addressed by the bit pointer to 1.
BTZ	Set bit to zero	Sets the bit addressed by the bit pointer to 0.
COB	Count bits	Counts the number of ones in one accumulator and adds that number to the second accumulator.
LOB	Locate lead bit	Counts the number of high-order zeros in one accumulator and adds that number to the second accumulator.
LRB	Locate and reset lead bit	Performs a *Locate Lead Bit* instruction and sets the lead bit to 0.
SNB	Skip on non-zero bit	Skips the next sequential word if the bit addressed by the bit pointer is 1.
SZB	Skip on zero bit	Skips the next sequential word if the bit addressed by the bit pointer is 0.
SZBO	Skip on zero bit and set to one	Sets the bit addressed by the bit pointer to 1 and skips the next sequential word if the bit was originally 0.

Table 7.9 C/350 bit manipulation instructions

Byte Manipulation Instructions

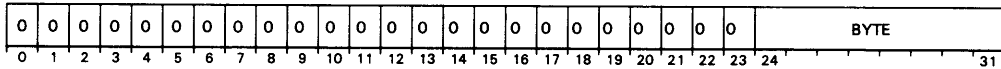
Bytes are treated as 8-bit unsigned binary integers. To reference a byte in memory, specify a 32-bit byte pointer. For information about the format of a byte pointer, refer to Chapter 3.

Table 7.10 lists the byte instructions. For more information about them, refer to the the individual instruction descriptions in the MV/8000 Instruction Dictionary, Chapter 16.

MNEM	Name	Action
LLDB	Long load byte	Loads a byte in memory into an accumulator.
LLEFB	Long load effective byte address	Calculates an effective byte address and loads it into an accumulator.
LPEFB	Long push byte address	Calculates an effective byte address and pushes it onto the wide stack.
LSTB	Long store byte	Stores a byte in an accumulator into a byte of memory.
VBP	Skip on valid byte pointer	Checks a byte pointer and skips the next word if the byte pointer is valid.
WLDB	Wide load byte	Loads a byte in memory into an accumulator.
WSTB	Wide store byte	Stores a byte in an accumulator into a byte of memory.
XLDB	Extended load byte	Loads a byte in memory into an accumulator.
XLEFB	Extended load effective byte address	Calculates an effective byte address and loads it into an accumulator.
XPEFB	Extended push byte address	Calculates an effective byte address and pushes it onto the wide stack.
XSTB	Extended store byte	Stores a byte in an accumulator into a byte of memory.

Table 7.10 MV/8000-specific byte instructions

The C/350 byte instructions supported by the MV/8000 are shown in Table 7.11. When an instruction moves a byte to an accumulator, the format of the destination accumulator will be as shown below. When an instruction moves a byte from an accumulator to memory, it leaves unchanged the other byte contained in that single word of memory.



Note that the byte pointers used by these instructions have a different format than that used by MV/8000-specific byte manipulation instructions. Refer to Chapter 2 for more information.

MNEM	Name	Action
LDB, ELDB	Load Byte	Places a byte of information into an accumulator.
STB, ESTB	Store Byte	Stores the right byte of an accumulator into a byte of memory.

Table 7.11 C/350 byte instructions

Fixed Point Single-Word Arithmetics

The following instructions manipulate 16-bit data. Most specify two accumulators, ACS and ACD. When these instructions are used, the processor performs the specified action on bits 16–31 of ACS and ACD, and produces a 16-bit result. The processor sign extends the result to 32 bits and loads it into ACD. The processor ignores the initial values of bits 0–15 of ACS and ACD.

If an instruction using an immediate field is specified, the processor will perform the specified operation on the contents of the 16-bit immediate field and the contents of bits 16–31 of the specified accumulator.

Note that the fixed point single-word adds and subtracts affect carry. Refer to individual instruction descriptions in the MV/8000 Instruction Dictionary, Chapter 16, for specifics.

MNEM	Name	Action
NADD	Narrow add	Adds the integer specified by bits 16–31 of ACS to the integer specified by bits 16–31 of ACD. Sign extends the result to 32 bits, and stores it in ACD.
NSUB	Narrow subtract	Subtracts the integer specified by bits 16–31 of ACS from the integer specified by bits 16–31 of ACD. Sign extends the result to 32 bits, and stores it in ACD.
NMUL	Narrow Multiply	Multiplies the integer specified by bits 16–31 of ACS by the integer specified by bits 16–31 of ACD. Sign extends the result to 32 bits, and stores it in ACD.
NDIV	Narrow divide	Divides the integer specified by bits 16–31 of ACS by the integer specified by bits 16–31 of ACD. Sign extends the result to 32 bits, and stores it in ACD.
NNEG	Narrow negate	Negates the integer specified by bits 16–31 of ACS. Sign extends the result to 32 bits, and stores it in ACD.
NADI	Narrow add integer	Adds the integer specified by *n* to the integer specified by bits 16–31 of the specified accumulator. Sign extends the result to 32 bits, and stores it in the specified accumulator.
NSBI	Narrow subtract integer	Subtracts the integer specified by *n* from the integer specified by bits 16–31 of the specified accumulator. Sign extends the result to 32 bits, and stores it in the specified accumulator.
NADDI	Narrow add immediate	Adds the integer specified by the specified immediate field to the integer specified by bits 16–31 of the specified accumulator. Sign extends the result to 32 bits, and stores it in the specified accumulator.

Table 7.12 Fixed point single-word arithmetic instructions

Fixed Point Double-Word Arithmetics

The 32-bit fixed point arithmetic instructions specify two accumulators, ACS and ACD. The processor loads the results of the specified operation into ACD. In most cases, *overflow* will contain the output of the ALU; refer to individual instruction descriptions in the MV/8000 Instruction Dictionary, Chapter 16, for more information.

Note that the fixed point double-word adds and subtracts affect carry. Refer to individual instruction descriptions in the Instruction Dictionary for specifics.

MNEM	Name	Action
SEX	Sign extend	Sign-extends the 16-bit integer specified by ACS to 32 bits. Stores the result in ACD.
WADC	Wide add complement	Adds the logical complement of the 32-bit integer specified by ACS to the 32-bit integer specified by ACD. Stores the result in ACD.
WADD	Wide add	Adds the 32-bit integer specified by ACS to the 32-bit integer specified by ACD. Stores the result in ACD.
WADDI	Wide add immediate	Adds the 32-bit integer specified by the immediate field to the 32-bit integer contained in the specified accumulator. Stores the result in the specified accumulator.
WADI	Wide add integer	Adds the value specified by $n + 1$ to the 32-bit integer contained in the specified accumulator. Stores the result in the specified accumulator.
WASH	Wide arithmetic shift	Arithmetically shifts the contents of ACS to the right or left a number of bits. Stores the result in ACD.
WDIV	Wide divide	Divides the 32-bit integer specified by ACS by the 32-bit integer specified by ACD. Stores the result in ACD.
WDIVS	Wide signed divide	Divides the 64-bit signed integer specified by AC0 and AC1 by the 32-bit signed integer specified by AC2. Stores the quotient in AC1 and the remainder in AC0.
WINC	Wide increment	Increments the 32-bit integer specified by ACS by one. Stores the result in ACD.
WHLV	Wide halve	Divides the 32-bit integer specified by ACS by two. Stores the result in ACD.
WMUL	Wide multiply	Multiplies the 32-bit integer specified by ACS by the 32-bit integer specified by ACD. Stores the result in ACD.
WMULS	Wide signed multiply	Multiplies the 32-bit signed integer specified by AC1 by the 32-bit signed integer specified by AC2. Adds the 32-bit signed integer contained in AC0 to the 64-bit result. Stores the result in AC0 and AC1.
WNADI	Wide add narrow immediate	Sign-extends the 16-bit value contained in the immediate field to 32 bits and adds it to the 32-bit integer contained in the specified accumulator.
WNEG	Wide negate	Loads the two's complement of the 32-bit integer specified by ACS into ACD.
WSUB	Wide subtract	Subtracts the 32-bit integer specified by ACS from the 32-bit integer specified by ACD. Stores the result in ACD.
ZEX	Zero extend	Zero-extends the 16-bit integer specified by ACS to 16 bits. Stores the result in ACD.

Table 7.13 Fixed point double-word arithmetic instructions

Fixed Point Single-Word Logicals

These C/350 instructions perform logical operations on the least significant 16 bits of the fixed point accumulators. They specify two accumulators, ACS and ACD. ACD will contain the results of the specified operation.

MNEM	Name	Action
ANC	AND with complemented source	ANDs the contents of of one accumulator and the logical complement of another accumulator.
AND	AND	ANDs the contents of two accumulators.
ANDI	AND immediate	ANDs the contents of an accumulator and and the contents of a 16-bit number contained in the instruction.
COM	Complement	Forms the logical complement of the contents of an accumulator.
DHXL	Double hex shift left	Shifts the 32-bit contents of two accumulators left 1 to 4 hex digits depending on the value of a 2-bit number contained in the instruction.
DHXR	Double hex shift right	Shifts the 32-bit contents of two accumulators right 1 to 4 hex digits depending on the value of a 2-bit number contatined in the instruction.
DLSH	Double logical shift	Shifts the 32-bit contents of two 16-bit accumulators left or fight depending on the contents of a third accumulator.
HXL	Hex shift left	Shifts the 16-bit contents of an accumulator left 1 to 4 hex digits depending on the value of a 2-bit number contained in the instruction.
HXR	Hex shift right	Shifts the 16-bit contents of an accumulator right 1 to 4 hex digits depending on the value of a 2-bit number contatined in the instruction.
IOR	Inclusive OR	Inclusively ORs the contents of two accumulators.
IORI	Inclusive OR immediate	Inclusively ORs the contents of an accumulator and the contents of a 16-bit number contained in the instruction.
LEF, ELEF	Load effective address	Places an effective address in an accumulator.
LSH	Logical shift	Shifts the contents of a 16-bit accumulators left or fight depending on the contents of a third accumulator.
XOR	Exclusive OR	Exclusively ORs the contents of two accumulators.
XORI	Exclusive OR immediate	Exclusively ORs the contents of an accumulator and the contents of a 16-bit number contained in the instruction.

Table 7.14 Fixed point single-word logical instructions

Fixed Point Double-Word Logicals

These instructions perform logical operations on the fixed point accumulators. They specify two accumulators, ACS and ACD. ACD will contain the results of the specified operation.

MNEM	Name	Action
WANC	Wide AND with complemented source	ANDs the one's complement of the contents of ACS with the contents of ACD.
WAND	Wide ADD	ANDs the contents of ACS and ACD.
WANDI	Wide AND immediate	Logically ANDs the contents of the immediate field and the specified accumulator.
WCOM	Wide complement	Forms the one's complement of the value contained in ACS and loads it into ACD.
WIOR	Wide inclusive OR	Inclusively ORs the contents of ACS and ACD.
WIORI	Wide inclusive OR immediate	Inclusively ORs the contents of the immediate field and the specified accumulator.
WLSH	Wide logical shift	Shifts the contents of ACD to the left or right.
WLSI	Wide logical shift immediate	Shifts the contents of an accumulator left as indicated by an immediate value.
WLSN	Wide load sign	Determines the sign of a number located in memory. Loads a value into AC1 which indicates the sign of the number.
WMOV	Wide move	Moves a copy of the 32-bit integer specified by ACS into ACD.
WXCH	Wide exchange accumulators	Loads the 32-bit integer specified by ACS into ACD, and the 32-bit integer specified by ACD into ACS.
WXOR	Wide exclusive OR	Exclusively ORs the contents of ACS and ACD.
WXORI	Wide exclusive OR immediate	Exclusively ORs the contents of the immediate field and the specified accumulator.

Table 7.15 Fixed point double-word logical instructions

Single-Word Compare Accumulators

These C/350 instructions compare the contents of the least significant 16 bits of two accumulators, then skip the next sequential 16-bit word if the test is true.

MNEM	Name	Action
CLM	Compare to limits	Compares a signed 16-bit integer with two other numbers and skips if the first integer is between the other two.
SGE	Skip if ACS greater than or equal to ACD	Compares two signed integers in two accumulators; skips if the first is greater than or equal to the second.
SGT	Skip if ACS greater than ACD	Compares two signed integers in accumulators; skips if the first is greater than the second.
SKP[*t*]	I/O skip	Skips if the I/O condition *t* is true.
SNB	Skip on non zero bit	References a single bit in memory via the bit pointer; skips if the bit is 1.
SZB	Skip on zero bit	References a single bit in memory via the bit pointer; skips if the bit is 0.
SZBO	Skip on zero bit, set to 1	References a single bit in memory via the bit pointer; skips if the bit is 0 and also sets the bit to 1.

Table 7.16 Single-word compare accumulator instructions

Double-Word Compare Accumulators

These instructions test and perform some action depending on the outcome of the test. They all perform a test between the contents of two accumulators, then skip the next sequential 16-bit word if the test is true. Note that the two accumulators are the same, then the processor compares the contents of the accumulator to zero, and skips or does not skip according to the outcome of the comparison.

MNEM	Name	Action
WSEQ	Wide signed skip on equal	Compares ACS to ACD and skips the next 16-bit word if ACS is equal to ACD.
WSGE	Wide signed skip on greater than or equal to	Compares ACS to ACD and skips the next 16-bit word if ACS is greater than or equal to ACD.
WSGT	Wide signed skip on greater than	Compares ACS to ACD and skips the next 16-bit word if ACS is greater than ACD.
WSLE	Wide signed skip on less than or equal to	Compares ACS to ACD and skips the next 16-bit word if ACS is less than or equal to ACD.
WSLT	Wide signed skip on less than	Compares ACS to ACD and skips the next 16-word if ACS is less than ACD.
WSNE	Wide signed skip on not equal	Compares ACS to ACD and skips the next 16-bit word if ACS is not equal to ACD.
WUSGE	Wide unsigned skip on greater than or equal to	Compares ACS to ACD and skips the next 16-bit word if ACS is greater than, or equal to, ACD.
WUSGT	Wide unsigned skip on greater than	Compares ACS to ACD and skips the next 16-bit word if ACS is greater than ACD.

Table 7.17 Double-word compare accumulator instructions

Other Instructions

The instructions in this group perform miscellaneous operations. For specific information about any of them, refer to individual instruction descriptions in the MV/8000 Instruction Dictionary, Chapter 16.

MNEM	Name	Action
BKPT	Breakpoint	Pushes a wide return block onto the wide stack, then performs a jump indirect through locations $10\text{-}11_8$.
LCPID	Load CPU identification	Loads a binary representation of the machine's model number into AC0.
OPESC	Opcode escape	Interprets the second word of this instruction as a new instruction.
PBX	Pop block and execute	Saves an instruction, pops a return block, then executes the instruction just stored.
WBLM	Wide load block	Moves words sequentially from one location to another.
WCLM	Wide compare to limits	Compares a signed 32-bit integer to two limit values and skips if the integer is between the limit values.

Table 7.18 Other MV/8000-specific fixed point instructions

The MV/8000 also supports the C/350 instructions shown in Table 7.19.

MNEM	Name	Action
SYC, SVC	System call	Pushes a return block onto the stack; places the address of the *System Call* handler in the program counter.
XOP, XOP0	Extended operation	Pushes a narrow return block onto the stack, indexes into the XOP table, and transfers control to another procedure.
XCT	Execute	Executes the contents of an accumulator as an instruction.

Table 7.19

Programming Notes

Fixed point overflow is enabled on the MV/8000. For information about this topic, refer to Chapter 11.

When the processor increments the PC, only the least-significant 28 bits take part in this operation. This means that all normal instruction references will remain in the current segment.

When an instruction performs a skip, it skips the *next sequential 16-bit word.* Make sure that a skip does not occur in the middle of a 32-bit or longer instruction.

C/350 ALC Manipulation

Each of the eight C/350 Arithmetic/Logic Class (ALC) instructions performs a specific function upon the contents of one or two accumulators and carry. The eight functions are *Add, AND, Subtract, Negate, Add Complement, Move, Increment*, and *Complement.* The instructions are identified by the mnemonics of the eight functions, which are **ADD, AND, SUB, NEG, ADC, MOV, INC**, and **COM**.

In addition to the specific functions performed by an individual instruction, there is a group of general functions all ALC instructions can perform. These general functions include shift operations, which rotate the data left or right, or swap the bytes. Also included are various tests that can be performed on the data. With each test the instructions can check the data for some condition and skip or not skip the next sequential word depending on the outcome of the test. Finally, the instructions can load or not load the results of the specific and general functions into the destination accumulator and the carry bit. The diagram below shows the format of the ALC instructions.

ALC Instructions

The C/350 ALC instructions are listed in Table 7.20.

MNEM	Name	Action
ADC	Add complement	Adds an unsigned integer to the logical complement of another unsigned number.
ADD	Add	Adds contents of one accumulator to the contents of another.
AND	AND	Forms the logical AND of the contents of two accumulators.
COM	Complement	Forms the logical complement of the contents of an accumulator.
INC	Increment	Increments the contents of an accumulator.
MOV	Move	Moves the contents of an accumulator through the ALU.
NEG	Negate	Forms the two's complement of the contents of an accumulator.
SUB	Subtract	Subtracts contents of one accumulator from the contents of another.

Table 7.20 C/350 ALC instructions

C/350 ALC Instruction Execution

The C/350 ALC instructions use the Arithmetic Logic Unit (ALU) to process data. The logical organization of the ALU is shown in Figure 7.1.

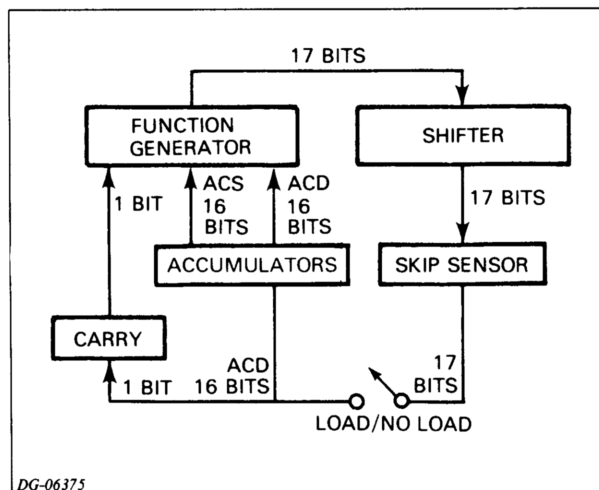


Figure 7.1

When an ALC instruction begins execution, it loads the contents of carry and the contents of the accumulator(s) to be processed into the ALU.

There are five distinct stages of ALU operation.

Carry

The ALU begins its manipulation of the data by determining a new value for carry. This new value is based upon three things: the old value of carry, bits 10–11 of the ALC instruction, and the ALC instruction being executed.

The ALU first determines the effect of the instruction bits 10–11 on the old value of carry. Table 7.21 shows each of the mnemonics that can be appended to the instruction mnemonic, the value of bits 10–11 for each choice, and the action each one takes.

Symbol	Value	Operation
[c] omitted	00	Leave carry unchanged
[c]=Z	01	Initialize carry to 0
[c]=O	10	Initialize carry to 1
[c]=C	11	Complement carry

Table 7.21

Function

The ALU next evaluates the effect of the specific function (bits 5–7) upon the data. For the instructions *AND, Move, Negate* and *Complement*, the ALU performs the function on the data word(s) and saves the result. The value of carry is as it was calculated above.

For the instructions *Add, Add Complement, Subtract* and *Increment*, the result of the function's action upon the data word(s) may be larger than $2^{16} - 1$. In this situation, the ALU saves the low-order 16 bits of the function result, but it complements the value of carry calculated above.

> **NOTE:** *At this stage of operation, the ALU does not load either the saved value of the function result into the destination accumulator, or the saved value of carry into carry.*

Shift Operations

Next the ALU performs any specified shift operation on the 17 bits output from the function generator (16 bits of data plus the calculated value of carry). Depending on which shift operation is specified in the instruction, the function generator output can be rotated left or right one bit, or have its bytes swapped. Table 7.22 shows the different shift operations that can be performed, the value of bits 8–9 for each choice, and the action each choice takes. Figure 7.2 shows how each shift operation works.

Symbol	Value	Operation
[sh] omitted	00	Do not shift the result of the ALC operation
[sh]=L	01	Rotate left the 17-bit combination of Carry bit and ALC operation result
[sh]=R	10	Rotate right the 17-bit combination of Carry bit and ALC operation result
[sh]=S	11	Swap the two 8-bit halves of the ALC operation result without affecting Carry bit

Table 7.22

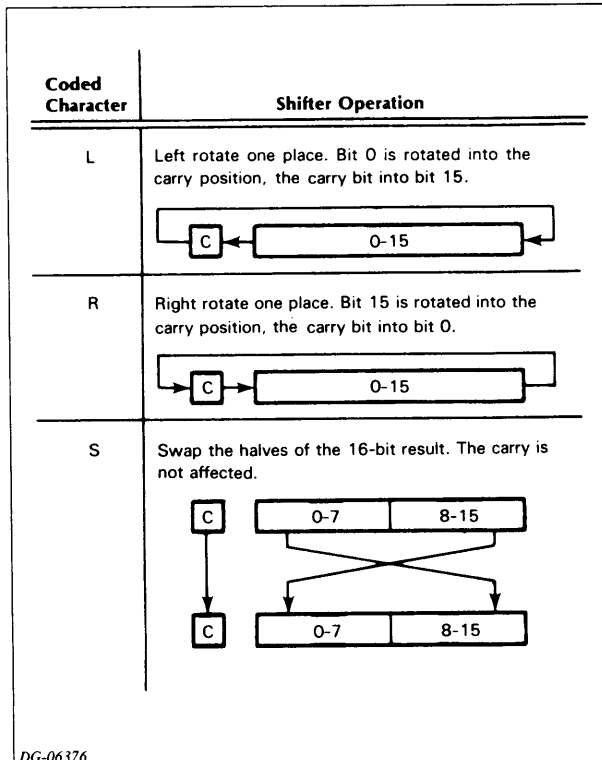


Figure 7.2

Skip Tests

The ALU can test the result of the shift operation for one of a variety of conditions, and skip or not skip the next instruction depending upon the result of the test. Table 7.23 shows the tests that can be performed, the value of bits 13–15 for each choice, and the action each choice takes.

Symbol	Value	Operation
[skip] omitted	000	No skip
[skip]=**SKP**	001	Skip unconditionally
[skip]=**SZC**	010	Skip if carry is zero
[skip]=**SNC**	011	Skip if carry is nonzero
[skip]=**SZR**	100	Skip if ALC result is zero
[skip]=**SNR**	101	Skip if ALC result is nonzero
[skip]=**SEZ**	110	Skip if either ALC result or carry is zero
[skip]=**SBN**	111	Skip if both ALC result and carry are nonzero

Table 7.23

Load/No-Load

If the no-load bit (bit 12) is 0, the ALU loads the result of the shift operation into the destination accumulator, and loads the new value of carry into carry. If the no-load bit is 1, then the ALU does not load the result of the shift operation into the destination accumulator, and does not load the new value of carry into carry, but all other operations, such as skip tests, take place.

This no-load option is particularly convenient to use when you want to test for some condition without destroying the contents of the destination accumulator. Table 7.24 shows how to code the load/no-load operation.

Symbol	Value	Operation
# omitted	0	Load the result of the shift operation into ACD.
#	1	Do not load the ALC operation result into ACD; restore carry to value it had before shifting.

Table 7.24

NOTE: *These instructions must not have both the* No-Load *and the* Never-Skip *options specified at the same time. These bit combinations are used by other instructions in the instruction set.*

Chapter 8
Floating Point Instructions

Chapter 6 described the format of floating point numbers. This chapter explains how to use these numbers in calculations. Appendix G summarizes the floating point operations and how they are performed.

Floating point numbers referenced by a word address can begin on any word boundary. Those referenced by a byte address (as used in commercial or character instructions, see Chapters 9 and 10) can begin on any byte boundary.

True and Impure Zero

Floating point zero is represented by a number with all bits zero, known as *true zero.* If a number has a zero mantissa but not a zero sign or exponent, it is called *impure zero.* When representing zero as a floating point number, use true zero; impure zero produces undefined results in calculations.

Normalized Format

A nonzero mantissa represents a fraction from 1/16 to $1\text{-}2^{-56}$. A floating point number represented in this way is said to be *normalized.* Note that impure zero is not in normalized form. Most floating point instructions require normalized operands if they are to produce correct results. Floating point numbers that are not normalized or are not true zero produce undefined results except as noted.

Magnitude

The magnitude of a floating point number is defined as follows:

$Mantissa \times 16^y$

where y is the true value of the exponent.

Guard Digits

In order to increase the accuracy of floating point operations, a *guard digit* (or *digits)* is appended to the least-significant bit of each mantissa. A guard digit is one hex digit (four bits) that contains zeroes. The value of bit 8 of the FPSR determines how many guard digits to append. If bit 8 is 0, then the processor appends one guard digit; if bit 8 is 1, it appends two guard digits.

In addition to choosing the number of guard digits to append, bit 8 of the FPSR also specifies a rounding algorithm to be performed on the result of the specified floating point operation. These algorithms use the guard digits, and are discussed later in this chapter.

Floating Point Operation

When a floating point operation between two floating point operands is specified, the processor first appends the appropriate number of guard digits (as determined by bit 8 of the FPSR). For single precision operands, the processor appends the guard digit or digits to bit 23 of the operand mantissas; for double, to bit 56. This means that the mantissa of an operand can be from 28 bits to 64 bits long, depending on the value of FPSR bit 8 and the specified precision.

After appending the guard digits, the processor performs the specified operation (see Appendix G). The result of the operation is called the *intermediate result.* The processor normalizes this value if necessary. It does this by shifting the intermediate result left one hex digit (4 bits) at a time until the high-order four bits (bits 0–3 of the mantissa) represent a nonzero quantity. Zeroes are filled in on the right. For every hex digit shifted, the processor decrements the exponent of the intermediate result by one.

Note that normalization can cause an exponent underflow or correct an exponent overflow. Because further processing can correct this underflow or overflow, however, the processor does not set the appropriate flags at this time. The processor maintains the normalized result internally until the final result is produced.

Maintaining Accuracy

After normalization, the mantissa of the intermediate value is more than 24 bits (single precision) or 56 bits (double precision) wide. Because the mantissa must fit into the specified location or register, the processor must perform one of two rounding algorithms to correctly size the mantissa. These algorithms are *truncation* and *unbiased rounding.* In truncation, the processor removes the least significant digit of the intermediate mantissa. In unbiased rounding, the processor uses the 2 least significant digits of the intermediate mantissa to round the remaining bits up or down to the correct size.

Truncation

Truncation occurs when bit 8 of the FPSR is 0. Because one guard digit was appended to each of the operands, the processor manipulates 28-bit single precision mantissas and 60-bit double precision mantissas. After normalization, the processor truncates the mantissa of the intermediate result to 24 or 56 bits. This is the final result.

Unbiased Rounding

Unbiased rounding occurs when bit 8 of the FPSR is 1. Because two guard digits were appended to each mantissa, the processor manipulates 32-bit single precision mantissas and 64-bit double precision mantissas. After performing the specified operation and any normalization, the processor truncates the intermediate result to 24 or 56 bits, but saves the 2 least significant digits (the guard digits) for rounding.

The processor rounds the mantissa in one of three ways. To determine the rounding method, the processor treats the two guard digits as an 8-bit integer. If the integer formed by the guard digits is within the range 0 to $7F_{16}$ inclusive, then the normalized intermediate result becomes the final result without change.

If the integer formed from the guard digits is exactly 80_{16}, then the processor adds the least significant bit of the intermediate result to the intermediate result. The sum becomes the final result. This forces even mantissas to be rounded down to the nearest integer and odd mantissas to be rounded up to the nearest integer.

If the integer formed from the guard digits is within the range $81–FF_{16}$ inclusive, then the processor adds 1 to the intermediate result to form the final result. Note that this can cause a mantissa overflow. If this occurs, the processor shifts the intermediate mantissa right one hex digit, places 0001 into the mantissa's most significant hex digit, and adds 1 to the intermediate exponent. The processor truncates the rightmost hex digit so that the intermediate mantissa is 24 or 56 bits long. This 24- of 56-bit value is the final result.

Finishing Up

After performing the appropriate rounding algorithm, the processor loads the final result into the specified destination, then checks for any exponent underflow or overflow that may have occurred during the calculations. If no underflow or overflow condition exists (the exponent is within the range of -64 to +63 inclusive), the instruction ends.

If an underflow or overflow does exist, the processor sets the appropriate flag in the FPSR. The exponent field in the destination location contains only the seven least significant digits of the true value of the exponent; this means that the stored value of the exponent will be 128 larger or smaller than the true value. Check individual instruction descriptions in the Instruction Dictionary for details on this topic.

NOTE: *Certain floating point instructions do not use unbiased rounding in their manipulations, even though bit 8 of the FPSR may specify a 1. Refer to the individual instruction definitions in the MV/8000 Instruction Dictionary, Chapter 16, for specifics.*

Floating Point Instructions

The MV/8000-specific floating point instructions are shown in Table 8.1, and the C/350 floating point instructions are shown in Table 8.2. Note that several instructions have two forms, one ending in *S* and one ending in *D*. The first form uses single precision floating point format; the second uses double precision floating point format. The function of the two forms is otherwise identical.

MNEM	Name	Action
LFAMD, LFAMS	Long add	Adds a floating point number in memory to the floating point number in an FPAC.
LFMD, LFDMS	Long divide	Divides the floating point number in an FPAC by a floating point number in memory.
LFLDD, LFLDS	Long load floating point	Loads a floating point number from memory into an FPAC.
LFLST	Long load floating point status	Loads the contents of memory into the FPSR.
LFMMD, LFMMS	Long multiply double	Multiplies a floating point number in memory by the floating point number in an FPAC.
LFSMD, LFSMS	Long subtract double	Subtracts the floating point number in memory from the floating point number in an FPAC.
LFSST	Long store floating point status	Stores the contents of the FPSR into memory.
LFSTD, LFSTS	Long store floating point double	Stores the contents of an FPAC into memory.
WFFAD	Wide fix to AC	Converts the integer portion of an FPAC to a double precision signed, two's complement integer and places the result in an accumulator.
WFLAD	Wide float from AC	Converts the double precision signed, two's complement integer in an accumulator to a double precision floating point number and places the result in an FPAC.
WFPOP	Wide pop floating point state	Pops a 20-word floating point block off the wide stack and alters the state of the floating point unit.
WFPSH	Wide push floating point state	Pushes a 20-word floating point block onto the wide stack.
XFAMD, XFAMS	Long add	Adds a floating point number in memory to the floating point number in an FPAC.
XFDMD, XFDMS	Long divide	Divides the floating point number in an FPAC by a floating point number in memory.
XFLDD, XFLDS	Long load floating point	Loads a floating-point number from memory into an FPAC.
XFMMD, XFMMS	Long multiply double	Multiplies a floating point number in memory by the floating point number in an FPAC.
XFSMD, XFSMS	Long subtract double	Subtracts the floating-point number in memory from the floating-point number in an FPAC.
XFSTD, XFSTS	Long store floating point double	Stores the contents of an FPAC into memory.

Table 8.1 MV/8000-specific floating point instructions

MNEM	Name	Action
FAB	Absolute value	Sets the sign bit of an FPAC to 0.
FAMS, FAMD	Add (memory to FPAC)	Adds the floating point number in memory to the floating point number in an FPAC.
FAS, FAD	Add (FPAC to FPAC)	Adds the floating point number in one FPAC to the floating point number in another FPAC.
FCLE	Clear errors	Sets bits 0–4 of the FPSR to 0.
FCMP	Compare floating point	Compares two floating point numbers and sets the Z and N flags accordingly.
FDMS, FDMD	Divide (FPAC by memory)	Divides the floating point number in an FPAC by a floating point number in memory.
FDS, FDD	Divide (FPAC by FPAC)	Divides the floating point number in one FPAC by the floating point number in another FPAC.
FEXP FFAS	Load exponent Fix to AC	Places bits 1–7 of AC0 in bits 1–7 of the specified FPAC. Converts the integer portion of a floating point number contained in an FPAC to a single precision, signed, two's complement integer and places the result in an accumulator.
FFMD	Fix to memory	Converts the integer portion of a floating point number to double-precision integer format contained in an FPAC and stores the result in two memory locations.
FHLV	Halve	Divides the floating point number in FPAC by 2.
FINT	Integerize	Sets the fractional portion of the floating point number in the specified FPAC to zero and normalizes the result.
FLAS	Float from AC	Converts a single precision, signed, two's complement integer in an accumulator to a single precision floating point number and places the result in an FPAC.
FLDS, FLDD	Load floating point	Loads a floating point number from memory to a specified FPAC.
FLMD	Float from memory	Converts a double precision, signed, two's complement integer in two memory locations to a double precision flaoting point number and stores the result in an FPAC.
FLST	Load floating point status	Loads the 32-bit contents of two specified memory locations into the FPSR.
FMMS, FMMD	Multiply (memory by FPAC)	Multiplies the floating point number in memory by the floating point number in an FPAC.
FMOV	Move floating point	Moves the contents of one FPAC to another FPAC.
FMS, FMD	Multiply (FPAC by FPAC)	Multiplies the floating point number in one FPAC by the floating point number in another FPAC.
FNEG	Negate	Inverts the sign bit of the FPAC.
FNOM	Normalize	Normalizes the floating point number in FPAC.
FNS	No skip	Executes the next sequential word.
FPOP	Pop floating point state	Pops an 18-word floating point block off the narrow stack and alters the state of the floating point unit.

Table 8.2 C/350 floating point instructions

MNEM	Name	Action
FPSH	Push floating point state	Pushes an 18-word floating point block onto the narrow stack.
FRH	Read high word	Places the high-order 16 bits of an FPAC in bits 16-31 of AC0.
FSA	Skip always	Skips the next sequential instruction.
FSCAL	Scale	Shifts the mantissa of the floating point number in FPAC either right or left, depending upon the contents of bits 17-23 of AC0.
FSEQ	Skip on zero	Skips the next sequential word if the *Z* flag of the FPSR is 1.
FSGE	Skip on greater than or equal to zero	Skips the next sequential word if the *N* flag of the FPSR is 0.
FSGT	Skip on greater than zero	Skips the next sequential word if both the *Z* and *N* flags of the FPSR are 0.
FSLE	Skip on less than or equal to zero	Skips the next sequential word if either the *Z* flag or the *N* flag of the FPSR is 1.
FSLT	Skip on less than zero	Skips the next sequential word if the *N* flag of the FPSR is 1.
FSMS, FSMD	Subtract (memory from FPAC)	Subtracts the floating point number in memory from the floating point number in an FPAC.
FSND	Skip on no zero divide	Skips the next sequential word if the divide by zero *(DVZ)* flag of the FPSR is 0.
FSNE	Skip on non-zero	Skips the next sequential word if the *Z* flag of the FPSR is 0.
FSNER	Skip on no error	Skips the next sequential word if bits 1-4 of the FPSR are all 0.
FSNM	Skip on no mantissa overflow	Skips the next sequential word if the mantissa overflow *(MOF)* flag of the FPSR is 0.
FSNO	Skip on no overflow	Skips the next sequential word if the overflow *(OVF)* flag of the FPSR is 0.
FSNOD	Skip on no overflow and no zero divide	Skips the next sequential word if both the overflow *(OVF)* flag and the divide by zero *(DVZ)* flag of the FPSR are 0.
FSNU	Skip on no underflow	Skips the next sequential word if the underflow *(UNF)* flag of the FPSR is 0.
FSNUD	Skip on no underflow and no zero divide	Skips the next sequential word if both the underflow *(UNF)* flag and the divide by zero *(DVZ)* flag of the FPSR are 0.
FSNUO	Skip on no underflow and no overflow	Skips the next sequential word if both the underflow *(UNF)* flag and the overflow *(OVF)* flag of the FPSR are 0.
FSS, FSD	Subtract (FPAC from FPAC)	Subtracts the floating point number in one FPAC from the floating point number in another FPAC.
FSST	Store floating point status	Moves the contents of the FPSR to two memory locations.
FSTS, FSTD	Store floating point	Stores the contents of a specified FPAC into memory.
FTD	Trap disable	Sets the Trap Enable flag of the FPSR to 0.
FTE	Trap enable	Sets the Trap Enable flag of the FPSR to 1.

C/350 floating point instructions (continued)

Floating Point Faults

Floating point faults can occur upon completion of any floating point instruction. If any of bits 1–4 of the FPSR have the value 1 at this time, then a floating point fault condition exists. Bit 5 of the FPSR (the Trap Enable bit) determines how the processor handles this condition.

If the Trap Enable bit has the value 0, then the processor continues execution with the next sequential instruction in the currently executing program. Program flow remains unchanged.

If the Trap Enable bit has the value 1, then a fault will occur. The processor performs an indirect jump through location 45_8 to the fault handler and examines the first word of the handler. If this word's bit 0 is 1 and bits 12–15 are 1001, the wide stack fault handler

services the fault.

For information about how the processor services floating point faults, refer to the section *Floating Point Faults* in Chapter 11.

Chapter 9
Commercial Instructions

Chapter 6 described the kinds of commercial formats used on the MV/8000. This chapter describes the instructions that manipulate commercial data. Some instructions listed here are useful for handling a variety of other data types as well as commercial types; others are for use with commercial formats only. Also included in this chapter is a discussion of commercial faults.

Decimal Instructions

The MV/8000 decimal numeric instructions:

- Load decimal numbers from memory into a 64-bit floating point accumulator,
- Store decimal numbers from an FPAC into memory,
- Load the sign of a number,
- Convert decimal integers to byte strings, then process the strings.

Table 9.1 lists these instructions.

MNEM	Name	Action
WEDIT	Wide Edit	Converts a decimal integer to a string of bytes controlled by an edit subprogram; or manipulates a string of bytes.
WLDI	Wide Load Integer	Converts a decimal integer to normalized floating point format and places it in the specified FPAC.
WLDIX	Extended Wide Load Integer	Distributes a decimal integer into four FPACs.
WLSN	Wide Load Sign	Evaluates a number in memory and returns a code that indicates the sign of the number.
WSTI	Wide Store Integer	Converts the contents of an FPAC to the specified format and stores the result into memory.
WSTIX	Extended Wide Store Integer	Converts the contents of 4 FPACs to integer format and uses the 8 low-order digits of each FPAC to form a 32-digit integer.

Table 9.1 Decimal numeric instructions

In addition, the MV/8000 supports the C/350 decimal numeric instructions summarized in Table 9.2.

MNEM	Name	Action
DAD	Decimal Add	Adds together the decimal digits found in bits 12–15 of two accumulators.
DSB	Decimal Subtract	Subtracts the decimal digit in bits 12–15 of one accumulator from the decimal digit in bits 12–15 of another acumulator.
EDIT	Edit	Converts a decimal integer to a string of bytes controlled by an edit subprogram; or manipulates a string of bytes.
LDI	Load Integer	Converts a decimal integer to normalized floating point form and places it in a specified floating point accumulator.
LDIX	Extended Load Integer	Distributes a decimal integer into four floating point accumulators.
LSN	Load	Evaluates a number in memory and returns a code indicating the sign of the number.
STI	Store Integer	Converts the contents of a floating point accumulator to a specified format and stores it in memory.
STIX	Extended Store Integer	Converts the contents of four floating point accumulators to integer form and uses the eight low-order digits of each to form a 32-digit integer.

Table 9.2 C/350 decimal arithmetic instructions

Commercial Faults

When the processor executes decimal instructions, it checks the data for invalid numbers and invalid data types. If the processor finds either of these, a *commercial fault (CF)* occurs.

When a fault occurs, the processor pushes a return block onto the stack (see Chapter 11). The size of the return block depends on which fault occurred and the instruction that caused it. The PC in the return block points to the instruction that caused the fault. The processor sets *OVK* and *OVR* to 0 and places the appropriate fault code in AC1. AC0 contains the value of the PC for the instruction that caused the fault. An indirect jump through location 46_8 transfers control to the commercial fault handler. Location 46_8 contains a 16-bit pointer to the fault handler.

Table 9.3 describes the commercial faults that can occur. The first column lists the code that will appear in AC1 when a C/350 fault occurs. The second column lists the code that will appear in AC1 when an MV/8000 fault occurs. The third column lists the instructions that can cause the fault. The last column describes the conditions that can cause the fault.

CF Number	C/350 Code	MV/8000 Code	Mnem	Meaning
0	0	100000	**WEDIT, EDIT**	An invalid digit or alphabetic character encountered during execution of one of the following subopcodes: **DMVA, DMVF, DMVN, DMVO, DMVS.**
1	1	100001	**WEDIT, EDIT, LDIX, STIX, WLDIX, WSTIX**	Invalid data type (6 or 7); AC3 contains the data size and precision.
2	2	100002	**WEDIT, EDIT**	**DMVA** or **DMVC** opcode with source data type 5; AC2 contains the data size and precision.
3	3	100003	**WEDIT, EDIT**	An invalid opcode; AC2 contains the data size and precision.
4	4	100004	**WLDI, WSTI, LDI, STI**	Number too large to convert to specified data type.
6	6	100006	**LSN, LDI, LDIX, WLSN, WLDI, WLIDX, WEDIT, EDIT**	Sign code is invalid for this data type.
7	7	100007	**LSN, LDI, LDIX, WLSN, WLDI, WLDIX**	Invalid digit.

Table 9.3 Commercial faults and corresponding fault conditions

Return Blocks

As mentioned above, the processor pushes a return block onto the appropriate stack when a commercial fault occurs. Depending on the type of fault and the instruction that caused it, the return block can vary in size and contents.

CF0, CF2 and CF3

When *CF0, CF2,* or *CF3* occur during **EDIT**, the narrow commercial fault handler services the fault. The processor pushes a narrow return block onto the narrow stack. The return block has the format shown in Table 9.4.

Word	Contents	Interpretation
1-4	---	Reserved.
5	AC0	Updated P (PC of **EDIT** subopcode.)
6	AC1	Data type indicator
7	AC2	Undefined.
8	AC3	Undefined.
9	Carry and PC	Carry and PC of instruction that caused the fault.

Table 9.4 Narrow Return block format

When *CF0, CF2,* or *CF3* occur during **WEDIT**, the wide commercial fault handler services the fault. The processor pushes a wide return block will be pushed onto the wide stack. The return block has the format shown in Table 9.5.

Word	Contents	Interpretation
1-2	PSR	Processor Status Register
3-4	AC0	Updated P (PC of **WEDIT** subopcode.)
5-6	AC1	Data type indicator
7-8	AC2	Undefined.
9-10	AC3	Undefined.
11-12	Carry and PC	Carry and PC of instruction that caused the fault.

Table 9.5 Wide return block format

CF1

When *CF1* occurs during execution of **WLDIX** or **WSTIX**, the wide handler services the fault. The return block has the format shown in Table 9.6.

Word	Contents	Interpretation
1-2	PSR	Processor status register.
3-4	AC0	Original value of AC0.
5-6	AC1	Data type indicator.
7-8	AC2	Original SI/DI.
9-10	AC3	Undefined.
11-12	Carry and PC	Carry and PC of instruction that caused the fault.

Table 9.6 CF1 return block

When *CF1* occurs during execution of **LDIX** or **STIX**, the narrow handler services the fault. The return block has the same format shown in Table 9.6 except that it is made up of single words and does not include the PSR.

When *CF1* occurs during execution of **WEDIT**, the wide fault handler services the fault. The return block is pushed onto the wide stack and has the format shown in Table 9.7.

Word	Contents	Interpretation
1-2	PSR	PSR at time of fault.
3-4	AC0	Original P.
5-6	AC1	Data type indicator.
7-8	AC2	Original SI/DI.
9-10	AC3	Undefined.
11-12	Carry and PC	Carry and PC of instruction that caused the fault.

Table 9.7 CF1 wide return block

When *CF1* occurs during execution of **EDIT**, the narrow handler services the fault. The return block is pushed onto the narrow stack and has the format shown in Table 9.8.

Word	Contents	Interpretation
1	ACO	Original P.
2	AC1	Undefined.
3	AC2	Undefined.
4	AC3	Undefined.
5	Carry and PC	Carry and PC of instruction that caused the fault.

Table 9.8 CF1 narrow return block

CF4 and CF7

When *CF4* or *CF7* occurs during an MV/8000 non-*Edit* instruction, the return block has the format shown in Table 9.9.

Word	Contents	Interpretation
1-2	PSR	Processor status register.
3-4	ACO	Original ACO.
5-6	AC1	Data type indicator.
7-8	AC2	Original SI/DI.
9-10	AC3	Undefined.
11-12	Carry and PC	Carry and PC of instruction that caused the fault.

Table 9.9 CF4 and CF7 return block

If *CF4* or *CF7* occurs during a C/350 non-*Edit* instruction, the return block has the same format as shown above except that it is made up of single words and does not include the PSR.

For *CF4* or *CF7* occurring during **WEDIT**, the format is as shown in Table 9.10.

Word	Contents	Interpretation
1-2	PSR	PSR at time of fault.
3-4	ACO	Original ACO.
5-6	AC1	Data type indicator.
7-8	AC2	Original SI/DI.
9-10	AC3	Undefined.
11-12	Carry and PC	Carry and PC of instruction that caused the fault.

Table 9.10 CF4 and CF7 wide return block

For *CF4* or *CF7* occurring during **EDIT**, the format is as shown in Table 9.10, except the return block is made up of single words and does not include the PSR.

Note that when *CF4* or *CF7* occurs, AC3 will contain a pointer to the next byte that would have been processed by the instruction had the fault not occurred.

CF6

When *CF6* occurs during **WEDIT**, the wide handler services the fault. The return block will be pushed onto the wide stack, and has the format shown in Table 9.11.

Word	Contents	Interpretation
1-2	PSR	PSR at time of fault.
3-4	AC0	Original P.
5-6	AC1	Undefined.
7-8	AC2	Undefined.
9-10	AC3	Undefined.
11-12	Carry and PC	Carry and PC of **WEDIT** instruction.

Table 9.11 CF6 return block

When *CF6* occurs during **EDIT**, the return block looks exactly as the one immediately above, except that it is made up of single words and does not include the PSR.

When *CF6* occurs during a non-*Edit* instruction, the return block has the format shown in Table 9.12.

Word	Contents	Interpretation
1-2	PSR	Processor status register.
3-4	AC0	Original AC0.
5-6	AC1	Data type indicator.
7-8	AC2	Original SI/DI.
9-10	AC3	Undefined.
11-12	Carry and PC	Carry and PC of instruction that caused the fault.

Table 9.12 CF6 wide return block

When *CF6* occurs during a C/350 non-*Edit* instruction, the return block has the same format as shown above except that is is made up of single words and does not contain the PSR.

NOTE: *The* **WEDIT** *instruction may use the stack for temporary storage. When a fault occurs, the processor pushes the return block after any words that the* **WEDIT** *instruction has pushed. Make sure that the comercial fault handler is able to handle this.*

Only MV/8000-specific instructions (as opposed to C/350 instructions) can cause MV/8000 commercial faults.

Chapter 10
Character String Instructions

Character Manipulation

The MV/8000 supports four instructions that manipulate a string of characters. A character string consists of a number of bytes, each containing one character. These strings can be of any data type.

Character Instructions

The character instructions listed in Table 10.1 can specify forward or backward moves and scans. When the move or scan is backwards, the processor checks for ring crossing violations. If a ring crossing would occur, the instruction does not execute and a protection fault occurs. AC1 will contain a 4.

MNEM	Name	Action
WCMP	Wide Character Compare	Compares one string of characters in memory to another string.
WCMT	Wide Character Move Until True	Moves a string of bytes from one area of memory to another until a table- specified delimiter character is encountered or the source string is exhausted.
WCMV	Wide Character Move	Moves a string of bytes from one area of memory to another under control of the values in the four accumulators.
WCTR	Wide Character Translate	Translates a string of bytes from one data representation to another and either moves it to another area of memory or compares it to a second string.

Table 10.1 Wide character string instructions

C/350 Character Instructions

The MV/8000 also supports the four C/350 character instructions shown in Table 10.2.

MNEM	Name	Action
CMP	Character compare	Compares one string of characters in memory to another string.
CMT	Character move until true	Moves a string of bytes from one area of memory to another until a table- specified delimiter character is encountered or the source string is exhausted.
CMV	Character move	Moves a string of bytes from one area of memory to another under control of the values in the four accumulators.
CTR	Character translate	Translates a string of bytes from one data representation to another and either moves it to another area of memory or compares it to a second string of bytes.

Table 10.2 C/350 character string instructions

Chapter 11
Stacks and Fault Handling

This chapter discusses stacks, the instructions that are used to manipulate stacks, and the use of stacks in fault handling.

Introduction

A stack is a series of consecutive locations in memory. In its simplest form, data is *pushed* onto the stack in sequential order and *popped* from the stack in reverse order. This means that the stack is used as storage for temporary data. More often, the stack is used to store a *return block*, which contains information that the processor uses when entering and exiting subroutines.

Three types of stacks are used on the MV/8000: the wide stack, the narrow stack and the vector stack. The *wide stack* is a series of 32-bit locations managed by four 32-bit registers. This stack is used in programs that use MV/8000-specific instructions. The *narrow stack* is also supported on the MV/8000. This stack is a series of 16-bit locations and is managed by four reserved storage locations. This stack is used in programs that contain C/350 stack instructions. The *vector stack* is used during interrupt service, and is discussed in Chapter 14.

The MV/8000 uses several stacks to maintain the system, but other stacks can be used in programs at the same time. Note that the same program may use both a wide stack and a narrow stack.

Stack instructions store the contents of accumulators on the stack, change the stack registers that control the stack, define new stacks, and other tasks.

The Wide Stack

Stack Registers

Four 32-bit registers manage the wide stack. They are:

- **WSP** – the wide stack pointer
- **WFP** – the wide frame pointer
- **WSB** – the wide stack base
- **WSL** – the wide stack limit

> NOTE: *The symbols for the four registers shown above are used throughout this book to distinguish them from the narrow stack terms.*

Wide Stack Pointer
WSP contains the address of the double-word at the top of the wide stack. When data is pushed onto the wide stack, the processor first increments **WSP** by 2 and then stores data to be pushed in the new location addressed by **WSP**. When data is popped, the processor takes the double-word addressed by **WSP** and places it in the specified register, then decrements **WSP** by 2. Note that all 32 bits of **WSP** take part in the increment or decrement.

Wide Frame Pointer
WFP contains the address of the first available double-word minus two in the current frame.

Wide Stack Limit
WSL contains an address used to determine stack overflow. After any push operation, the processor compares **WSP** to **WSL**. If **WSP** is greater than **WSL**, a stack fault occurs.

Wide Stack Base
WSB contains an address used to determine stack underflow. After any pop operation, the processor compares **WSP** to **WSB**. If **WSP** is less than **WSB**, a stack fault occurs.

The processor initializes **WSL** and **WSB** when a cross-ring call is made. There are also instructions that allow **WSP** and **WFP** to be changed directly.

Wide Stack Operation

To be meaningful, the address contained in the stack limit must be 24 to 32 words less than the address of the last word in the stack. This is because the processor can only detect stack overflow at the end of a push operation (with two exceptions that are discussed below). After pushing a 12- to 20-word return block onto the stack, the processor checks for overflow. If overflow has occurred, the processor signals a stack overflow, and the stack fault handler pushes another 12-word return block onto the stack. Depending upon the size of the first return block, the potential overflow can be 24 to 32 words long.

Usually the processor detects overflow only at the end of a push operation. The exceptions are the use of the **WSAVR, WSAVS, WSSVR,** or **WSSVS** instructions or when the processor copies parameters during an inward cross-ring call. In either of these cases, the processor checks for overflow *before* pushing data onto the stack.

A segment's stack registers are stored in one of two places. For the current segment, the contents are stored in the hardware registers. For a segment that is not the current one, the contents are stored in locations $20–27_8$ of that segment.

To reference the stack registers of the current segment, the appropriate instructions must be used. To reference the stack registers of any other segment, a memory reference instruction to the appropriate location in the range $20–27_8$ of the desired segment should

be used. Note that a program must not reference locations $20\text{–}27_8$ of the segment containing it.

Make sure that the initial values of the stack registers are even. This means that the registers should address locations that are aligned on double-word boundaries. The stack and stack operations can be used if the registers contain odd addresses, but the performance for some stack instructions will be slower.

Figure 11.1 shows a typical wide stack in use and the locations referenced by the four stack registers.

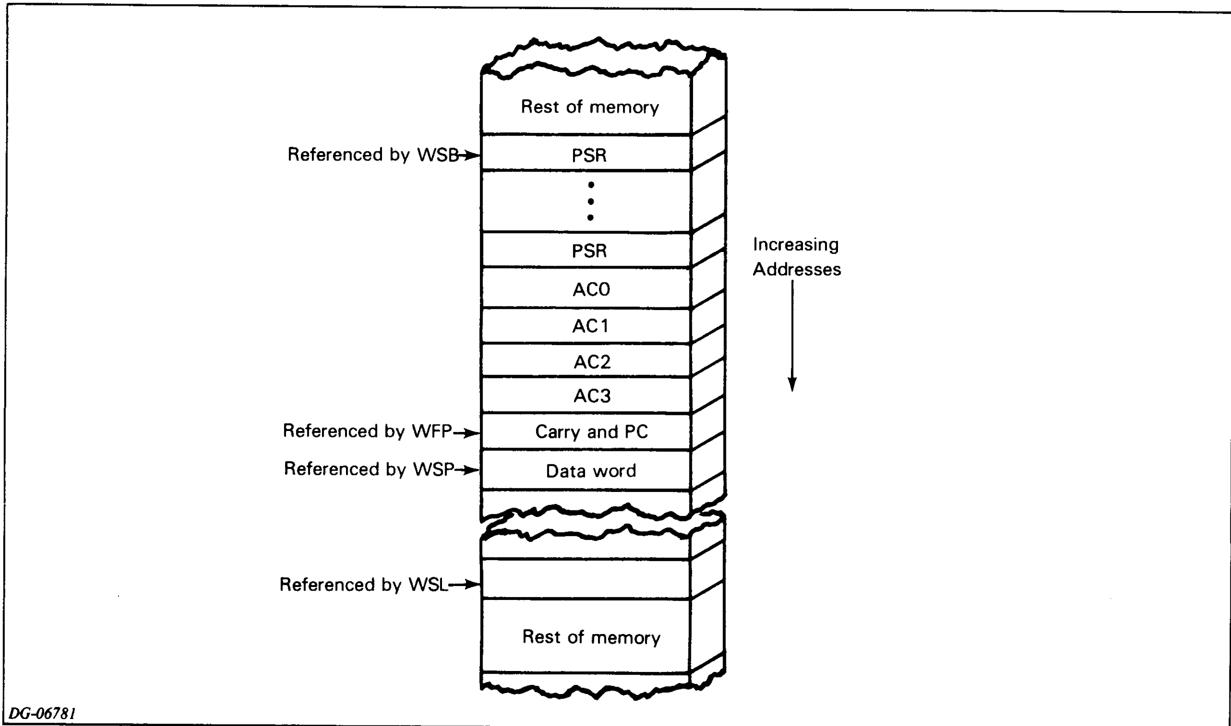


Figure 11.1 A typical wide stack

Wide Stack Instructions

The instructions shown in the tables below manipulate the wide stack. Table 11.1 lists the instructions that operate on the four wide stack registers.

MNEM	Name	Action
LDASP	Load accumulator with WSP	Loads the specified accumulator with the contents of WSP.
STASP	Store accumulator in WSP	Stores the contents of the specified accumulator in WSP.
LDASL	Load accumulator with WSL	Loads the specified accumulator with the contents of WSL.
STASL	Store accumulator in WSL	Stores the contents of the specified accumulator in WSL.
LDASB	Load accumulator with WSB	Loads the specified accumulator with the contents of WSB.
STASB	Store accumulator in WSB	Stores the contents of the specified accumulator in WSB
LDAFP	Load accumulator with WFP	Loads the specified accumulator with the contents of WFP.
STAFP	Store accumulator in WFP	Stores the contents of the specified accumulator in WFP.

Table 11.1 Wide stack register instructions

The instructions shown in Table 11.2 use WSP as a pointer to temporary storage.

MNEM	Name	Action
LDATS	Load accumulator with contents pointed to by WSP	Loads the contents of the word addressed by WSP into the specified accumulator.
STATS	Store accumulator at location pointed to by WSP	Stores the contents of the specified accumulator into the location addressed by WSP.
ISZTS	Increment word addressed by WSP and skip if zero	Increments the double-word addressed by the WSP and skips the next 16-bit word if the incremented value is zero. Indivisible operation.
DSZTS	Decrement word addressed by WSP and skip if zero	Decrements the double-word addressed by the WSP and skips the next 16-bit word if the decremented value is zero. Indivisible operation.

Table 11.2 WSP instructions

The two instructions in Table 11.3 push data onto or pop data off of the wide stack.

MNEM	Name	Action
WPSH	Wide push accumulators	Pushes the contents of the specified accumulators onto the wide stack.
WPOP	Wide pop accumulators	Pops up to four double words off the wide stack and places them in the specified accumulators.

Table 11.3 Wide stack operation instructions

The Narrow Stack

The narrow stack uses four control words to manipulate the stack:

- The stack pointer
- The frame pointer
- The stack limit
- The stack fault address handler

Narrow Stack Pointer

The narrow stack pointer is the address of the current top of the narrow stack. When a single word of data is pushed onto the stack, the processor increments the stack pointer by 1, then places the data to be pushed in the new location addressed by the stack pointer. When a single word of data is popped, the processor takes the single word addressed by the stack pointer and places it in the specified accumulator, then decrements the stack pointer by 1.

The initial value of the stack pointer should be set to one less than the address of the first word in the stack. This determines the *lower limit.*

Location 40_8 contains the current value of the stack pointer.

Narrow Frame Pointer

Unlike the stack pointer, the frame pointer does not change when push or pop operations take place. If the frame pointer is initially set to the same value as the stack pointer, it becomes a useful reference, since it preserves the stack pointer's original value.

The C/350 *Save* and *Return* instructions use the frame pointer to store and reset the value of the narrow stack pointer when entering or exiting subroutines. The frame pointer can also define the boundary between words placed on the narrow stack by different routines in a program. A routine can use the frame pointer to reference back into the narrow stack and retrieve variables left there by the preceding procedure.

Location 41_8 contains the current value of the frame pointer.

The Narrow Stack Limit

The narrow stack limit is the *upper limit* of the narrow stack area. After each push operation, the processor compares the stack pointer to the stack limit. If the stack pointer is greater than the stack limit, an overflow condition exists.

Location 42_8 contains the current value of the stack limit.

Narrow Stack Fault Address

If a narrow stack overflow or underflow occurs, the processor transfers control to the narrow stack fault handler. Location 43_8 contains the (possibly indirect) address of the handler.

A diagram of a narrow stack area is shown in Figure 11.2.

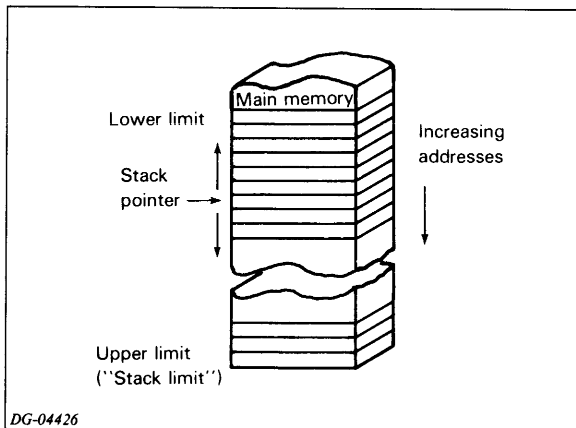


Figure 11.2 The narrow stack area

Return Block Format

The return block can take several forms, but it usually consists of five words: the contents of the four accumulators, the program counter or the frame pointer, and carry in bit 0 of the last word pushed.

Narrow Stack Instructions

The MV/8000 supports the narrow stack instructions shown in Table 11.4.

MNEM	Name	Action
MSP	Modify stack pointer	Changes the value of the stack pointer and checks for potential overflow.
POP	Pop multiple accumulators	Pops up to four words off the narrow stack and places them in the specified accumulators.
POPB	Pop Block	Returns control from a *System Call* routine or an I/O interrupt handler.
POPJ	Pop PC and Jump	Pops the top word off of the narrow stack and places it in the PC.
PSH	Push multiple accumulators	Pushes the contents of up to four accumulators onto the narrow stack.
PSHJ	Push jump	Pushes the address of the next sequential instruction onto the stack and loads the PC with an effective address.
PSHR	Push return address	Pushes the address of this instruction plus two onto the narrow stack.
RSTR	Restore	Returns control from certain types of I/O interrupts.
RTN	Return address	Returns control from subroutines that issue a *Save* instruction at their enrtypoints.
SAVE	Save	Saves the information required by the *Return* instruction.

Table 11.4 Narrow stack instructions

Stack Faults

Wide Stack Faults

The two types of wide stack faults are stack overflow and stack underflow.

Stack Overflow

After every push operation, the processor checks for stack overflow by comparing the contents of **WSP** to the contents of **WSL**. If the contents of **WSP** are greater than the contents of **WSL**, then overflow has occurred.

When an overflow occurs, the processor pushes a wide return block onto the wide stack. The value of the program counter in the return block points to the instruction following the instruction that caused the fault. The processor then sets *OVK*, *OVR*, and *IRES* to 0. After setting bit 0 of **WSP** to 0 and bit 0 of **WSL** to 1, the processor updates the contents of **WSP** and locations $24\text{-}25_8$ of the current segment, then jumps to the stack fault handler via location 14_8. This location contains a 16-bit pointer to the start of the wide stack fault handler. (See also "Stack Fault Codes" below.)

Loading -1 (all one's) into the stack limit disables overflow-caused stack faults.

Stack Underflow

After every pop operation, the processor checks for stack underflow. The processor compares the contents of **WSP** to the contents of **WSB**. If the contents of **WSP** are less than the contents of **WSB**, then underflow has occurred.

When underflow occurs, the processor sets **WSP** equal to **WSL**, then pushes a wide return block. The program counter in the return block points to the instruction immediately following the instruction that cause the fault. The processor sets bit 0 of **WSP** to 0 and bit 0 of **WSL** to 1. It then updates the contents of **WSP** and locations $24\text{-}25_8$ of the current segment and executes a jump indirect to the stack fault handler. Location 14_8 contains the starting address of the stack fault handler.

Loading the most negative signed integer (-2^{31}; i.e., 1 followed by zeroes) into **WSB** disables underflow-caused stack faults.

Stack Fault Codes

When a stack fault occurs, AC0 contains the address of the instruction that caused the fault, and AC1 contains a code describing the type of fault. These codes and their meanings are shown in Table 11.5.

MNEM	Action
0	Overflow on any stack operation except **SAVE**, **WMSP**, or ring crossing.
1	Underflow or overflow would occur if the instructions **WMSP**, **WSAVR**, or **WSAVS** were executed. The PC in the return block references the instruction that caused the stack fault.
2	Too many arguments on a cross-ring call.
3	Stack underflow.
4	Overflow caused by a return block pushed during service of a microinterrupt or fault.

Table 11.5 Wide stack fault codes

Because MV/8000-specific instructions manipulate the wide stack, the processor automatically invokes the wide stack fault handler.

Narrow Stack Faults

Stack overflows and underflows may also occur when using the narrow stack. Narrow stack faults are discussed below.

Overflow

Narrow stack overflow occurs when a program pushes data into the area beyond that allocated for the narrow stack, i.e., beyond the C/350 stack limit. If the locations beyond the address specified by the stack limit are reserved for other purposes, overflow will overwrite whatever information is there.

The narrow stack limit provides overflow protection. If a narrow stack instruction pushes data into the area beyond the location specified by the stack limit, the processor pushes a return block onto the narrow stack and transfers control to the narrow stack fault handler.

To disable overflow protection, set bit 0 of the stack pointer to 0 and bit 0 of the stack limit ot 1.

To be meaningful, the address specified by the narrow stack limit must be 10 to 23 words less than the address of the last word in the narrow stack. This is because the processor can only detect stack overflow at the end of a push operation (with one exception explained below). This means that a 5- to 18-word return block can be pushed onto the narrow stack, starting at the narrow stack limit; the processor would not signal overflow until after it pushed the last word of the return block. After pushing the last word, the processor signals a stack overflow, and the stack fault handler pushes another 5-word return block onto the stack. Depending upon the size of the first return block (from the normal 5 words up to the 18 words used by the floating point instruction set), the potential overflow can be 10 to 23 words long.

Underflow
Narrow stack underflow occurs when a program pops data from the area below that allocated for the narrow stack (i.e., pops more words off than were pushed on). If this occurs, the program will be operating with incorrect and unpredictable information. Furthermore, it is possible that the program will push data into the underflow area, overwriting data or instructions.

For underflow protection to be enabled, the area allocated to the narrow stack must begin at location 401_8, and the narrow stack pointer must be initialized to 400_8. If the narrow stack pointer is less than 400_8 after a pop operation, underflow occurs.

Stack underflow protection can be disabled in two ways:

- Start the narrow stack at a location greater than 401_8.
- Set bit 0 of either the narrow stack pointer or the narrow stack limit to 1.

If the narrow stack starts at a location greater that 401_8, underflow will occur only if the value of the stack pointer is less than 400_8. Note that this does not completely disable underflow protection, since it is always possible to pop enough words off the narrow stack to underflow it.

If bit 0 of the narrow stack pointer or stack limit is set to 1, one of two things will happen: all or part of the stack may reside in page zero (locations 0–377_8), or the stack may underflow into page zero without interference from the narrow stack fault handler.

Narrow Stack Overflow Protection
The C/350 *Save* and *Modify Stack Pointer* instructions check for overflow *before* executing. For every other instruction that pushes data onto the narrow stack, the processor checks for overflow *after* the instruction executes. In both cases, the processor treats the stack pointer and stack limit as unsigned 16-bit integers and compares them. If the comparison shows that overflow has occurred, the processor:

- Sets bit 0 of the narrow stack pointer to 0;
- Sets bit 0 of the narrow stack limit to 1;
- Pushes a return block onto the narrow stack;
- Executes a *jump indirect* to the narrow stack fault address.

The processor sets bit 0 of the narrow stack pointer and stack limit as indicated so that the stack limit will (temporarily) be larger than the stack pointer. The program counter in the return block points to the instruction immediately following the stack instruction that caused the fault (unless the instruction that caused the return block to be pushed is MSP or SAVE; the PC saved in these instructions points to the instruction that caused the fault).

Narrow Stack Underflow Protection
After every pop operation, the processor checks for underflow. If the narrow stack pointer is less than 400_8, and bit 0 of the narrow stack limit is 0, then a narrow stack underflow occurs. When this happens, the processor:

- Sets the narrow stack pointer equal to the narrow stack limit;
- Sets bit 0 of the narrow stack pointer to 0;
- Sets bit 0 of the narrow stack limit to 1;
- Pushes a return block onto the narrow stack;

- Executes a *jump indirect* to the narrow stack fault address.

The processor sets bit 0 of the narrow stack pointer and stack limit as indicated so that the narrow stack limit will (temporarily) be larger than the narrow stack pointer. This ensures that the return block pushed onto the narrow stack by the underflow mechanism (starting at the stack limit) will not cause an overflow fault. The program counter in the return block points to the instruction immediately following the narrow stack instruction that caused the fault.

Narrow Stack Fault Handler

The software narrow stack fault handler determines the nature of the fault. It also resets the appropriate values and takes any other appropriate action, such as allocating more stack space or terminating the program. Note that the narrow stack fault handler must reset bit 0 of the stack pointer and stack limit to their original values.

Examples

Narrow stack area 50_8 words with underflow protection:

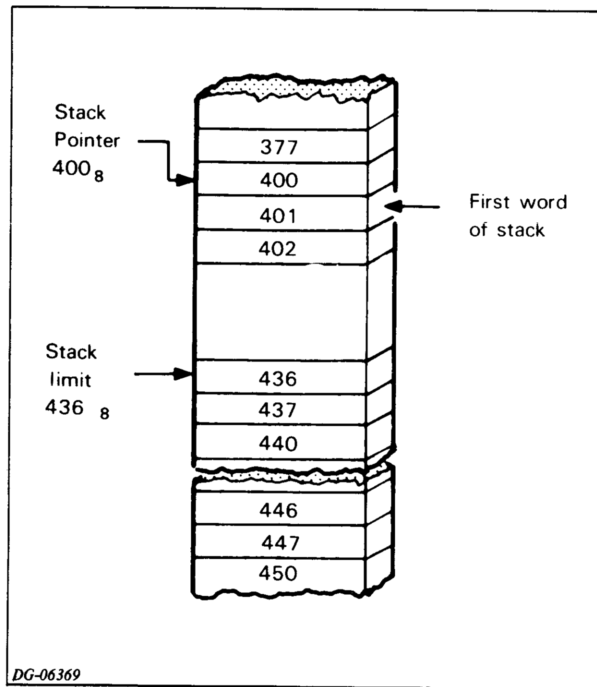


Figure 11.3

Narrow stack area 50_8 words in page zero with overflow protection:

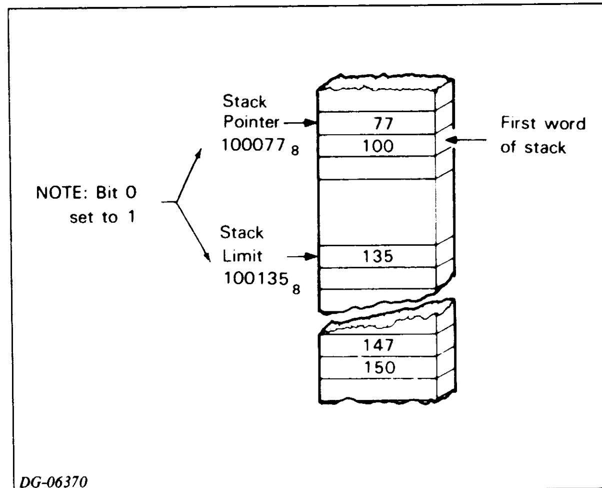


Figure 11.4

Narrow stack area 100_8 words, no protection:

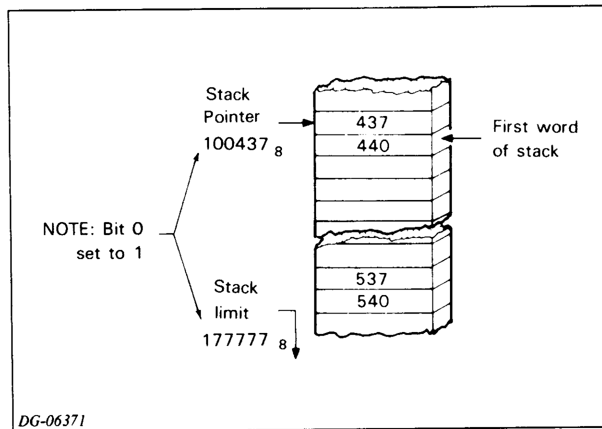


Figure 11.5

Stack Usage

As mentioned in the introduction to this chapter, the MV/8000 allows several stacks to be in use at the same time. This section describes some of the ways the system uses the stack.

The Protection Mechanism

When a ring crossing is performed, the stack parameters in the source segment are preserved and a new wide stack is created in the destination segment (see Chapter 4). To do this, the processor first stores the source segment **WFP** and **WSP** in source segment locations 20–23_8 (**WSL** and **WSB** are assumed to be unchanged). After saving these values, the processor sets up the new stack in the destination segment. It then loads the contents of page zero locations 22–27_8 into the destination's **WSP, WSL,** and **WSB.**

A check for potential stack overflow occurs before any arguments are copied onto the new stack. The number of parameters to be copied is specified by the **LCALL** or **XCALL** instruction that signalled the ring crossing. The processor uses this information to determine if the number of parameters to copy exceeds the size of the stack.

If overflow would occur, no parameters are copied, and a stack fault occurs. AC1 will contain 2. Note that since the ring crossing is valid, the stack fault occurs in the destination segment, not the source segment. The PC contains the address specified in the gate array entry; this is the address of the first instruction to be executed in the destination segment.

If overflow would not occur, then the processor copies the parameters from the source stack to the destination stack. The order of the arguments in the new stack matches the order of the arguments in the old stack. This means that references can be made to parameters exactly as if no ring crossing had occurred.

Fixed Point Overflow

The processor signals fixed point overflow when a division by zero is attempted or when the processor calculates a two's complement number that does not fit in the specified location or register. To signal the overflow, the processor sets *OVR* to 1. Note that *OVK* and *overflow* determine whether a fixed point overflow trap occurs. If *OVK*, *OVR*, and *overflow* contain a 1, then a fixed point overflow trap occurs (unless the instruction currently executing is **SPSR, XVCT, WPOPB, WRSTR, WDPOP, WRTN,** or **WSAVS**). If either *OVK* or *overflow* contains a 0, then no fixed point overflow trap occurs even if *OVR* contains a 1.

The processor signals fixed point overflow at the end of the current instruction cycle. When such a fault occurs, the processor pushes a wide return block onto the wide stack. The block has the format:

Word number in block	Contents
1-2	PSR, 16 0's
3-4	AC0
5-6	AC1
7-8	AC2
9-10	AC3
11-12	Bit 0 = carry, bits 1-31 = return address

Table 11.6 Wide return block format

where the PSR contains *OVR* set to 0, *OVK* set to 1, and *IRES* unchanged, and ***return address*** is the address of the next instruction to be executed.

After the push, AC0 contains the address of the instruction that caused the fault. The processor sets the PSR to 0. Control transfers to the fixed point fault handler through the 16-bit pointer (which may be indirect) contained in location 37_8.

The OVR Flag

If the *OVR* flag is set to 1, then it remains 1 regardless of the new values of *overflow* generated by MV/8000 instructions. This is a convenient feature to use when checking for overflow traps in sections of code. *OVR* is altered, however, when any of the following actions occur:

- Interrupts
- Faults
- Power ups
- I/O or system resets
- Execution of the **WPOPB, WRTN, XCALL, WSSVR, WSSVS, WRSTR,** or **SPSR** instructions.

Note that the processor loads the values of *OVK* and *OVR* produced by the instructions listed above into *OVK* and *OVR*. Unless they *directly* alter them, subsequent instructions will not change these values of *OVR* and *OVK*. Check the individual topics listed for more information about changes to *OVR*.

Floating Point Fault

When a floating point instruction causes a fault, the processor determines whether the instruction that caused the fault is an MV/8000-specific instruction or a C/350 instruction. It does this by checking the values of bit 0 and bits 12-15 of the instruction that caused the fault.

Wide Stack Fault Routine
If bit 0 is 1 and bits 12-15 of the instruction are 1001, the processor pushes a wide return block onto the wide stack. The wide return block has the format shown in table 11.7.

Word number in block	Contents
1-2	PSR, 16 0's
3-4	AC0
5-6	AC1
7-8	AC2
9-10	AC3
11-12	Bit 0 = carry, bits 1-31 = return address

Table 11.7 Wide return block format

The return address in the block is the address of the next user instruction that the processor will execute after servicing the fault. Use the **LFSST** instruction to determine the address of the floating point instruction that caused the fault.

After pushing the return block, the processor sets the FPSR to 0 and the Trap Enable bit to 0. If pushing the return block causes a stack overflow, a stack fault occurs. The processor will service the stack fault before continuing with the floating point fault handler.

If pushing the return block causes no stack fault, the processor continues to execute the floating point fault handler.

Narrow Stack Fault Routine
If the first instruction of the fault handler does not meet the condition specified for bit 0 and bits 12—15 (i.e., either bit 0 is not 1 or bits 12—15 are not 1001), the processor pushes a narrow return block onto the narrow stack. Table 11.8 shows the format of the narrow return block.

Word number in block	Contents
1	AC0
2	AC1
3	AC2
4	AC3
5	Carry in bit 0; return address in bits 1-15

Table 11.8 Narrow return block format

The return address in the return block is the address of the next instruction that will execute after the processor services the fault. Use the **FSST** instruction to determine the address of the floating point instruction that caused the fault.

After pushing the return block, the processor sets the Trap Enable bit to 0. If pushing the return block causes a stack overflow, a stack fault occurs. The processor will service the stack fault before continuing with the floating point fault handler.

If pushing the return block causes no stack fault, the processor continues to execute the floating point fault handler.

Program Flow

As mentioned above, the stack can be used when the sequential order of program execution is changed. The next chapter describes the various methods used to alter sequential instruction execution.

Chapter 12
Program Flow Instructions

The previous chapter mentioned that program flow can be altered using the stack. Another way to alter sequential flow is to jump directly from one place to another. This can be done in two ways: by using a jump instruction, or by using an instruction that tests a conditions and jumps on the result of the test. This type of program flow alteration is called *direct alteration.*

Direct Alteration

In sequential program execution, the processor executes one instruction after another. Sequential flow can be altered directly in one of two ways:

- Use a *Jump* instruction that changes the PC.
- Use a *Conditional Skip* instruction that tests some condition and increments the PC by one if the test is true.

When a jump or conditional instruction changes the PC, program execution continues with the instruction addressed by the new value of the PC. Refer to Figure 12.1.

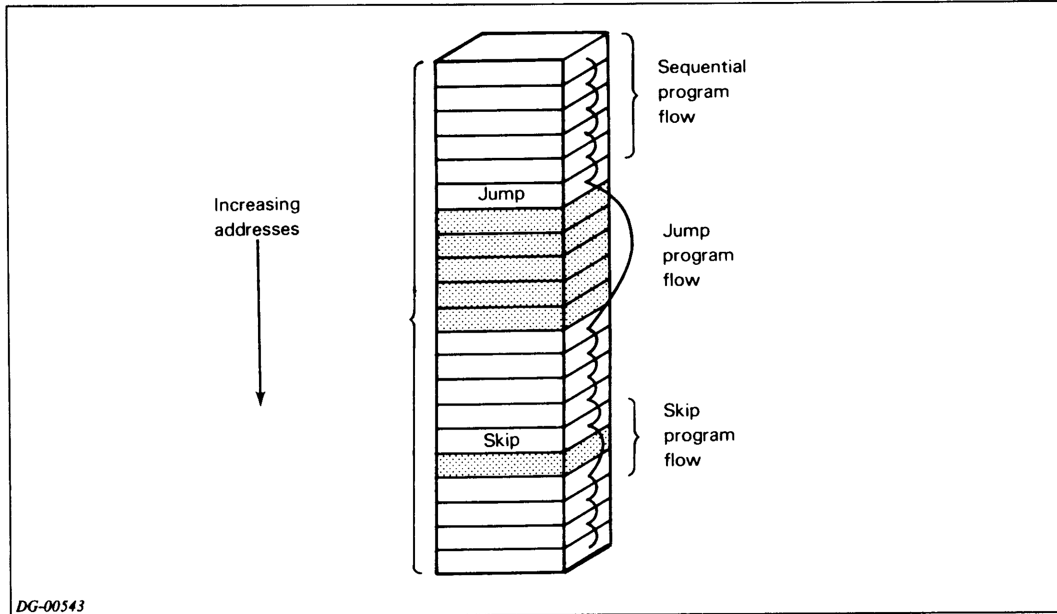


Figure 12.1 Program flow alteration

Note that the term *skip* refers to incrementing the PC by 1, which causes one 16-bit instruction or word to be skipped.

The tables below list the instructions that change program flow by altering the PC. Table 12.1 lists jump instructions and Table 12.2 lists conditional skip instructions.

MNEM	Name	Action
LDSP	Long Dispatch	Dispatches through a table of 28-bit self-relative addresses indexed by the PC.
LJMP	Long Jump	Loads an effective address into the PC.
LJSR	Long Jump to Subroutine	Saves a return address and transfers control to a subroutine.
WBR	Wide Load PC	Adds a specified value to the PC.
WCLM	Wide Compare to Limits	Compares a 32-bit integer to two limit values and skips if the integer is between the two limit values.
XJMP	Extended Jump	Loads an effectve address into the PC.
XJSR	Extended Jump to Subroutine	Saves a return address and transfers control to a subroutine.

Table 12.1 Program flow jump instructions

MNEM	Name	Action
DSZTS	Decrement Word Addressed by ESP and Skip if Zero	Decrements the contents of the word addressed by ESP and skips if the decremented value is zero.
ISZTS	Increment Word Addressed by ESP and Skip if Zero	Increments the contents of the word addressed by ESP and skips if the incremented value is zero.
WSALA	Wide Skip on All Bits Set in Accumulator	Logically ANDs ACS and an immediate field and skips if the result of the AND is zero.
WSALM	Wide Skip on All Bits Set in Memory Location	Logically ANDs an immediate field and memory location and skips if the result of the AND is zero.
WSANA	Wide Skip on Any Bit Set in Accumulator	Logically ANDs ACS and an immediate field and skips if the result of the AND is nonzero.
WSANM	Wide Skip on Any Bit Set in Memory Location	Logically ANDs an immediate field and memory location and skips if the result of the AND is zero.
NSALA	Narrow Skip on All Bits Set in Accumulator	Logically ANDs ACS and an immediate field and skips if the result of the AND is zero.
NSALM	Narrow Skip on All Bits Set in Memory Location	Logically ANDs an immediate field and a memory location and skips if the result of the AND is zero.
NSANA	Narrow Skip on Any Bit Set in Accumulator	Logically ANDs ACS and an immediate field and skips if the result of the AND is non-zero.
NSANM	Narrow Skip on Any Bit Set in Memory Location	Logically ANDs an immediate field and a memory location and skips if the result of the AND is zero.
SNOVR	Skip on *OVR* Reset	Skips if *OVR* is 0.
WSEQ	Wide Skip if Equal	Compares ACS to ACD and skips if the two values are equal.
WSGE	Wide Skip if Greater Than or Equal	Compares ACS to ACD and skips if ACS is greater than, or, equal to ACD.
WSGT	Wide Skip if Greater	Compares ACS to ACD and skips if ACS is greater than ACD.
WSKBO	Wide Skip on Bit Set to One	Tests a bit in AC0 and skips if the bit is one.
WSKBO	Wide Skip on Bit Set to Zero	Tests a bit in AC0 and skips if the bit is zero.

Table 12.2 Program flow conditional skip instructions

The MV/8000 also supports the C/350 program flow instructions listed in the tables below. Table 12.3 shows C/350 jump instructions and Table 12.4 shows C/350 conditional skip instructions.

MNEM	Name	Action
CLM	Compare to Limits	Compares a signed 16-bit integer with two other numbers and skips if the first integer is between the other two.
DSPA	Dispatch	Compares a signed integer with two other numbers and skips if the first integer is not between the others; otherwise, uses the integer as an index into a table and places the indexed value in the program counter.
JMP, EJMP	Jump	Loads an effective address in the program counter.
JSR, EJSR	Jump to Subroutine	Increments the program counter and stores the incremented value in AC3; then places a new address in the program counter.
SYC,	System	Turns the MAP off if on. Pushes a
SCL, SVC	Call	return block onto the stack and places the address of the System Call handler in the program counter.
XCT	Execute	Executes the contents of an accumulator as an instruction.

Table 12.3 C/350 program flow jump instructions

MNEM	Name	Action
DSZ, EDSZ	Decrement and Skip if Zero	Decrements the addressed word, then skips if the decremented value is zero.
ISZ, EISZ	Increment and Skip if Zero	Increments the addressed word, then skips if the incremented value is zero.
SGE	Skip if ACS Greater Than or Equal to ACD	Compares two signed integers in two accumulators and skips if the first is greater than or equal to the second.
SGT	Skip if ACS Greater Than ACD	Compares two signed integers in the accumulators; skips if the first is greater than the second.
SKP/*t*/	I/O Skip	Skips if the I/O condition *t* is true.
SNB	Skip on Non-zero Bit	References a single bit in memory via a bit pointer; skips if the bit is 1.
SZB	Skip on Zero Bit	References a single bit in memory via a bit pointer; skips if the bit is 0.
SZBO	Skip on Zero Bit, Set to 1	References a single bit in memory via a bit pointer; skips if the bit is 0 and also sets the bit to 1.

Table 12.4 C/350 program flow conditional skip instructions

Stack Changes

Internal conditions, such as I/O interrupts, may interrupt the normal flow of a program at some point. When this occurs, the processor saves the address of the next instruction in the program. This will enable the processor to return to the correct place in the program after it services the interrupting condition.

After saving the correct return address, the processor places the starting address of the proper fault or interrupt handler in the PC. Sequential operation continues with the handler.

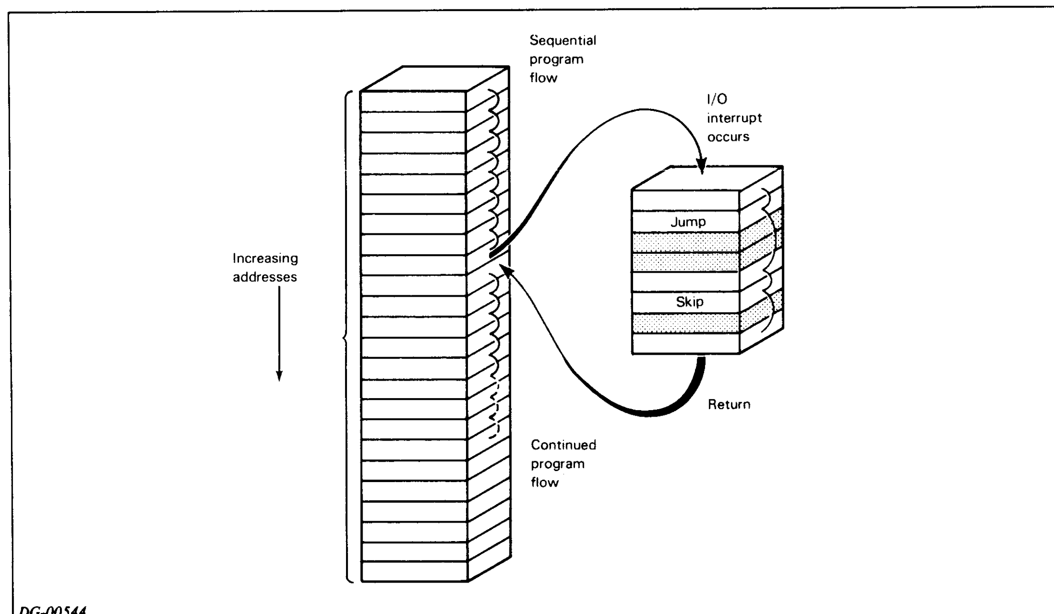


Figure 12.2 Program flow interruption

Figure 12.2 shows how the processor transfers control from normal flow to an I/O interrupt handler and back. Table 12.5 lists the wide stack manipulation instructions.

MNEM	Name	Action
BKPT	Breakpoint	Pushes a return block onto the wide stack and performs a jump indirect through locations 10–11.
LCALL	Call Subroutine (long displacement)	Evaluates the address of a subroutine and jumps to the subroutine if the address is valid.
LDSP	Long Dispatch	Dispatches through a table of 28-bit addresses indexed by the PC.
LPSHJ	Long Push Jump	Pushes the PC onto the wide stack and jumps to a subroutine.
PBX	Pop Block and Execute	Pushes a return block onto the wide stack and executes a specified instruction.
WBR	Wide Load PC	Adds a specified value to the PC.
WDPOP	Pop MV/8000 Context Block	Restores the state of the CPU to what it was at the time of the last page fault. This is a privileged instruction.
WPOP	Wide Pop	Pops up to four double-words from the wide stack and places them in accumulators.
WPOPB	Wide Pop Block	Pops six double-words from the wide stack and places them in specified locations.
WPOPJ	Wide Pop Jump	Pops a double-word from the wide stack and places it in the PC.
WPSH	Wide Push Accumulators	Pushes the contents of the accumulators onto the wide stack.
WRSTR	Wide Restore	Returns control from an interrupt.
WRTN	Wide Return	Returns control from a subroutine.
XCALL	Call Subroutine (extended displacement)	Evaluates the address of a subroutine and jumps to the subroutine if the address is valid.
XPSHJ	Extended Push Jump	Pushes the PC onto the wide stack and jumps to a subroutine.

Table 12.5 Wide stack manipulation instructions

Although some MV/8000-specific program flow instructions have 31-bit displacements, usually only 28 bits are significant. Only the **WPOPB, WRSTR, XCALL, LCALL** and **WRTN** instructions can legally effect the current segment number.

For all other program flow instructions, only the least significant 28 bits of the PC are altered. The segment field of the effective address is ignored. This means that all references will be within the current segment.

If the effective address is a pointer obtained via indirection, the least signifcant 28 bits of the pointer replace the least signifcant 28 bits of the PC.

In addition to the MV/8000-specific program flow instructions, the C/350 program flow instructions shown in the following tables are available.

MNEM	Name	Action
POPJ	Pop PC and Jump	Pops the top word off the stack and places it in the program counter.
PSHJ	Push	Pushes the address of the next sequential instruction onto the stack and places a new address in the program counter.
RSTR	Restore	Returns control from I/O interrupt handlers that use the stack change facility of the **VCT** instruction.
RTN	Return	Returns control from a subroutine entered via the *Save* instruction.
VCT	Vector on Interrupting Device Code	Identifies the highest priority interrupt; passes control through a table to a handler routine for the device.

Table 12.6 Narrow stack manipulation instructions

The extended operation feature (XOP) provides an efficient method of transferring control to and from procedures. It allows control to transfer to any one of 32 procedure entry points. The instructions that invoke the XOP feature are shown in Table 12.7.

MNEM	Name	Action
WXOP	Extended Operation	Pushes a return block on the wide stack, placing the address in the stack of the specified accumulators into AC2 and AC3, and transfers control to one of 32 other procedures via the XOP table.
WXOP1	Extended Operation	Same as **WXOP** except that 32 is added to the entry number before entering the XOP table, and only 16 table entries can be specified.

Table 12.7 Extended operation instructions

Table 12.8 shows the C/350 XOP instruction.

MNEM	Name	Action
XOP0	Extended Operation	Pushes a return block onto the narrow stack, indexes into the XOP table and transfers control to another procedure.

Table 12.8 C/350 extended operation instruction

Table 12.9 lists the skip instructions that test condition codes in the floating point status register.

MNEM	Name	Action
FNS	No skip	Executes the next sequential word.
FSA	Skip always	Skips the next sequential instruction.
FSEQ	Skip on zero	Skips the next sequential word if the Z flag in the FPSR is 1.
FSGE	Skip on greater than or equal to zero	Skips the next sequential word if the N flag of the FPSR is 0.
FSGT	Skip on greater than zero	Skips the next sequential word if both the Z and N flags of the FPSR are 0.
FSLE	Skip on less than or equal to zero	Skips the next sequential word if either the Z flag or the N flag of the FPSR is 1.
FSLT	Skip on less than zero	Skipss the next sequential word if the N flag of the FPSR IS 1.
FSND	Skip on no zero divide	Skips the next sequential word if the divide by zero (DVZ) flag of the FPSR is 0.
FSNE	Skip on non-zero	Skips the next sequential word if the Z flag of the FPSR is 0.
FSNER	Skip on no error	Skips the next sequential word if bits 1–4 of the FPSR are all 0.
FSNM	Skip on no mantissa overflow	Skips the next sequential word if the mantissa overflow (MOF) flag of the FPSR is 0.
FSNO	Skip on no overflow	Skips the next sequential word if the overflow (OVF) flag of the FPSR is 0.
FSNOD	Skip on no overflow and no zero divide	Skips the next sequential word if both the overflow (OVF) flag and the divide by zero (DVZ) flag of the FPSR are 0.
FSNU	Skip on no underflow	Skips the next sequential word if the underflow (UNF) flag of the FPSR is 0.
FSNUD	Skip on no underflow and no zero divide	Skips the next sequential word if both the underflow (UNF) flag and the divide by zero (DVZ) flag of the.FPSR are 0.
FSNUO	Skip on no underflow and no overflow	Skips the next sequential word if both the underflow (UNF) flag and the overflow (OVF) flag of the FPSR are 0.

Table 12.9 Floating point test instructions

Table 12.10 lists the condition tests available for the **SKIP***[t]* instruction. (This instruction tests condition codes of a peripheral device, the power-fail monitor or the interrupt system.)

Symbol	Value	Test
[t] = **BN**	00	Tests Busy flag for nonzero.
[t] = **BZ**	01	Tests Busy flag for zero.
[t] = **DN**	10	Tests Done flag for nonzero.
[t] = **DZ**	11	Tests Done flag for zero.

Table 12.10 Skip instruction test conditions

Table 12.11 summarizes *skip* options of the C/350 ALC instructions.

Symbol	Value	Operation
[skip] omitted	000	No skip.
[skip] = **SKP**	001	Skip unconditionally.
[skip] = **SZC**	010	Skip if carry is zero.
[skip] = **SNC**	011	Skip if carry is nonzero.
[skip] = **SZR**	100	Skip if ALC result is zero.
[skip] = **SNR**	101	Skip if ALC result is nonzero.
[skip] = **SEZ**	110	Skip if either ALC result. or carry is zero.
[skip] = **SBN**	111	Skip if both ALC result. and carry is nonzero.

Table 12.11 ALC skip options

Chapter 13
System Control Instructions

System control instructions fall into two groups: privileged instructions, and queue instructions. Privileged instructions can be executed only when the current segment is Segment 0. Queue instructions, however, can be executed in any segment.

Privileged Instructions

The privileged instructions alter the contents of the SBRs, purge the ATU, reset the referenced and modified bits, and perform other system functions. These instructions can be executed from Segment 0 only. If a privileged instruction is executed from any other ring, a protection fault occurs. AC1 will contain the code 9 when such a fault occurs.

MNEM	Name	Action
LSBRA	Load all segment base registers	Loads new information into all eight SBRs.
LSBRS	Load some segment base registers	Loads new information into SBR1 through SBR7.
PATU	Purge ATU	Purges the ATU of all entries.
RRFB	Reset the referenced bit	Loads a number of the referenced bits with zero's.
ORFB	OR the referenced bits	Inclusively ORs a number of the referenced bits with a bit string and stores the result in a second bit string.
LMRF	Load the modified and referenced bits	Loads the values contained in a number of modified and referenced bits into AC1.
SMRF	Store the modified and referenced bits	Stores the values specified by AC1 into a number of modified and referenced bits.
WDPOP	Pop the context block	Restores the state of the processor to what it was at the time of the last page fault.
XVCT	MV/8000 vector on interrupting device	Identifies the highest priority interrupt and passes control through a table to a device handler.

Table 13.1 Privileged instructions

Note that when the ATU is enabled, the C/350 **LMP, SYC** or any of the MAP instructions cannot be executed. An attempt to execute these instructions when the ATU is enabled will result in the same type of protection fault described above.

Table 13.1 lists the privileged instructions and describes them briefly.

Queues

A *queue* is a variable-length list of linked entries that has a beginning and an end. The operating system uses queues to keep track of processes that it must run (ready queue), files that must be printed on the line printer, pages that are resident in physical memory, etc.

An entry in a queue is called a *data element*. Adding a data element to a queue is called *enqueuing*. Removing a data element is called *dequeuing*. The ends of a queue are called the *head* and the *tail*. A typical first in, first out (FIFO) queue has data elements enqueued at the tail and dequeued at the head.

One of the advantages of using a queue rather than a single threaded list is that queue data elements reference the data elements that precede *and* follow them. In other words, MV/8000 queues use a priority-based structure. This means that data elements can be *enqueued* anywhere in the queue, not just at the head. Conversely, data elements can be *dequeued* anywhere in the queue, not just at the tail.

New entries are added to the queue when service, (such as the name of a new file to be printed) is required, and they are removed from the queue after they are of no further use. A queue may be empty, it may have only one entry, or it may have many entries.

Building a Queue

For the data elements to be linked together, each data element must contain two pointers, called *links*. One of the links contains the effective word address of the following data element in the queue: the *forward link*. The other link contains the effective word address of the preceding data element in the queue: the *backward link*.

The forward and backward links do more than reference the adjacent queue data elements. They also indicate the elements that are currently at the head and tail of the queue. If a data element's forward link contains a –1, then that data element is at the tail of the queue. If a data element's backward link contains a –1, then that data element is at the head of the queue. Note that a data element containing –1 in both its forward and backward links is the only data element currently in the queue.

A data element contains user information as well as the forward and backward links. This user information can either precede or follow the forward and backward links, as shown in Tables 13.2 and 13.3. The structure and the meaning of the information is determined by the user.

Position in data element	Contents
First double-word	Forward link.
Second double-word	Backward link.
Next *n* double-words	User information.

Table 13.2 Data element with user data following links

Position in data element	Contents
First *n* double-words	User information.
(*n* + 1)th double-word	Forward link.
(*n* + 2)th double-word	Backward link.

Table 13.3 Data element with user data preceding links

Also, note that the length of the user information in the data elements can vary, since the links of each data element always reference other links and not user information. The *Search Queue* instructions, however, do reference the user information, so make sure that any programs using these instructions take the length of the user information into account.

Queue Descriptor

Each queue uses a *queue descriptor* that indicates the current head and tail of the queue. A queue descriptor is two 32-bit words. The first double-word contains the address of the data element that is currently at the head of the queue; the second contains the address of the data element that is currently at the tail of the queue.

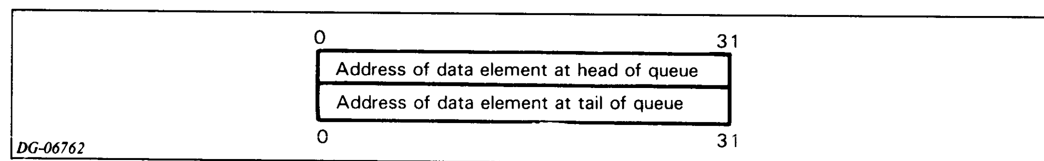


Figure 13.1 Format of queue descriptor

Setting Up and Modifying a Queue

To define an empty queue, create a queue descriptor that contains –1 in both of its pointers. To enqueue a data element into the empty queue, load the address of the data element into both double-words of the queue descriptor (indicating a one element queue) and load –1 into the data element's forward and backward links. To enqueue or dequeue a data element anywhere in the queue, specify the queue descriptor and the address of some data element in the queue. The descriptor and address specified acts as a reference point that the processor uses to enqueue the data element at the right point, or to dequeue the appropriate data element.

Note that a new one-element queue can be created in one step. To create a one-element queue, create a queue descriptor that contains the address of a data element in both double-words. Then load both of the links of the particular data element with –1.

Examples

The examples below demonstrate how queues are formed, how enqueuing and dequeuing works, and how the processor updates the various links and descriptors.

Queue Descriptor of an Empty Queue
Figure 13.2 shows the queue descriptor for an empty queue.

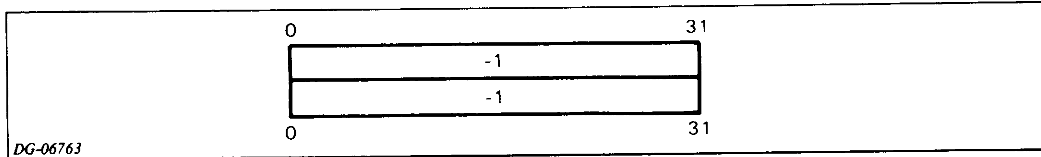


Figure 13.2 Queue descriptor for an empty queue

Enqueuing a Data Element into an Empty Queue

The second example enqueues a data element (located at location A) into an empty queue. (See Figure 13.3.)

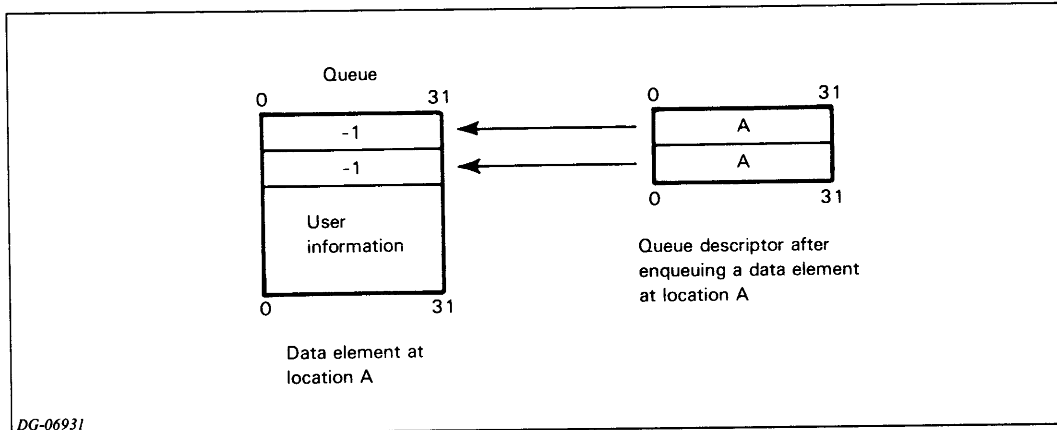


Figure 13.3

The processor has enqueued data element A into the queue, and updated the queue descriptor. The descriptor shows that the queue has only one element, A. At location A, the first word of the data element contains the forward link of –1. The last word contains the backward link of –1.

Enqueuing a Data Element at the Head of a Queue

The third example enqueues a data element (located at location B) at the head of the queue before data element A. (See Figure 13.4.)

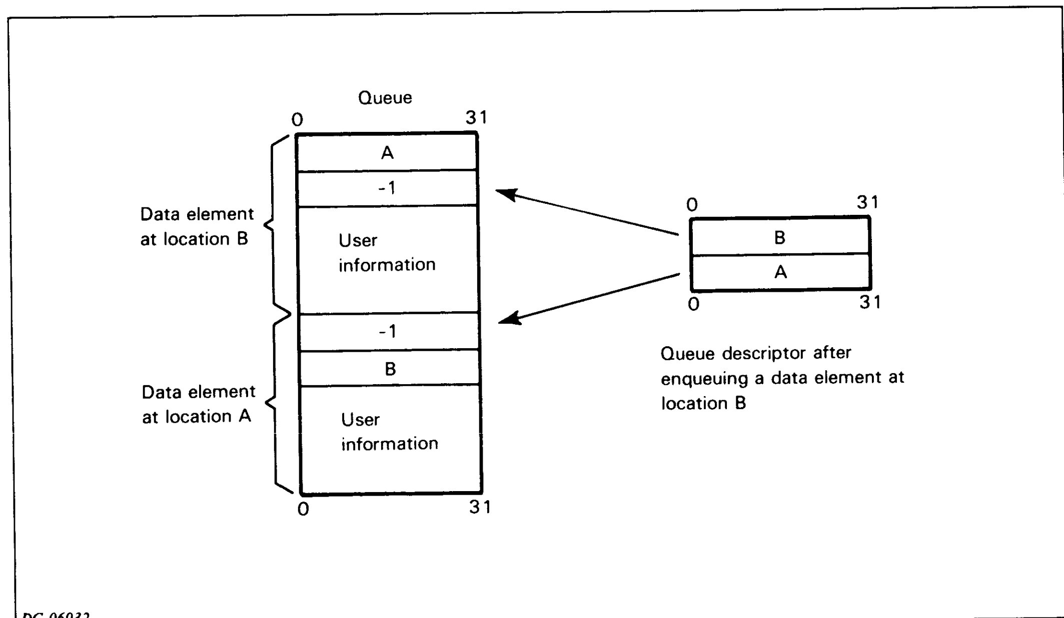


Figure 13.4

After the enqueue, the processor updates the queue descriptor to reference the new head and tail. It also changes the backward link of data element A to reference the preceding data element (B). The links of data element B show that it is the head of the queue, and that data element A follows it.

Enqueuing a Data Element at the Tail of a Queue

The fourth example enques a data element (located at location C) at the tail of the queue, after data element A. (See Figure 13.5.)

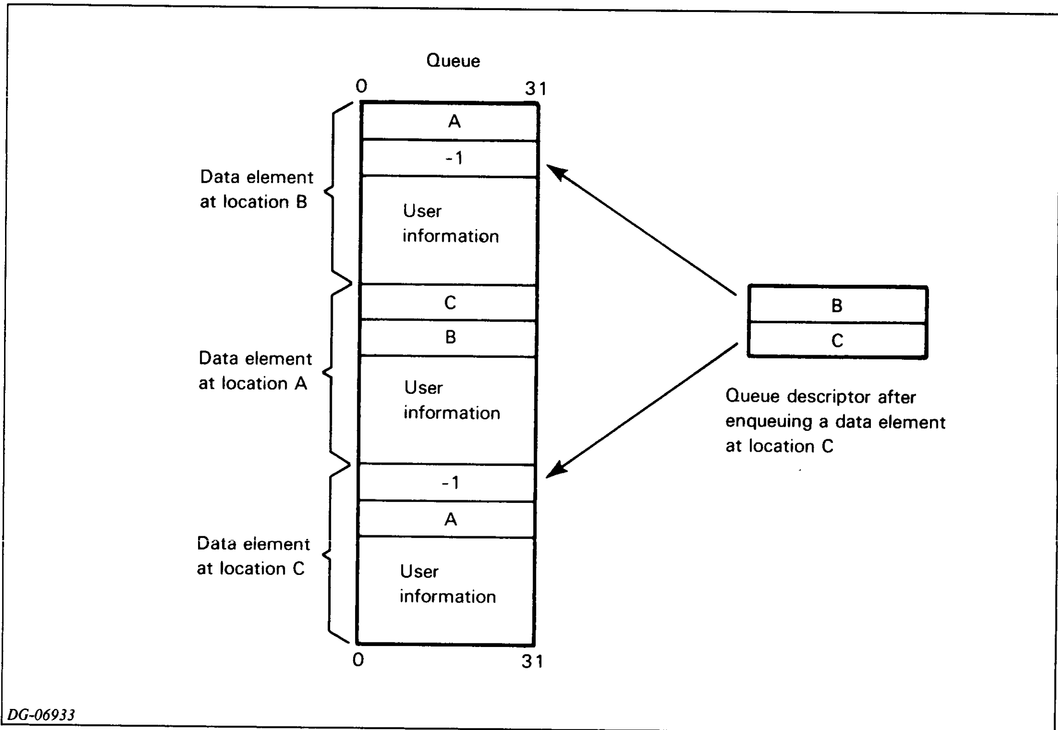


Figure 13.5

The –1 in data element B's backward link shows that B is the head of the queue. The –1 in data element C's forward link shows that C is the tail of the queue. The queue descriptor also indicates the new head and tail of the queue.

Dequeueing a Data Element

The last example dequeues data element B. (See Figure 13.6.)

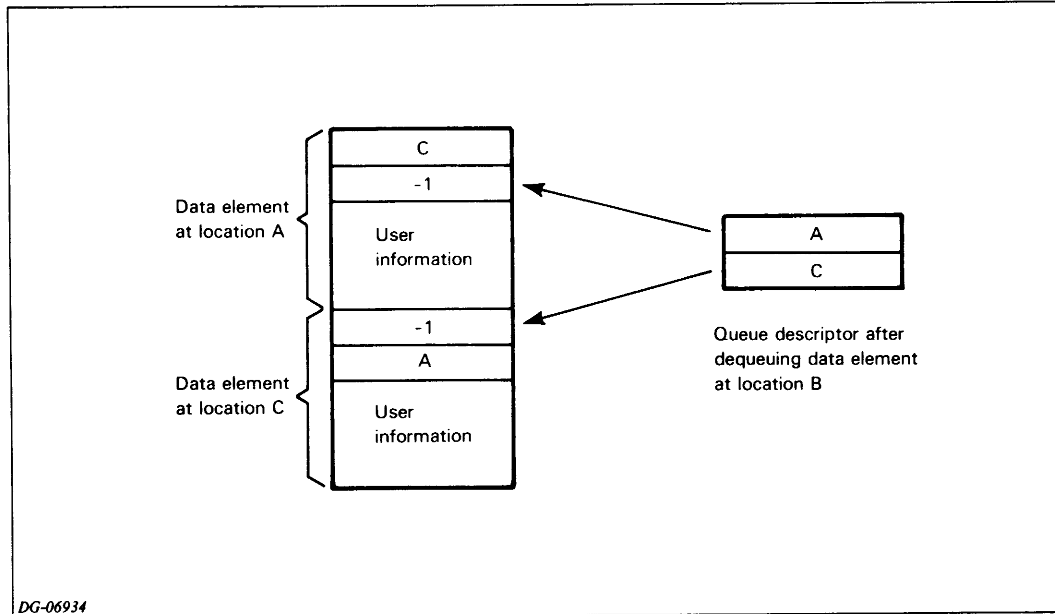


Figure 13.6
The processor dequeues data element B from the queue and updates the queue descriptor to show the new head (A). A's backward link shows that it is the new head. C's links remain unchanged, since C is still the tail of the queue, and A is still the following data entry.

Queue Instructions

The MV/8000 uses the instructions shown in Table 13.4 to manipulate queues. Two of the instructions enqueue data elements onto queues and one dequeues data elements. The remaining 32 instructions perform queue searches.

MNEM	Name	Action
ENQH	Enqueue towards the head	Places a data element at the beginning of the specified queue.
ENQT	Enqueue towards the tail	Places a data element at the end of the specified queue.
DEQUE	Dequeue a queue data element	Removes a data element from the specified queue.
NBSSS	Search queue	Performs a queue search. The direction of the search is backwards. The field to search is 16 bits wide. Searches for a data element and sets some of the bits to reflect the contents of the mask.
NBSSC	Search queue	Performs a queue search. The direction of the search is backwards. The field to search is 16 bits wide. Searches for a data element and clears some of the bits to reflect the contents of the mask.
NBSAS	Search queue	Performs a queue search. The direction of the search is backwards. The field to search is 16 bits wide. Searches for a data element and sets all the bits in the word to reflect contents of the mask.
NBSAC	Search queue	Performs a queue search. The drection of the search is backwards. The field to search is 16 bits wide. Searches for a data element and clears all the bits in the word to reflect contents of the mask.
NBSE	Search queue	Performs a queue search. The direction of the search is backwards. The field to search is 16 bits wide. Searches for a data element that contains data equal to the contents of the mask.
NBSGE	Search queue	Performs a queue search. The direction of the search is backwards. The field to search is 16 bits wide. Searches for a data element that contains data greater than or equal to the contents of the mask.
NBSLE	Search queue	Performs a queue search. The direction of the search is backwards. The field to search is 16 bits wide. Searches for a data element that contains data less than or equal to the contents of the mask.
NBSNE	Search queue	Performs a queue search. The direction of the search is backwards. The field to search is 16 bits wide. Searches for a data element that contains data not equal to the contents of the mask.
WBSSS	Search queue	Performs a queue search. The direction of the search is backwards. The field to search is 32 bits wide. Searches for a data element and sets some of the bits to reflect the contents of the mask.

Table 13.4 Queue instructions

MNEM	Name	Action
WBSSC	Search queue	Performs a queue search. The direction of the search is backwards. The field to search is 32 bits wide. Searches for a data element and clears some of the bits to reflect the contents of the mask.
WBSAS	Search queue	Performs a queue search. The direction of the search is backwards. The field to search is 32 bits wide. Searches for a data element and sets all the bits in the word to reflect the contents of the mask.
WBSAC	Search queue	Performs a queue search. The direction of the search is backwards. The field to search is 32 bits wide. Searches for a data element and clears all the bits in the word to reflect the contents of the mask.
WBSE	Search queue	Performs a queue search. The direction of the search is backwards. The field to search is 32 bits wide. Searches for a data element that contains data equal to the contents of the mask.
WBSGE	Search queue	Performs a queue search. The direction of the search is backwards. The field to search is 32 bits wide. Searches for a data element that contains data greater than or equal to the contents of the mask.
WBSLE	Search queue	Performs a queue search. The direction of the search is backwards. The field to search is 32 bits wide. Searches for a data element that contains data less than or equal to the contents of the mask.
WBSNE	Search queue	Performs a queue search. The direction of the search is backwards. The field to search is 32 bits wide. Searches for a data element that contains data not equal to the contents of the mask.
WFSSS	Search queue	Performs a queue search. The direction of the search is forwards. The field to search is 32 bits wide. Searches for a data element and sets some of the bits to reflect the contents of the mask.
WFSSC	Search queue	Performs a queue search. The direction of the search is forwards. The field to search is 32 bits wide. Searches for a data element and clears some of the bits to reflect the contents of the mask.
WFSAS	Search queue	Performs a queue search. The direction of the search is forwards. The field to search is 32 bits wide. Searches for a data element and sets all the bits in the word to reflect the contents of the mask.
WFSAC	Search queue	Performs a queue search. The direction of the search is forwards. The field to search is 32 bits wide. Searches for a data element and clears all the bits in the word to reflect the contents of the mask.
WFSE	Search queue	Performs a queue search. The direction of the search is forwards. The field to search is 32 bits wide. Searches for a data element that contains data equal to the contents of the mask.

Queue instructions (cont.)

MNEM	Name	Action
WFSGE	Search queue	Performs a queue search. The direction of the search is forwards. The field to search is 32 bits wide. Searches for a data element that contains data greater than or equal to the contents of the mask.
WFSLE	Search queue	Performs a queue search. The direction of the search is forwards. The field to search is 32 bits wide. Searches for a data element that contains data less than or equal to the contents of the mask.
WFSNE	Search queue	Performs a queue search. The direction of the search is forwards. The field to search is 32 bits wide. Searches for a data element that contains data not equal to the contents of the mask.
NFSSS	Search queue	Performs a queue search. The direction of the search is forwards. The field to search is 16 bits wide. Searches for a data element and sets some of the bits to reflect the contents of the mask.
NFSSC	Search queue	Performs a queue search. The direction of the search is forwards. The field to search is 16 bits wide. Searches for a data element and clears some of the bits to reflect the contents of the mask.
NFSAS	Search queue	Performs a queue search. The direction of the search is forwards. The field to search is 16 bits wide. Searches for a data element and sets all the bits in the word to reflect the contents of the mask.
NFSAC	Search queue	Performs a queue search. The direction of the search is forwards. The field to search is 16 bits wide. Searches for a data element and clears all the bits in the word to reflect the contents of the mask.
NFSE	Search queue	Performs a queue search. The direction of the search is forwards. The field to search is 16 bits wide. Searches for a data element that contains data equal to the contents of the mask.
NFSGE	Search queue	Performs a queue search. The direction of the search is forwards. The field to search is 16 bits wide. Searches for a data element that contains data greater than or equal to the contents of the mask.
NFSLE	Search queue	Performs a queue search. The direction of the search is forwards. The field to search is 16 bits wide. Searches for a data element that contains data less than or equal to the contents of the mask.
NFSNE	Search queue	Performs a queue search. The direction of the search is forwards. The field to search is 16 bits wide. Searches for a data element that contains data not equal to the contents of the mask.

Queue instructions (cont.)

Chapter 14
Input/Output

The MV/8000 has a comprehensive I/O structure that combines three separate systems monitored by a central I/O controller. Programmed I/O allows the transfer of individual pieces of data. Data channel I/O allows the transfer of blocks of information between medium-speed I/O devices and memory. Burst multiplexor channel I/O is available for transferring blocks of data quickly between high-speed I/O devices and memory.

The MV/8000 has built-in I/O devices that provide services to the system. The programmable interrupt timer allows the generation of interrupts at selected intervals. The real-time clock provides accurate timing information. The asynchronous line controller connects the host to the console.

The I/O System

The MV/8000 has a 6-bit device code corresponding to bits 10–15 in the I/O instruction format. The devices are connected to the I/O system in such a way that each device will only respond to commands sent with its own device code. With a 6-bit device code, 64 devices can be individually controlled. Some of these device codes are reserved for the processor and certain options, but the rest are available for referencing I/O devices. The assembler recognizes mnemonics for those devices assigned a code by Data General. A complete list of these is provided in Appendix A of this manual.

Programmed I/O

Programmed I/O transfers data one word or part of a word at a time under direct program control. This type of I/O allows data to be examined piece by piece as it is transferred.

The MV/8000 executes all C/350 programmed I/O instructions exactly as the ECLIPSE C/350 does.

Data Channel I/O

Data channel I/O permits data to be transferred in blocks of words, with program control necessary only at the start and end of the operation. The transfer is made directly to or from memory via the system cache; no additional steps are required. Data channel I/O is an efficient method of transferring blocks of data between memory and a

medium-speed I/O device.

Data channel transfers are set up by a program that specifies the address of the first word to transfer and the total number of words to transfer. The program then specifies if a read or write is to take place. Once the device has these parameters, the transfer takes place in two phases. In phase 1, the device specifies the address of the word to transfer and the direction of the transfer. In phase 2, the device transfers the contents of the specified address. These two phases are repeated until the total number of words have been transferred.

When a data channel device is ready to send or receive data, it issues a data channel request. At the beginning of every memory cycle, the I/O channel synchronizes any requests that are then being made and controls the transfers between the I/O bus and the system cache. When a request is honored, a word is transferred directly between the device and memory via the data channel.

To map the data channel, use the **WLMP** instruction. Note that the I/O processor can change the MV/8000 data channel map without host intervention. For more information, see *Changing the Host Data Channel Map From the IOP* in Chapter 15.

Burst Multiplexor I/O

The burst multiplexor channel transfers blocks of data directly between devices and the system cache via the BMC bus. A given block of data is transferred in sub-blocks of up to 256 words. Like the data channel, the BMC transfers are set up by a program that specifies the address of the first word in the block to transfer, and the total number of words in the block. The program then specifies if a read or write is to take place. Once the device has these parameters, the transfer of the sub-blocks takes place in two phases. In phase 1, the device specifies the starting address of the sub-block to transfer, the number of words in the sub-block, and the direction of the transfer. In phase 2, the device transfers the sub-block. These two phases are repeated until all of the sub-blocks have been transferred.

The burst multiplexor channel has two address modes. In unmapped mode, the device transmits a 21-bit word address to the BMC. The BMC passes this address directly to the system cache.

In mapped mode, the device sends a 20-bit word address to the BMC. The BMC uses its map tables to convert the 10 high-order bits of the logical address to a 14-bit physical page number. The BMC then concatenates this page number to the 10 low-order bits of the logical address to form a physical address.

Note that the **CIO, CIOI**, and all BMC programmed I/O instructions specify an 11-bit physical page address. The **WLMP** instruction specifies a 14-bit physical page address. It is convenient to use **WLMP** when dealing with a very large number of physical pages, although it can be used even if the system contains less than 14 bits worth of pages.

Busy and Done Flags

I/O devices are controlled by manipulating their Busy and Done flags. However, some devices require that several programmed I/O instructions are properly set up before the devices can be started with the flags. The value of these flags can be changed by appending optional flag control command mnemonics to the instruction. When Busy and Done are both 0, the device is idle. To start a device, the program sets Busy to 1 and Done to 0. When the device has finished its operation and is ready to start another, it sets

Busy to 0 and Done to 1.

Interrupt On Flag

The processor uses the Interrupt On (ION) flag to control the status of the interrupt system. If the flag is set to 1, the processor responds to and services interrupts. If the flag is 0, the processor ignores all incoming interrupt requests and does not service them.

Priority Mask

As mentioned above, the MV/8000 uses a priority-based interrupt system. To control the priorities, the processor uses a priority mask. Each I/O device is associated to one of 16 bits in the priority mask. (Note that more than one device can be associated with each bit.) When a bit in the priority mask is 1, the devices associated with that bit are inhibited from making an interrupt request, even if their Busy flags are 0 and their Done flags are 1. Because the mask can be changed in a program (see the *Mask out* instruction), different devices can be inhibited at different times according to the need.

I/O Instructions

Some I/O instructions have special mnemonics that can be used in place of the standard mnemonics. Note that the mnemonics for controlling the state of flags cannot be appended to these special instruction mnemonics. For example, to alter the state of the ION flag while performing a *Mask Out* instruction, use the full mnemonic:

DOB*f* *ac*,**CPU**

instead of the special mnemonic:

MSKO *ac*

This special mnemonic sets bits 8 and 9 to **00**.

Table 14.1 describes the MV/8000-specific instructions, and Table 14.2 describes the C/350 I/O instructions.

MNEM	Name	Action
CIO	Command I/O	Performs a read or write data operation on the I/O system bus.
CIOI	Command I/O Immediate	Performs a read or write data operation on the I/O system bus.
PIO	Program I/O out A	Performs a specified operation on the I/O system bus.
WLMP	Wide Load Map	Loads information into the the specified map slots.

Table 14.1 MV/8000-specific I/O instructions

MNEM	Name	Action
DIA	Data in A	Transfers data from the A buffer of an I/O device to an accumulator.
DIB	Data in B	Transfers data from the B buffer of an I/O device to an accumulator.
DIC	Data in C	Transfers data from the C buffer of an I/O device to an accumulator.
DOA	Data out A	Transfers data from an accumulator to the A buffer of an I/O device.
DOB	Data out B	Transfers data from an accumulator to the B buffer of an I/O device.
DOC	Data out C	Transfers data from an accumulator to the C buffer of an I/O device.
HALT (DOC, CPU)	Halt	Stops the processor.
INTA (DIB,CPU)	Interrupt acknowledge	Returns the device code of an interrupting device.
INTDS (NIOC,CPU)	Interrupt disable	Sets the Interrupt On flag to 0.
INTEN (NIOS,CPU)	Interrupt enable	Sets the Interrupt On flag to 1.
IORST (DIC,CPU	Reset	Sets all Busy and Done flags and the priority mask to 0.
MSKO (DOB,CPU	Mask out	Changes the priority mask.
NIO	No I/O transfer	Changes a flag without causing any other effect.
READS (DIA,CPU)	Read switches	Places the contents of the console data switches into an accumulator.
SKP	I/O skip	Tests a flag and skips the next sequential word if the test condition is true.
SKP, CPU	CPU skip	Tests the Interrupt On or Power Fail flag and skips the next sequential word if the test condition is true.
VCT	Vector on interrupting device	Identifies highest priority interrupt and passes control through a table to a device handler.

Table 14.2 C/350 I/O instructions

Interrupts

When an interrupt occurs, the processor disables further interrupts by setting the ION flag to 0. The actions that follow depend on whether the ATU is enabled.

ATU Enabled/Disabled

The flow chart in Figure 14.1 summarizes the interrupt sequence.

Interrupt Sequence, ATU Disabled
When the ATU is not enabled, the processor checks if the C/350 MAP is enabled or disabled. If the MAP is enabled, the narrow interrupt handler services the interrupt. The processor treats page zero as a C/350 page zero. Note that MV/8000 programs are not executable when the C/350 MAP is enabled.

If the C/350 MAP is disabled, then the processor is operating in *physical mode*, and the interrupt occurred during a MV/8000 program. The processor fetches the address of the interrupt handler and prepares to resolve any indirection.

Interrupt Sequence, ATU Enabled
When the ATU is enabled, the processor fetches the contents of logical location 1 in page zero of Ring 0. This location contains the address of the interrupt handler. The processor next determines the current segment of execution. If it is not Segment 0, the processor performs a ring crossing to Segment 0 (see Figure 14.1). Next, the interrupt handler address must be resolved.

Address Resolution

If the fetched address of the interrupt handler is indirect, the processor resolves it to a final direct address. This address is used to reference the first instruction of the handler.

Handler Identification

The first instruction of the interrupt handler will be one of three types:

- An **XVCT** instruction,
- Any instruction whose bit 0 is 1 and bits 12–15 are 1001 (type 1),
- Any other instruction (type 2).

Type 1 instructions are MV/8000-specific instructions. Type 2 instructions are C/350 instructions, **WBR**, and some memory to accumulator instructions.

C/350 Interrupt

If the first instruction of the handler is a type 2 instruction, the processor stores the contents of the PC in location 0 in page zero of Segment 0. The PC contains the address of the next instruction in the program. After storing the PC, the processor jumps to the narrow handler.

Immediate Interrupt

If the first instruction of the handler is a type 1 instruction, then the processor executes an *immediate interrupt*. The processor loads the contents of the PC into logical locations 2 and 3 of Segment 0. The PC contains the address of the next instruction in the program. The processor then jumps to the interrupt handler.

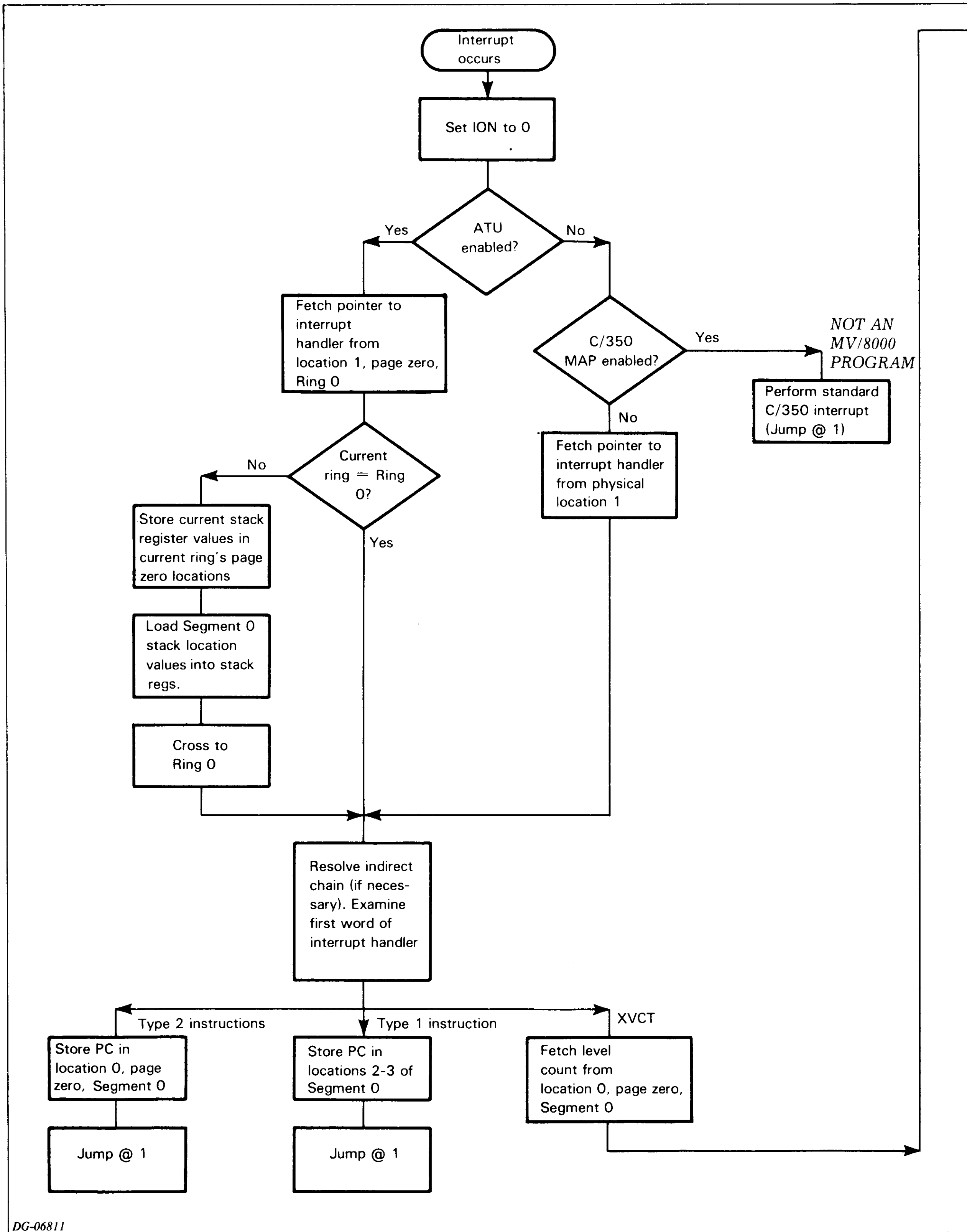
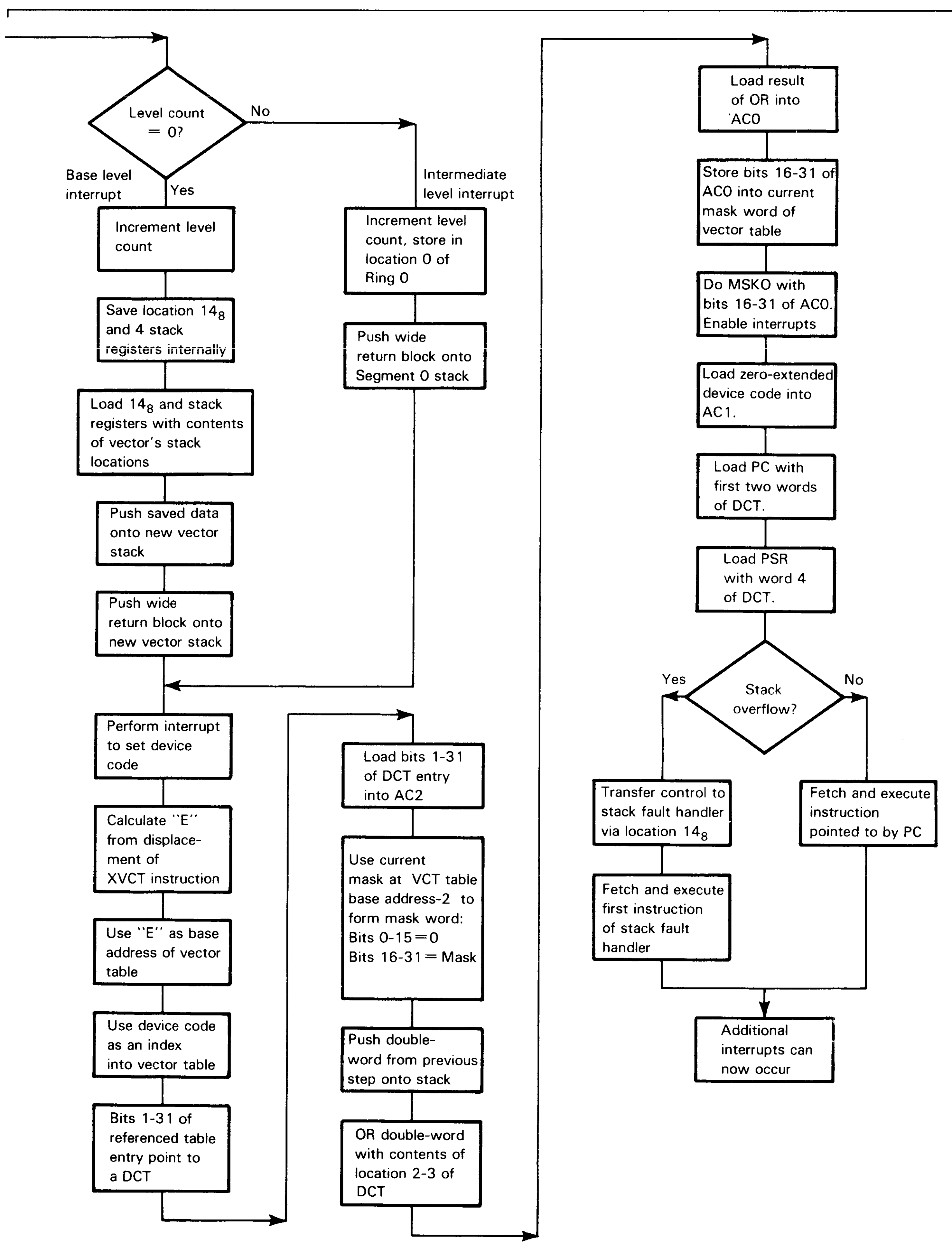


Figure 14.1



Vectored Interrupt

If the first instruction is an **XVCT** instruction, then the processor executes a *vectored interrupt*. The processor fetches the contents of location 0 of Segment 0. If the contents are equal to 0, then the processor will begin *base level* interrupt processing. If the contents are non-zero, then the processor will begin *intermediate level* processing. The

processor increments the contents by one, then stores them back into location 0 of Segment 0.

> NOTE: *Software, as part of its interrupt return program, decrements location 0 by 1. When the* XVCT *instruction is executed, locations 2 and 3 are not updated with the value of the program counter pointing to the next instruction.*

Base-Level Interrupt Processing

The initial actions of base-level processing depend on whether the current segment is 0. Later actions for Segment 0 and non Segment 0 processing are the same.

Segment 0, Initial Processing

When the current segment is Segment 0, the processor saves the contents of location 14_8 (address of the stack fault handler) and the four stack registers in internal processor state. Execution continues with the common sequence.

Outer Segment, Initial Processing

When the current segment is not Segment 0, the processor restores WSP and WFP to their locations in page zero of the current segment. (The register values of WSB and WSL are assumed to be the same as the page zero, current segment values of WSB and WSL.) Next, the processor performs a ring crossing to Segment 0. It then saves the contents of location 14_8 (address of the stack fault handler) and the four stack registers in internal processor state. Execution continues with the common sequence.

Common Sequence

The processor loads the five stack parameters (location 14_8 and the four stack registers) from the vector stack locations. The locations and their contents are shown in Table 14.3.

Ring 0 Location	Contents	Moved into
4	Vector stack pointer	WSP
4	Vector stack pointer	WFP
4	Vector stack pointer	WSB
6	Vector stack limit	WSL
7	Vector stack fault address	Location 14_8

Table 14.3 Vector stack locations and contents

> NOTE: *The processor interprets the contents of locations 4 and 6 as 16-bit word offsets. This means that the vector stack is initially limited to 128 Kbytes.*
>
> *The processor zero extends the contents of vector stack pointer and limit locations before loading them into the appropriate registers. This enables stack underflow and overflow.*

After loading the vector stack information, the processor pushes the old stack parameters that were stored in internal processor state onto the new vector stack. It next pushes a wide return block onto the vector stack. Execution continues with the final sequence.

Intermediate Level Interrupt Processing

Intermediate level processing occurs when the contents of location 0 in page zero of Segment 0 are nonzero. As in base-level processing, the initial action depend on whether the current segment is Segment 0 or not.

Segment 0 Processing
When the current segment is Segment 0, the processor pushes a wide return block onto the Segment 0 stack. Execution continues with the final sequence.

Outer Segment Processing
When the current segment is not Segment 0, the processor restores WSP and WFP to their locations in page zero of the current segment. The values of WSL and WSB in the register set and those in the current segment are identical. Next, the processor performs a ring crossing to Segment 0, where it loads the stack registers with the contents of the appropriate page zero locations. The processor pushes a wide return block and continues execution with the final sequence.

Final Sequence
All interrupts, whether base-level or intermediate, execute the same sequence of actions to conclude interrupt service. Figure 14.2 will help explain this sequence.

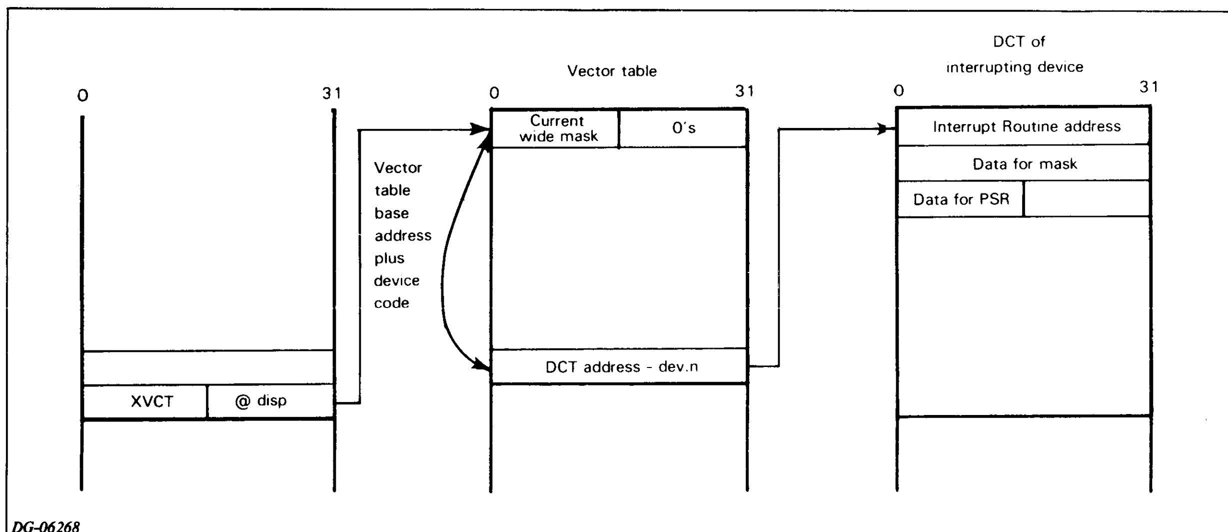


Figure 14.2

The processor calculates the effective address (*E*) from the displacement of the **XVCT** instruction. *E* addresses the base of a *vector table.* This table contains 64 double-word entries, one for each device. The device code of the interrupting device specifies a double-word offset from the base of the vector table. The processor adds this offset to the effective address *E* to produce the address of some entry in the vector table. Bits 1–31 of this vector table entry contain a pointer to the interrupting device's *device control table* (DCT). The processor loads this value into AC2.

The processor next pushes a double-word onto the current stack. Bits 0–15 of the double-word contain 0. Bits 16–31 contain the contents of the current mask (found in the word preceding the start of the vector table). The processor then loads AC0 with the logical OR of the double-word just pushed and the double-word contained in words 2–3 of the DCT.

After loading AC0, the processor loads the current mask in the vector table with bits 16-31 of AC0. Once the new mask is in place, the processor does a maskout from bits 16-31 of AC0, enables interrupts, and loads AC1 with the device code of the interrupting device. The device code is a 6-bit value, and the processor zero extends it to 32 bits before loading it into AC1.

The processor loads the PC with the contents of the first two words of the DCT. These two words contains the address of the device interrupt routine. Then, using the PSR value loaded into word 4 of the DCT, the processor initializes the *OVK, OVR* and *IRES* flags.

Once all registers are loaded, the processor checks for stack overflow. If overflow occurred, control transfers to the stack fault handler (the vector stack and vector stack fault handler are already initialized). Note that the first instruction of the stack fault handler will execute before any other interrupts will be honored. If no overflow occurred, execution continues with the next sequential instruction addressed by the PC.

NOTE: *The addresses contained in the vector table and the DCT are not restricted to Segment 0. References to other segments must conform, however, to ring crossing rules.*

Interrupting An Instruction

When an interrupt is honored, program execution stops. How the processor halts program execution to service the interrupt depends upon the instruction currently executing within the program. The currently executing instruction will be one of three types:

- A non-interruptable instruction,
- A restartable instruction,
- A resumable instruction.

Non-interruptable Instructions
If an instruction is non-interruptable, the processor finishes executing that instruction before it services the interrupt. Examples of non-interruptable instructions are *Add*, *Load Accumulator*, and *Complement*.

Note that the processor does not set bit 2 of the PSR to 1 if an interrupt occurs during a non-interruptable instruction.

Restartable Instructions
If an instruction is restartable, the processor services the interrupt before the instruction finishes. When an interrupt occurs, the processor saves the address of the interrupted instruction in the PC, and then services the interrupt. When servicing is complete, the processor can restart the interrupted instruction in one of two ways.

If the parameters of the restartable instruction are *unchanged*, then the processor restarts the instruction from the beginning. That is, if an interrupt occurs during a *Floating Point Divide* instruction, the processor would restart the instruction from the beginning because there has been no change in the accumulators containing the operands.

If, however, the parameters of the interrupted instruction have been *updated*, the processor restarts execution with the updated values. *Block Move* is an example of this type of instruction. This instruction uses pointers to source and destination locations and updates them after each one-word move. After servicing the interrupt, the processor restarts execution with the current values of the source and destination pointers, not the original values.

Note that the processor does not set bit 2 of the PSR if an interrupt occurs during a restartable instruction.

Resumable Instructions

As with restartable instructions, the processor services an interrupt before finishing a resumable instruction. The processor must save a copy of internal processor state, however, if it is to restart a resumable instruction correctly. The following paragraphs show what happens when an interrupt occurs during execution of a resumable instruction.

Before interrupting resumable instructions, ensure that:

- A stack has been defined;
- The interrupt handler uses **WPOPB, WRSTR, WRTN** or **LPSR** to return to the interrupted program. These instructions restore bit 2 of the PSR when interrupt service completes.

When an interrupt occurs, the processor pushes a copy of all necessary processor state information (the micro state block) onto the current stack. The information needed depends upon the interrupted instruction. The processor then sets bit 2 of the PSR to 1.

After pushing the block, the processor checks for stack overflow. If it detects a stack overflow, the processor services the fault after resuming the interrupted program. In other words, the processor services the interrupt before servicing the stack fault.

After servicing the interrupt, the processor restores bit 2 of the PSR using the appropriate return instruction, then tests bit 2. If bit 2 contains a 1, the processor examines the micro state block on the current wide stack to determine the type of microinterrupt.

If the micro state block is valid, the processor resumes executing the interrupted instruction. If the block is invalid, actions depend on the interrupted instruction:

- An MV/8000-specific instruction causes a a protection fault to occur. AC1 will contain the code 12 to indicate the invalid micro state block.
- A C/350 floating point instruction causes a floating point fault to occur. The processor sets bit 9 of the FPSR to 1 to indicate the invalid micro state block.
- A C/350 commercial instruction causes a narrow commercial fault to occur. AC1 will contain the code 5 to indicate the invalid micro state block.

NOTE: *When an interrupt occurs during a ring crossing, the saved PC points to the first instruction of the called procedure.*

Table 14.4 shows how the processor sets bit 2 of the PSR and bit 9 of the FPSR when an interrupt occurs during execution of a resumable instruction.

Instruction	PSR bit 2	FPSR bit 9
C/350	Unchanged	1
MV/8000-specific	Function of interrupted instruction	1

Table 14.4 State of PSR bit 2 and FPSR bit 9

Standard I/O Devices

The MV/8000 contains three standard I/O devices: the Programmable Interval Timer (PIT), the Real-Time Clock (RTC), and the Asynchronous Line Controller (ALC) .

Programmable Interval Timer

The programmable interval timer is a CPU-independent time base which can be programmed to initiate program interrupts at fixed intervals ranging from 100 microseconds to 6.5536 seconds in increments of 100 microseconds. It can also be sampled with I/O instructions at any point in its cycle to determine the time until the next interrupt. The PIT is used in multiprogram operating systems to allocate CPU time to different programs on a "time slice" basis.

The PIT consists of a 16-bit initial count register and a 16-bit counter. During operation, the PIT counter is loaded with the contents of the initial count register. It is then incremented at 100 microsecond intervals until the count reaches 177777_8. The PIT then initiates a program interrupt request. At the end of the next 100 microsecond interval, it is again loaded with the contents of the initial count register and the counting process is repeated. A Busy flag and a Done flag control the operation of the device.

Table 14.5 lists the instructions used to program the PIT.

MNEM	Name	Action
DOA PIT	Specify initial count	Selects the value which will be loaded into the counter each time the PIT is started or overflows.
DIA PIT	Read count	Reads the current value of the PIT counter.

Table 14.5 PIT instructions

Programming Notes

In order to obtain a particular time interval between program interrupt requests, load the two's complement of the number of 100 microsecond intervals between interrupt requests into the initial count register. When you first start the PIT, the interval to the first program interrupt request may be anywhere from 0 to 6.5536 seconds. After the first interrupt request, the time between program interrupt requests will be the value selected by the contents of the initial count register.

Real-Time Clock

The real-time clock generates low frequency I/O interrupts for performing time calculations independent of CPU timing. These interrupts may be used as a time base in programs which require it. The frequency of the clock is program selectable to a.c. line frequency, 10Hz, 100Hz, and 1000Hz. A Busy and a Done flag control the operation of the device.

One instruction programs the real time clock, as shown in Table 14.6.

MNEM	Name	Action
DOA RTC	Select RTC frequency	Selects the frequency of real time clock interrrupts.

Table 14.6 RTC instruction

Programming Notes

After first starting the real-time clock, the first program interrupt request can come at any time up to the selected clock period. After the first interrupt has occurred, succeeding interrupts come at the clock frequency, provided that the program always sets Busy to 1 before the clock period expires. After power up or **IORST**, the clock is set to the line frequency. After power up, the line frequency pulses are available immediately, but five seconds must elapse before a steady pulse train is available from the clock for other frequencies.

Asynchronous Line Controller

The Asynchronous Line Controller (ALC) is the communication link between the MV/8000 computer and the system's master terminal. It supports asynchronous communication at selected rates from 110 to 9600 baud in 7-bit codes with program generated parity, or 8-bit codes with no parity. One or two stop bits may be used with either format.

Because the asynchronous communications input and output can generate program interrupts independently, each has its own device code and is controlled by its own set of Busy and Done flags. The ALC is program compatible with Data General's Model 4010 controller.

A single instruction (shown in Table 14.7) programs the asynchronous line input (ALI).

MNEM	Name	Action
DIA ALI	Read input buffer	Reads a character from the input buffer.

Table 14.7 ALI instruction

A single instruction (shown in Table 14.8) programs the asynchronous line output (ALO).

MNEM	Name	Action
DOA ALO	Load output buffer	Places a character in the output buffer.

Table 14.8 ALO instruction

Programming Notes

The ALC is set up to transmit and receive 8-bit characters without parity checking. 7-bit characters can be sent or received with even, odd, or mark parity under program control by using the high order bit in the 8-bit character (bit 8 in the accumulator) as a parity bit. On transmission, the program which drives the asynchronous line controller calculates and inserts the correct parity bit. On reception, the program calculates and checks parity on the received character.

There are timing constraints on the *receive* portion of the controller. As each character is received, it is placed in an input character buffer, the Done flag is set to 1, and the Busy flag is set to 0. If the program controlling the receiver does not transfer the character before the next character is received, the contents of the input character buffer will be overwritten and the previous character will be lost. Typically, the inter-character time at

110 baud is 100 milliseconds and at 9600 baud the inter-character time is approximately 104 microseconds.

Chapter 15
The I/O Processor

The IOP is a 16-bit ECLIPSE processor that resides within the cabinet of the MV/8000. The IOP features standard facilities such as stack, standard I/O bus, C/350 instruction set with character instructions, priority interrupt system, etc. The IOP handles asynchronous communications support. It is not user programmable.

Forms of Host–IOP Communication

Communication between the MV/8000 processor and the IOP is necessary to coordinate their operation. For example, the IOP must be able to signal the host when it has completed a task or needs more information. The IOP MAP and two groups of special instructions provide the MV/8000 and the IOP with the necessary ability to communicate.

The MAP

The IOP hardware MAP allows the IOP to communicate directly with host memory. The MAP interprets memory references made by the IOP and determines whether the reference is to a local IOP memory location or a host memory location. When the reference is to host memory, the MAP sends the reference along the host data channel and through the host map to the correct memory location. This means that the IOP has direct access to information stored in the host.

Communication Instructions

The host uses more indirect means to manipulate the IOP. A group of special instructions allow the host to access IOP memory in much the same way as the control panel switches on a traditional machine (see Table 15.1). With these instructions the host can load the IOP map, check the map status, and manipulate the contents of various IOP registers. This allows the host to oversee the general operation of the IOP.

The IOP also has a special group of instructions that allow it to modify some aspects of the host's operation (see Table 15.2). While they do not give the IOP supervisory control over the host, they do allow the IOP to change the host data channel map and determine where information will be loaded into host memory.

Each of the host's and the IOP's special instructions contains an optional group of mnemonics that can be used to manipulate host and IOP flags. These flags are the Busy, Done, and Interrupt Request flags; they are used to indicate the state of the processors. By setting some of its flags, the host or the IOP can signal the continuation or completion of a task or request an interrupt.

Elements of the IOP

The IOP includes the following:

- A narrow stack
- Standard C/350 instruction set
- Data General's standard data channel for medium- to high-speed devices
- Programmed I/O with priority interrupt handling and vectoring capability
- Extended operation feature
- Arithmetic logic unit

IOP Memories

The IOP contains 64 Kbytes of local semiconductor memory. It is not expandable. The word length is 16 bits. The address range is from 0 to 77777_8.

MAP

The IOP MAP can map 2 Kbyte pages of IOP address space into either IOP local memory or host memory. To do this, the MAP contains two sets of pointers for IOP data channel and programmed memory references. Each set contains one pointer for each page. When either type of memory reference occurs, the five most significant bits of the logical address select a 1-bit pointer from the MAP. This pointer determines if the address is to local memory or host memory. In references to local memory, the unaltered logical address references some local memory location. In references to host memory, the address is sent across the host data channel to the host map. The host map interprets the physical location of the address sent by the MAP.

User and Data Channel Maps

The IOP has both a user map and data channel map supported on its own I/O bus. The IOP uses the IOP data channel map when transferring data to IOP I/O devices. The IOP uses the *host* data channel and *host* data channel maps when referencing host memory.

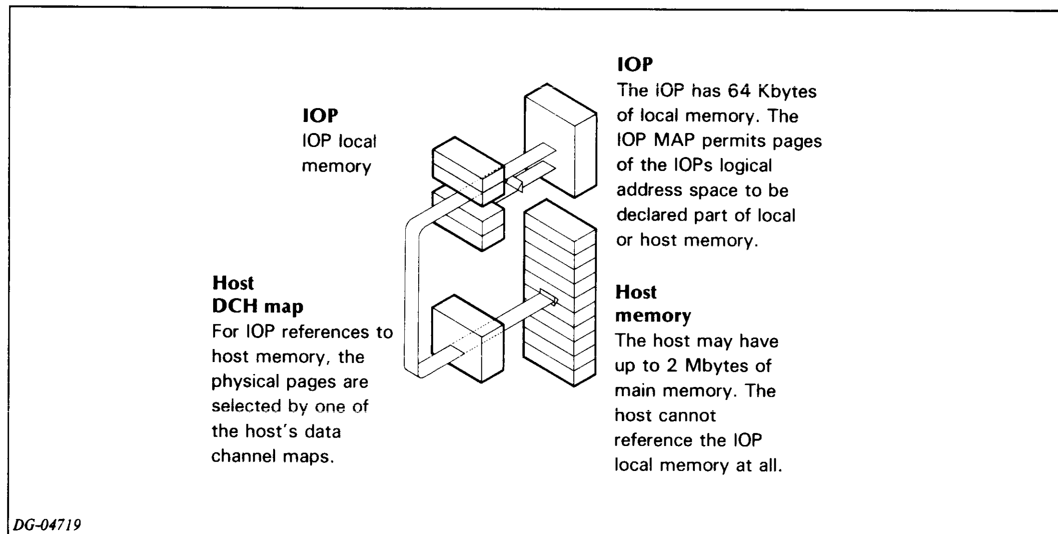


Figure 15.1

Parity Generator

The IOP parity generator provides a parity bit for every word written into IOP local memory. To enable parity checking for IOP local memory, use the *Read Map Status and Parity Control* instruction. This instruction sets the Parity Enable bit (bit 4) in the map status and parity control register. Detection of an error resets the IOP and its devices and sets the host interrupt request flag.

Host–IOP Interface

The host–IOP interface is a group of registers in the IOP that function as a communications link between the IOP and the host:

- The registers act as the console of the IOP. They are used to examine and modify the contents of IOP memory, step through the execution of a program, and other similar operations.
- The registers can be used by the host to control and monitor the IOP. The host does this by reading information from, or writing information to, these registers.

Interface Elements

A description of the interface's main elements and the host instructions that access them is given below. Unless otherwise specified, the registers are host-accessible via programmed I/O only.

Console Switch Register

This register takes the place of the data switches on a standard console. Load this register from the host using a *Load Console Switch Register* instruction.

Console Function Register

This register takes the place of the function switches on a standard console. Load this register using a *Load Console Function Register* instruction.

Address Register
This register holds the address of the location last referenced by the IOP. Use the *Read Address Register* to examine the contents of this register.

PC Save Register
Each time the IOP halts, the value of the PC is stored in this register. Use the *Read PC Save Register* instruction to examine the contents of this register.

Console Register
This register serves the same purpose as the data lights on a standard console. Use the *Read Console Buffer* instruction to examine the contents of this register.

Cross Interrupts

When the IOP needs information from the host, it signals the host for an interrupt by setting the host's *Interrupt Request* flag. The host checks this flag for interrupts at the end of each memory cycle. If an interrupt is pending, then the host halts its program execution long enough to service the interrupt and set its Interrupt Request flag to 0.

Similarly, when the host needs information from the IOP, it signals the IOP for an interrupt by setting the IOP's Interrupt Request flag.

Setting the Interrupt Request Flags

To set the Interrupt Request flags, use the appropriate flag control command appended to the mnemonic of one of the host I/O instructions or one of the IOP I/O instructions. The specific ways to set the Interrupt Request flags are described below.

The host can set the IOP's Interrupt Request flag by issuing an S flag control command to the IOP device code. This can be cleared by issuing a **C** flag control command from the IOP to device code 4.

The IOP can set the host's Interrupt Request flag by issuing an S flag control command to device code 4. This can be cleared by issuing a **C** flag control command from the host to the IOP device code. Note that a parity error in IOP local memory will also cause a host interrupt and IOP system reset. This interrupt can be cleared by issuing a **P** flag control command from the host to the IOP.

The methods described above will set other flags at the same time they set the Interrupt Request flags. These other flags are listed below, with tables of the flag control commands and their actions.

Busy and Done Flags
Both the host and the IOP have a Busy flag and a Done flag to indicate the state of the processor. The IOP's Busy and Done flags are in the host; the host's Busy and Done flags are in the IOP. Setting the IOP's Done flag to 1 causes an interrupt in the host. Setting the host's Busy flag to 1 will cause an interrupt in the IOP. The IOP's Busy flag is set to 1 whenever the IOP is running.

Parity Error Flag
The IOP parity generator sets this flag to 1 when it finds a parity error. If the IOP does not mask out parity errors, this flag generates a host interrupt when set to 1.

Host-IOP Communications Instructions

Table 15.1 lists the instructions used by the host to communicate with the IOP.

MNEM	Name	Action
DIA IOP	Read PC save register	Loads the contents of the IOP PC save register into an accumulator.
DIB IOP	Read console buffer	Loads the contents of the IOP console buffer into an accumulator.
DIC IOP	Read address buffer	Loads the contents of the IOP address buffer into an accumulator.
DOA IOP	Control console function register	Stores the contents of an accumulator into the IOP console function register.
DOB IOP	Control switch register	Stores the contents of an accumulator into the IOP switch register.

Table 15.1 Host communications instructions

The following flag control commands can be used with the host communications instructions:

f=S Sets the host Busy and the IOP Interrupt Request flags to 1

f=C Sets the IOP Done and host Interrupt Request flags to 0

f=P Sets the IOP Parity Error and host Interrupt Request flags to 0

IORST Sets the IOP Done flag, host Interrupt Request flag and host interrupt mask bit to 0; also resets the IOP processor and its I/O devices

NOTE: *The* C *flag control command does not clear a host interrupt caused by an IOP parity error. The* P *flag control command does not clear a host interrupt caused by the IOP Done flag.*

Table 15.2 lists the instructions used by the IOP to communicate with the host.

MNEM	Name	Action
DIA IOPI	Read map status	Loads the map status and parity control bits and MAP host/local flag into an accumulator.
DOA IOPI	Control map and parity	Stores the contents of an accumulator into the map status and parity control register.
LMP	Load map	Loads a number of map entries from a table in memory to the IOP MAP.

Table 15.2 IOP communications instructions

The following flag control commands can be used with the IOP instructions:

f=S Sets the IOP Done and host Interrupt Request flags to 1

f=C Sets the host Busy and IOP Interrupt Request flags to 0

f=P Sets the host Done flag to 0

IORST Sets bits 2–4, 14, and 15 of the map status and parity control register, host Done flag, IOP Interrupt Request flag, host Busy flag, and IOP interrupt mask bits to 0

Programming Examples

The following two programming examples illustrate host–IOP communications. The first example shows how to load data into the IOP. The second example is more complex; it shows how to start and load the IOP.

Example 1

The IOP instructions place data in IOP memory by using the IOP console registers. To place a word of data into an IOP local memory location, first load the IOP address receiving the data into the switch register. Then specify the Examine function by loading the function register with the numerical representation of Examine. Next, load the data value into the switch register, and load the numerical representation of the Deposit function into the function register.

The following program uses this method of loading data into the IOP. See the IOP Instruction Dictionary (Chapter 18) for a discussion of the host instructions plus a table of numerical representations of the functions.

```
                         .
                         .        ; Previous part of program.
                         .
                                  ; This part of the program loads two
                                  ; words of data from the host into IOP
                                  ; locations 100 and 101.
CO:        0                      ; Constant 0.
C100:      100                    ; This is the address of the IOP location
                                  ; that will receive the first data word.
EX:        050000                 ; Code for the Examine function.
DP:        040000                 ; Code for the Deposit function.
DN:        044000                 ; Code for the Deposit Next function.
DATA:      LDA       1,CO         ; These are the two words to be
           STA       1,STR        ; loaded into the IOP.
LSTART:    LDA       0,C100
           DOB       0,IOP        ; Put address of IOP location in data
                                  ; switches.
           LDA       1,EX
           DOA       1,IOP        ; Load console function register with code
                                  ; of Examine function.
           LDA       0,DATA
           DOB       0,IOP        ; Put first data word in data switches.
           LDA       1,DP
           DOA       1,IOP        ; Load console function register with
                                  ; code of Deposit function.
           LDA       0,DATA+1
           DOB       0,IOP        ; Put second data word into data switches.
           LDA       1,DN
           DOA       1,IOP        ; Load console function register with code
                                  ; of deposit next function and deposit
                                  ; next word into location 101.
           .
           .                      ; Rest of program.
```

Example 1

Example 2

This example shows how to start the IOP. The program below is put into host memory, starting at location 30. It loads a small bootstrap program and map data into IOP locations 0–53 via the IOP console registers. It then transfers control to bootstrap which remaps the IOP and, using the **BLM** instruction, transfers the contents of 8K of host memory locations starting at location 40000 to the first 8K of IOP memory.

```
                                        ;This program demonstrates the use of the
                                        host/IOP communications facilities. The host
                                        program places a 43-word bootstrap program in
                                        the IOP via the console functions and starts the
                                        IOP. Using its mapping capability, the IOP then
                                        moves 8K words into its local memory from host
                                        memory.
        .TITL       START
        .TXTM       1
                                        ; AC0 holds switch data for IOP
                                        ; AC1 holds function data for IOP
                                        ; AC2 holds array address for IOP boot
                                        ; AC3 unused
START:  INTDS
        SUB         0,0
        DOB         0,IOP               ; Deposit zero in data switches (this ; will be the
                                        starting location of the
                                        ; boot program in the IOP)
        LDA         1,EX
        DOA         1,IOP               ; Specify the examine function
        LDA         2,.A
        LDA         0,0,2               ; Address of IOP bootstrap in host
        DOB         0,IOP               ; deposit word 1 of boot in switches
        LDA         1,DP
        DOA         1,IOP               ; Specify deposit function
        INC         2,2
        LDA         1,DN                ; Load deposit next function in AC1 for loop
LOOP:   LDA         0,0,2
        DOB         0,IOP               ; Deposit next word of boot in switch
        DOA         1,IOP               ; Specify deposit next function
        INC         2,2
        DSZ         CNT                 ; Test for last word of IOP boot
        JMP         LOOP
        LDA         0,C40               ; Load the number of words loaded in IOP LMP
        DOB         0,IOP               ; Deposit number of words in switches
        LDA         1,DP1
        DOA         1,IOP               ; Specify deposit into AC1 function
        LDA         0,IMAP
        DOB         0,IOP               ; Load IOP map data address into switches
        LDA         1,DP2
        DOA         1,IOP               ; Specify deposit into AC2 function
```

Example 2

```
DONE:   SUB     0,0
        DOB     0,IOP      ; Deposit zero into switches (where to
                           ; start execution in IOP)
        LDA     1,STR
        DOA     1,IOP      ; Specify start function
 A:     A+1
CNT:    54                 ; Number of words in IOP boot
EX:     050000             ; Examine function
DP:     040000             ; Deposit function
DN:     044000             ; Deposit next function
STR:    060000             ; Start function
C40:    40                 ; Number of words in IOP LMP
DP1:    024000             ; Deposit in AC1 function
DP2:    030000             ; Deposit in AC2 function
IMAP:   10                 ; address of IOP MAP data
                           ; This bootstrap is loaded into the
                           ; IOP by the preceding host program
                           ; via the IOP console functions. The
                           ; IOP then remaps itself so that its
                           ; upper 16K address space is mapped into
                           ; the host. The IOP then moves 8K of
                           ; data from the host to its lower 8K,
                           ; overwriting the bootstrap.
A:      (A+1)
        DICC    0,CPU      ; Reset IOP
        LMP                ; Load the map
        LDA     0,MAPE
        DOA     0,IOPI     ; Load map status
        LDA     1,K8       ; Number of words to move in BLM
        LDA     2,ACS      ; Location in Host of data to be moved
        LDA     3,ACD      ; Location in IOP to put data
        BLM                ; Move the data
```

Example 2, continued

```
MDATA:   000000                 ; The IOP map data
         002000
         004000
         006000
         010000
         012000
         014000
         016000
         020000
         022000
         024000
         026000
         030000
         032000
         034000
         036000
         140000
         142000
         144000
         146000
         150000
         152000
         154000
         156000
         160000
         162000
         164000
         166000
         170000
         172000
         174000
         176000
ACD:     0                      ; Destination of IOP BLM.
ACS:     40000                  ; Source of IOP BLM.
K8:      17777                  ; Number of words moved by BLM.
MAPE:    100001
         .LOC     040000
                                ; This area of host memory contains the
                                ; 8K of data to be moved into the IOP.
                                ; After control is transferred to the
                                ; IOP the programs contained in this
                                ; data will begin execution.
         .END     START
```

Example 2, continued

Changing the Host Data Channel Map from the IOP

When the IOP references host memory, the references must travel across the host data channel and through the host data channel map to the appropriate location. Generally, the IOP uses the same host data channel map for all the references to host memory during a given program. The host data channel map, however, may change so that when the IOP makes its next reference the wrong map is loaded. This means that a new host

data channel map must be loaded before the reference from the IOP can be serviced. The MV/8000 IOP is able to change the host data channel map to the desired map whenever it references host memory.

To load a new host data channel map from the IOP, set the **DCH MAP LOAD** flag to one. This indicates that a host data channel map slot is to be loaded with a new value. Bit 7 of the IOP's map status and parity control register controls the value of this flag. The format of this register is shown below.

MS	LOGICAL PAGE	0	DML		DS	DL	UL	SEL	DCH	U
0	1–5	6	7	8	9	10	11	12–13	14	15

BITS	NAME	CONTENTS or FUNCTION
0	MS	Permit loading of map status (bits 12-15).
1-5	LOGICAL	Selects logical page.
6	PCL	Parity control load; must be 0.
7	DML	Set DCH MAP LOAD flag.
8	-	Reserved for future use.
9	DS	Suppress loading of DCH map select (bits 12,13).
10	DL	Suppress loading of DCH mode (bit 14).
11	UL	Suppress loading of USER mode (bit 15).
12,13	SEL	Selects host data channel map for references to host memory: 00 DCH A 01 DCH B 10 DCH C 11 DCH D
14	DCH	DCH mode on.
15	U	USER mode on.

When a **DOA***[f]* *ac*,**4** instruction is issued from the IOP, and bit 7 contains a 1, then the processor sets **DCH MAP LOAD** to 1. When this instruction is issued and bit 7 is 0, the processor sets **DCH MAP LOAD** to 0. Note that an IOP system reset or an IOP I/O reset will set **DCH MAP LOAD** to 0.

A page of the IOP's memory must also be mapped to the host. Bits 1-5 of the IOP's map status and parity control register specify the page to be mapped to the host.

Finally, the new information to be loaded into the host map slot should be specified. Since a map slot is more than 16 bits wide in the I/O system, two 16-bit words of information are needed to completely load a particular slot. The format of the two words is shown below.

V	Z	0	0	0	RESERVED	PHYSICAL PAGE NUMBER
0	1	2	3	4	5–17	18–31

where
V is the valid page bit;
Z is the transfer zeroes bit;
Physical page number specifies the address of a map slot.

The high order 7 bits of the address select a slot; the least significant bit selects the portion of the slot to be loaded.

The following example shows a typical way to load a host data channel map.

```
MAPO    =116415         ; New contents of map status and parity ; control register
                        bits:
                        ; Allow loading of map status - -
                        ; bit 0 = 1.
                        ; Pick logical page 7 - -
                        ; bits 1-5 = 00111.
                        ; Bit 6 = 0.
                        ; Set DCH MAP LOAD on - -
                        ; bit 7 = 1.
                        ; Bits 8-11 = 0.
                        ; Select host DCH D - -
                        ; bits 12-13 = 11.
                        ; DCH mode is off - - bit 14 = 0.
                        ; USER mode is on - - bit 15 = 1.
LEF     0,MAPO          ;
DOA     0,IOPI          ; Load new contents into map status and parity
                        ; control register.
LDA     0,MAP1          ; MAP1 and MAP2 contain the information to load
LDA     1,MAP2          ;into the map slot.
STA     0,1024*7        ; This loads map slot 7 in the host with the
                        ;contents of AC0.
STA     1,(1024*7)+1    ; This loads the rest of map slot 7 in the
                        ;host with the contents of AC1.
```

Loading a host data channel map

In the example, first load the map status and parity control register with the required contents. Then, set the **DCH MAP LOAD** flag, specify a page to map to the host, and set user mode to 1. The desired information should then be loaded into the memory locations MAP1 and MAP2. The information in these locations must have the format shown above.

Once all the preliminary information has been set up, the host data channel map can be loaded. When a write memory reference to the page mapped into the host is made, the processor goes through the host data channel to the host data channel map. There the processor loads the information in the specified accumulators (AC0 and AC1) into the slot at the specified address (1024*7 and (1024*7)+1). This writes the new map information into the specified slot.

One of the advantages to using this method of loading the host data channel map is that the process is transparent to the host. This means that programs executing during an IOP reference to host memory will not be interrupted when the map load occurs. Another advantage is that the IOP has control of the host data channal map it requires.

Chapter 16

The MV/8000 Instruction Dictionary

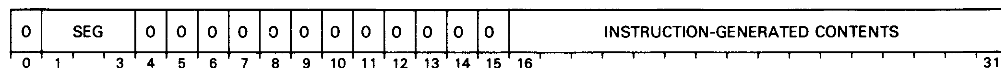
General Programming Notes

The instruction entries that follow contain references to a value called *overflow*. This value has meaning only while an instruction is executing. *Overflow* indicates if the currently executing instruction has resulted in an overflow condition. The processor inclusively ORs *overflow* with the current value of the **OVR** flag to determine the new value of **OVR**.

When a C/350 instruction is executed when the C/350 MAP is enabled, the processor uses only bits 16–31 in the specified operation (with exceptions, see below). If the result of the operation is stored in an accumulator, then bits 16–31 of the accumulator contain the result. Bits 0– 15 of the accumulator are indeterminate.

If the processor tries to execute any MV/8000 instruction when the C/350 MAP is enabled, an I/O Protection violation occurs. When this happens, the processor sets bit 2 of the map status register to 1 and pushes a narrow return block onto the narrow stack. The PC in the narrow return block references the instruction that caused the fault.

All C/350 program flow instructions and those that load effective addresses (such as **JMP, RTN,** and **ELEF**) alter the PC or a specified accumulator. Those that alter the PC leave PC bits 1–16 unchanged. Those that alter an accumulator change only bits 16–31; Bits 0–15 will always have the format shown in the figure below:



where *SEG* contains the number of the current segment.

C/350 ALC instructions specifying the no-load option (bit 12 is 1) do not alter the accumulators.

All C/350 instructions specifying an accumulator as a source of information only (such as **CLM**) leave the specified accumulator unchanged.

If **LMP** or **SYC**, or any C/350 MAP instructions execute when the ATU is enabled, a protection fault occurs. AC1 contains the code 9.

If any instruction specifies a word address, the processor ignores bit 0 of the word address.

An instruction that specifies PC- or AC-relative addressing modes produces a 31-bit address. This means that bit 0 of the indexed register does not alter the logical address produced by the instruction.

Instruction

The following is an alphabetical (by mnemonic) listing of all the instructions supported by the ECLIPSE MV/8000 processor.

The two indexes at the back of this manual enable particular instruction definitions to be found quickly: if the mnemonic is known; and if the instruction name is known.

Add Complement

ADD*[c][sh][#]* *acs,acd[,skip]*

1	ACS	ACD	1	0	0	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 15

Adds the logical complement of an unsigned integer to another unsigned integer.

Initializes carry to the specified value, adds the logical complement of the unsigned, 16-bit number in bits 16-31 of ACS to the unsigned, 16-bit number in bits 16-31 of ACD, and places the result in the shifter. The instruction then performs the specified shift operation, and loads the result of the shift into bits 16-31 of ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped. For this instruction, *overflow* is 0.

If the load option is specified, bits 0-15 of ACD are undefined.

NOTE: *If the sum of the two numbers being added is greater than 65,535 the instruction complements carry.*

Add

ADD*[c][sh][#]* *acs,acd[,skip]*

1	ACS	ACD	1	1	0	SH	C	#	SKIP
0	1 2	3 4	5	6	7	8 9	10 11	12	13 15

Performs unsigned integer addition and complements carry if appropriate.

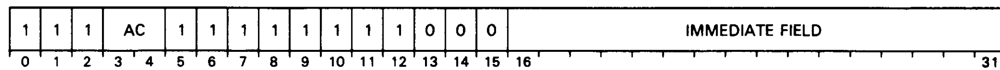
Initializes carry to the specified value, adds the unsigned, 16-bit number in bits 16-31 of ACS to the unsigned, 16-bit number in bits 16-31 of ACD, and places the result in the shifter. The instruction then performs the specified shift operation and places the result of the shift in bits 16-31 of ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped. For this instruction, *overflow* is 0.

If the load option is specified, bits 0-15 of ACD are undefined.

> **NOTE:** *If the sum of the two numbers being added is greater than 65,535, the instruction complements carry.*

Extended Add Immediate

ADDI *i,ac*



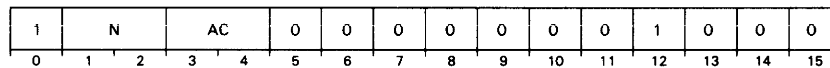
Adds a signed integer in the range -32,768 to +32,767 to the contents of an accumulator.

Treats the contents of the immediate field as a signed, 16-bit, two's complement number and adds it to the signed, 16-bit, two's complement number contained in bits 16-31 of the specified accumulator, placing the result in bits 16-31 of the same accumulator. Carry remains unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Add Immediate

ADI *n,ac*



Adds an unsigned integer in the range 1–4 to the contents of an accumulator.

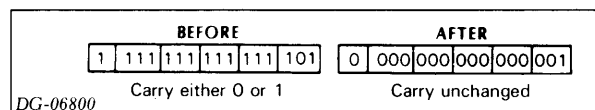
Adds the contents of the immediate field *N*, plus 1, to the unsigned, 16-bit number contained in bits 16-31 of the specified accumulator, placing the result in bits 16-31 of the same accumulator. Carry remains unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

> **NOTE:** *The assembler takes the coded value of* n *and subtracts 1 from it before placing it in the immediate field. Therefore, the programmer should code the exact value that he wishes to add.*

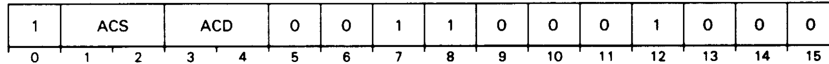
Example

Assume that AC2 contains 177775_8. After the instruction **ADI 4,2** is executed, AC2 contains 000001_8 and carry is unchanged.



AND With Complemented Source

ANC *acs,acd*

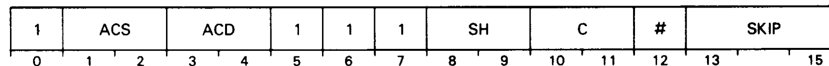


Forms the logical AND of the logical complement of the contents of bits 16-31 of ACS and the contents of bits 16-31 of ACD and places the result in bits 16-31 of ACD. The instruction sets a bit position in the result to 1 if the corresponding bit position in ACS contains 0. The contents of carry and ACS remain unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

AND

AND*[c][sh][#]* *acs,acd[,skip]*



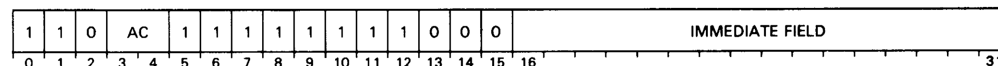
Forms the logical AND of the contents of two accumulators.

Initializes the carry bit to the specified value. Places the logical AND of bits 16-31 of ACS and bits 16-31 of ACD in the shifter. Each bit placed in the shifter is 1 only if the corresponding bit in both ACS and ACD is one; otherwise the resulting bit is 0. The instruction then performs the specified shift operation and places the result in bits 16-31 of ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped. *Overflow* is 0.

If the load option is specified, bits 0-15 of ACD are undefined.

AND Immediate

ANDI *i,ac*

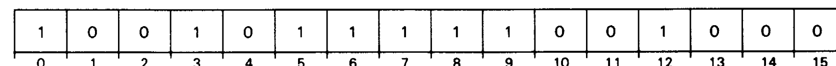


Places the logical AND of the contents of the immediate field and the contents of bits 16-31 of the specified accumulator in bits 16-31 of the specified accumulator. Carry is unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Block Add and Move

BAM



Moves memory words from one location to another, adding a constant to each one.

Moves words sequentially from one memory location to another, treating them as unsigned, 16-bit integers. After fetching a word from the source location, the instruction adds the unsigned, 16-bit integer in bits 16-31 of AC0 to it. If the addition produces a carry of 1 out of the high-order bit, no indication is given.

Bits 17–31 of AC2 contain the address of the source location. Bits 17–31 of AC3 contain the address of the destination location. The address in bits 17–31 of AC2 or AC3 is an indirect address if bit 16 of that accumulator is 1. In that case, the instruction follows the indirection chain before placing the resultant effective address in the accumulator.

The unsigned, 16-bit number in bits 16-31 of AC1 is equal to the number of words moved. This number must be greater than 0 and less than or equal to 32,768. If the number in AC1 is outside these bounds, no data is moved and the contents of the accumulators remain unchanged.

AC	Contents
0	Addend
1	Number of words to be moved
2	Source address
3	Destination address

For each word moved, the count in AC1 is decremented by one and the source and destination addresses in AC2 and AC3 are incremented by one. Upon completion of the instruction, AC1 contains zeroes, and AC2 and AC3 point to the word following the last word in their respective fields. The contents of carry and AC0 remain unchanged. *Overflow* is 0.

The 32-bit effectibe address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Words are moved in consecutive, ascending order according to their addresses. The next address after 77777_8 is 0 for both fields. The fields may overlap in any way.

> NOTE: *Because of the potentially long time that may be required to perform this instruction it is interruptable. If a* Block Add and Move *instruction is interrupted, the program counter is decremented by one before it is placed in location 0 so that it points to the interrupted instruction. Because the addresses and the word count are updated after every word stored, any interrupt service routine that returns control to the interrupted program via the address stored in memory location 0 will correctly restart the* Block Add and Move *instruction.*

When updating the source and destination addresses, the *Block Add And Move* instruction forces bit 0 of the result to 0. This ensures that upon return from an interrupt, the *Block Add And Move* instruction will not try to resolve an indirect address in either AC2 or AC3.

Breakpoint

BKPT

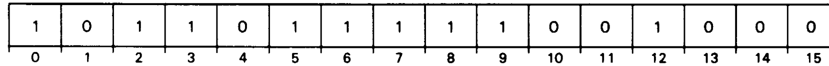
1	1	0	0	0	1	1	1	1	0	0	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pushes a wide return block onto the present stack.

The value of the PC in the return block is the address of this instruction. After pushing the block, the instruction checks for stack overflow. If no overflow occurred, the instruction sets the PSR to zero and performs a wide jump indirect through locations $10–11_8$ in page zero of the current segment. If overflow occurred, a stack fault occurs and AC1 contains the code 0; after the fault is handled, the PSR is set to zero and the jump indirect occurs. Carry remains unchanged by this instruction.

Block Move

BLM



Moves memory words from one location to another.

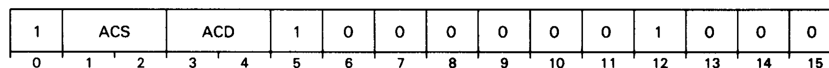
The *Block Move* instruction is the same as the *Block Add And Move* instruction in all respects except that no addition is performed and AC0 is not used. Carry remains unchanged and *overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

> **NOTE:** *The* Block Move *instruction is interruptable in the same manner as the* Block Add And Move *instruction.*

Set Bit To One

BTO *acs,acd*



Sets the specified bit to 1.

Forms a 32-bit bit pointer from the contents of bits 16-31 of both ACS and ACD. Bits 16-31 of ACS contains the high-order 16 bits and bits 16-31 of ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16-bits of the bit pointer and assumes the high-order 16 bits are 0. Carry remains unchanged and *overflow* is 0.

The instruction then sets the addressed bit in memory to 1, leaving the contents of ACS and ACD unchanged.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

> **NOTE:** *The bit pointer contained in ACS and ACD must not make indirect memory references.*

Set Bit To Zero

BTZ *acs,acd*

1	ACS		ACD		1	0	0	0	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sets the addressed bit to 0.

Forms a 32-bit bit pointer from the contents of bits 16-31 of both ACS and ACD. Bits 16-31 of ACS contains the high-order 16 bits and bits 16-31 of ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16 bits of the bit pointer and assumes the high-order 16 bits are 0. Carry remains unchanged and *overflow* is 0.

The instruction then sets the addressed bit in memory to 0, leaving the contents of ACS and ACD unchanged.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

NOTE: *The bit pointer contained in ACS and ACD must not make indirect memory references.*

Compare To Limits

CLM *acs,acd*

1	ACS		ACD		1	0	0	1	1	1	1	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Compares a signed integer with two other integers and skips if the first integer is between the other two. The accumulators determine the location of the three integers.

Compares the 16-bit, signed, two's complement integer in bits 16-31 of ACS to two 16-bit, signed, two's complement limit values, *L* and *H*. If the number in bits 16-31 of ACS is greater than or equal to *L* and less than or equal to *H*, the next sequential word is skipped. If the number in bits 16-31 of ACS is less than *L* or greater than *H*, the next sequential word is executed.

If ACS and ACD are specified as different accumulators, the address of the limit value *L* is contained in bits 16–31 of ACD. The limit value *H* is contained in the word following *L*. Bits 0-15 of ACD are ignored.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

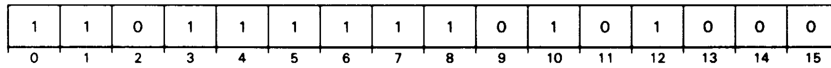
If ACS and ACD are specified as the same accumulator, then the integer to be compared must be in that accumulator and the limit values *L* and *H* must be in the two words following the instruction. *L* is the first word and *H* is the second word. The next sequential word is the third word following the instruction.

When *L* and *H* are in line, this instruction can be placed anywhere in the 32-bit address space.

This instruction leaves carry unchanged; *overflow* is 0.

Character Compare

CMP



Under control of the four accumulators, compares two strings of bytes and returns a code in AC1 reflecting the results of the comparison.

The instruction compares the strings one byte at a time. Each byte is treated as an unsigned 8-bit binary quantity in the range $0\text{–}255_{10}$. If two bytes are not equal, the string whose byte has the smaller numerical value is, by definition, the *lower valued* string. Both strings remain unchanged. The four accumulators contain parameters passed to the instruction. Two accumulators specify the starting address, the number of bytes, and the direction of processing (ascending or descending addressed) for each string.

Bits 16-31 of AC0 specify the length and direction of comparison for string 2. If the string is to be compared from its lowest memory location to the highest, bits 16-31 of AC0 contain the unsigned value of the number of bytes in string 2. If the string is to be compared from its highest memory location to the lowest, bits 16-31 of AC0 contain the two's complement of the number of bytes in string 2.

Bits 16-31 of AC1 specify the length and direction of comparison for string 1. If the string is to be compared from its lowest memory location to the highest, bits 16-31 of AC0 contain the unsigned value of the number of bytes in string 1. If the string is to be compared from its highest memory location to the lowest, bits 16-31 of AC1 contain the two's complement of the number of bytes in string 1.

Bits 16-31 of AC2 contain a byte pointer to the first byte compared in string 2. When the string is compared in ascending order, AC2 points to the lowest byte. When the string is compared in descending order, AC2 points to the highest byte.

Bits 16-31 of AC3 contain a byte pointer to the first byte compared in string 1. When the string is compared in ascending order, AC3 points to the lowest byte. When the string is compared in descending order, AC3 points to the highest byte.

Code	Comparison Result
- 1	string 1 < string 2
0	string 1 = string 2
+ 1	string 1 > string 2

The strings may overlap in any way. Overlap will not effect the results of the comparison.

Upon completion, bits 16-31 of AC0 contain the number of bytes left to compare in string 2. AC1 contains the return code as shown in the table above. Bits 16-31 of AC2 contains a byte pointer either to the failing byte in string 2 (if an inequality were found), or to the byte following string 2 (if string 2 were exhausted). Bits 16-31 of AC3 contains a byte pointer either to the failing byte in string 1 (if an inequality were found), or to the byte following string 1 (if string 1 were exhausted). Carry remains unchanged. *Overflow*

is 0.

If AC0 and AC1 both contain 0 (both string 1 and string 2 have length zero), the instruction compares no bytes and returns 0 in AC1. If the two strings are of unequal length, the instruction pads the shorter string with space characters $<040_8>$ and continues the comparison.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

> **NOTE:** *The original contents of AC2 and AC3 must be valid byte pointers to an area in the user's address space. If the pointers are invalid a protection fault occurs, even if no bytes are to be compared. AC1 contains the code 2.*

Character Move Until True
CMT

1	1	1	0	1	1	1	1	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Under control of the four accumulators, moves a string of bytes from one area of memory to another until either a table-specified delimiter character is moved or the source string is exhausted.

The instruction copies the string one byte at a time. Before it moves a byte, the instruction uses that byte's value to determine if it is a delimiter. It treats the byte as an unsigned 8-bit binary integer (in the range $0–255_{10}$) and uses it as a bit index into a 256-bit delimiter table. If the indexed bit in the delimiter table is zero, the byte pending is not a delimiter, and the instruction copies it from the source string to the destination string. If the indexed bit in the delimiter table is 1, the byte pending is a delimiter; the instruction does not copy it, and the instruction terminates.

The instruction processes both strings in the same direction, either from lowest memory locations to highest (*ascending order*), or from highest memory locations to lowest (*descending order*). Processing continues until there is a delimiter or the source string is exhausted. The four accumulators contain parameters passed to the instruction.

Bits 16-31 of AC0 contain the address (word address), possibly indirect, of the start of the 256-bit (16-word) delimiter table.

Bits 16-31 of AC1 specify the length of the strings and the direction of processing. If the source string is to be moved to the destination field in ascending order, bits 16-31 of AC1 contain the unsigned value of the number of bytes in the source string. If the source string is to be moved to the destination field in descending order, bits 16-31 of AC1 contain the two's complement of the number of bytes in the source string.

Bits 16-31 of AC2 contain a byte pointer to the first byte to be written in the destination field. When the process is performed in ascending order, bits 16-31 of AC2 point to the lowest byte in the destination field. When the process is performed in descending order, bits 16-31 of AC2 point to the highest byte in the destination field.

Bits 16-31 of AC3 contain a byte pointer to the first byte to be processed in the source string. When the process is performed in ascending order, bits 16-31 of AC3 point to the lowest byte in the source string. When the process is performed in descending order, bits 16-31 of AC3 point to the highest byte in the source string.

The fields may overlap in any way. However, the instruction moves bytes one at a time, so certain types of overlap may produce unusual side effects.

Upon completion, bits 16-31 of AC0 contain the resolved address of the translation table and AC1 contain the number of bytes that were not moved. Bits 16-31 of AC2 contain a byte pointer to the byte following the last byte written in the destination field. Bits 16-31 of AC3 contain a byte pointer either to the delimiter or to the first byte following the source string. Carry remains unchanged. *Overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kword of the current segment.

NOTES: *If AC1 contains the number 0 at the beginning of this instruction, no bytes are fetched and none are stored. The instruction becomes a No-Op.*

The original contents of AC0, AC2, and AC3 must be valid pointers to some area in the user's address space. If they are invalid a protection fault occurs, even if no bytes are to be moved. AC1 contains the code 2.

Character Move
CMV

1	1	0	1	0	1	1	1	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Under control of the four accumulators, moves a string of bytes from one area of memory to another and returns a value in carry reflecting the relative lengths of source and destination strings.

The instruction copies the source string to the destination field, one byte at a time. The four accumulators contain parameters passed to the instruction. Two accumulators specify the starting address, number of bytes to be copied, and the direction of processing (ascending or descending addresses) for each field.

Bits 16-31 of AC0 specify the length and direction of processing for the destination field. If the field is to be processed from its lowest memory location to the highest, bits 16-31 of AC0 contain the unsigned value of the number of bytes in the destination field. If the field is to be processed from its highest memory location to the lowest, bits 16-31 of AC0 contain the two's complement of the number of bytes in the destination field.

Bits 16-31 of AC1 specify the length and direction of processing for the source string. If the string is to be processed from its lowest memory location to the highest, bits 16-31 of AC1 contain the unsigned value of the number of bytes in the source string. If the field is to be processed from its highest memory location to the lowest, bits 16-31 of AC1 contain the two's complement of the number of bytes in the source string.

Bits 16-31 of AC2 contain a byte pointer to the first byte to be written in the destination field. When the field is written in ascending order, bits 16-31 of AC2 point to the lowest byte. When the field is written in descending order, bits 16-31 of AC2 point to the highest byte.

Bits 16-31 of AC3 contain a byte pointer to the first byte copied in the source string. When the field is copied in ascending order, bits 16-31 of AC3 point to the lowest byte. When the field is copied in descending order, bits 16-31 of AC3 point to the highest byte.

The fields may overlap in any way. However, the instruction moves bytes one at a time, so certain types of overlap may produce unusual side effects.

Upon completion, AC0 contains 0 and bits 16-31 of AC1 contain the number of bytes left to fetch from the source field. Bits 16-31 of AC2 contain a byte pointer to the byte following the destination field; bits 16-31 of AC3 contain a byte pointer to the byte following the last byte fetched from the source field. *Overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64Kbyte of the current segment.

NOTES: *If AC0 contains the number 0 at the beginning of this instruction, no bytes are fetched and none are stored. If AC1 is 0 at the beginning of this instruction, the destination field is filled with space characters.*

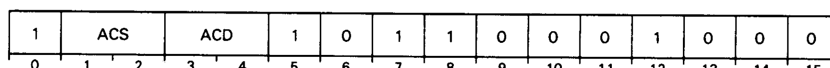
The original contents of AC2 and AC3 must be valid pointers to some area in the user's address space. If they are invalid a protection fault occurs, even if no bytes are to be moved. AC1 contains the code 2.

If the source field is longer than the destination field, the instruction terminates when the destination field is filled and sets carry to 1. In any other case, the instruction sets carry to 0.

If the source field is shorter than the destination field, the instruction pads the destination field with space characters <040_8>.

Count Bits

COB *acs,acd*



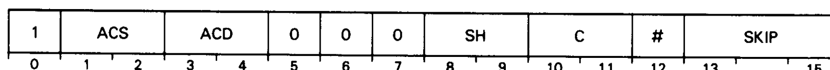
Adds a number equal to the number of ones in bits 16-31 of ACS to the signed, 16-bit, two's complement number in bits 16-31 of ACD. The instruction leaves the contents of ACS and the state of carry unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *If ACS and ACD are the same accumulator, the instruction functions as described above, except the contents of ACS will be changed.*

Complement

COM*[c][sh][#]* *acs,acd[,skip]*



Forms the logical complement of the contents of an accumulator.

Initializes carry to the specified value, forms the logical complement of the number in bits 16-31 of ACS, and performs the specified shift operation. The instruction then places the result in bits 16-31 of ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

If the load option is specified, bits 0-15 of ACD are undefined.

For this instruction, *overflow* is 0.

Complement Carry

CRYTC

1	0	1	0	0	1	1	1	1	1	1	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Complements the value of carry. *Overflow* is 0.

Set Carry to One

CRYTO

1	0	1	0	0	1	1	1	1	1	0	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Unconditionally sets the value of carry to 1. *Overflow* is 0.

Set Carry to Zero

CRYTZ

1	0	1	0	0	1	1	1	1	1	0	1	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Unconditionally sets the value of carry to 0. *Overflow* is 0.

Character Translate

CTR

1	1	1	0	0	1	1	1	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Under control of the four accumulators, translates a string of bytes from one data representation to another and either moves it to another area of memory or compares it to a second translated string.

The instruction operates in two modes; translate and move, and translate and compare.

When operating in translate and move mode, the instruction.translates each byte in string 1, and places it in a corresponding position in string 2. Translation is performed by using each byte as an 8-bit index into a 256-byte translation table. The byte addressed by the index then becomes the translated value.

When operating in translate and compare mode, the instruction translates each byte in string 1 and string 2 as described above, and compares the translated values. Each translated byte is treated as an unsigned 8-bit binary quantity in the range $0\text{-}255_{10}$. If

two translated bytes are not equal, the string whose byte has the smaller numerical value is, by definition the *lower valued* string. Both strings remain unchanged.

Bits 16-31 of AC0 specify the address, either direct or indirect, of a word which contains a byte pointer to the first byte in the 256-byte translation table.

Bits 16-31 of AC1 specify the length of the two strings and the mode of processing. If string 1 is to be processed in translate and move mode, bits 16-31 of AC1 contain the two's complement of the number of bytes in the strings. If the strings are to be processed in translate and compare mode, bits 16-31 of AC1 contain the unsigned value of the number of bytes in the strings. Both strings are processed from lowest memory address to highest.

Bits 16-31 of AC2 contain a byte pointer to the first byte in string 2.

Bits 16-31 of AC3 contain a byte pointer to the first byte in string 1.

Upon completion of a translate and move operation, bits 16-31 of AC0 contain the address of the word which contains the byte pointer to the translation table and AC1 contains 0. Bits 16-31 of AC2 contain a byte pointer to the byte following string 2 and bits 16-31 of AC3 contain a byte pointer to the byte following string 1. Carry remains unchanged. *Overflow* is 0.

Upon completion of a translate and compare operation, bits 16-31 of AC0 contain the address of the word which contains the byte pointer to the translation table. AC1 contains a return code as calculated in the table below. Bits 16-31 of AC2 contain a byte pointer to either the failing byte in string 2 (if an inequality was found) or the byte following string 2 if the strings were identical. Bits 16-31 of AC3 contain a byte pointer to either the failing byte in string 1 (if an inequality was found) or the byte following string 1 if the strings were identical. Carry contains an indeterminate value. *Overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

Code	Result
-1	Translated value of string 1 < Translated value of string 2
0	Translated value of string 1 = Translated value of string 2
+1	Translated value of string 1 > Translated value of string 2

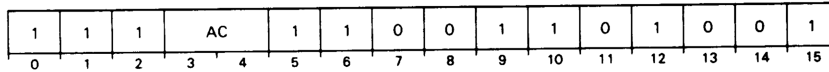
If the length of both string 1 and string 2 is zero, the compare option returns a 0 in AC1.

The fields may overlap in any way. However, processing is done one character at a time, so unusual side effects may be produced by certain types of overlap.

NOTE: *The original contents of AC0, AC2, and AC3 must be valid byte pointers to some area in the user's address space. If they are invalid a protection fault occurs, even if no bytes are to be moved or compared. AC1 contains the code 2.*

Convert to 16-Bit Integer

CVWN *ac*

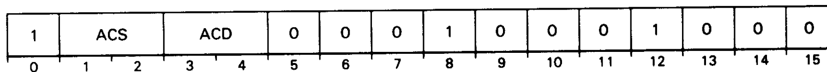


Converts a 32-bit integer to a 16-bit integer.

The instruction converts the 32-bit contents of the specified accumulator to a 16-bit integer by extending bit 17 into bits 0–16. If the 17 most significant bits do not contain the same value (i.e., all 1's or all 0's) before conversion takes place, then this instruction sets *overflow* to 1 before performing the conversion. Carry is unchanged.

Decimal Add

DAD *acs,acd*



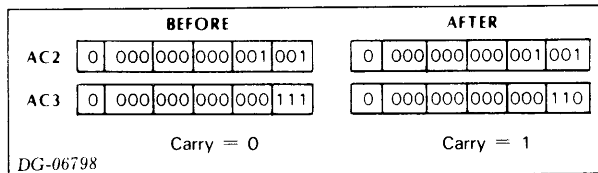
Performs decimal addition on 4-bit binary coded decimal (BCD) numbers and uses carry for a decimal carry.

Adds the unsigned decimal digit contained in bits 28–31 of ACS to the unsigned decimal digit contained in bits 28–31 of ACD. Carry is added to this result. The instruction then places the decimal units' position of the final result in bits 28–31 of ACD, and the decimal carry in carry. The contents of ACS and bits 0–27 of ACD remain unchanged. *Overflow* is 0.

> **NOTE:** *No validation of the input digits is performed. Therefore, if bits 28–31 of either ACS or ACD contain a number greater than 9, the results will be unpredictable.*

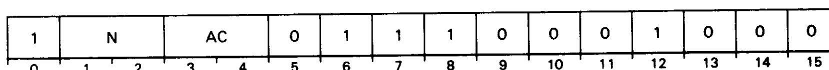
Example

Assume that bits 28–31 of AC2 contain 9; bits 28–31 of AC3 contain 7; and the carry bit is 0. After the instruction **DAD 2,3** is executed, AC2 remains the same; bits 28–31 of AC3 contain 6; and carry is 1, indicating a decimal carry from this *Decimal Add*.



Double Hex Shift Left

DHXL *n,ac*



Shifts the 32-bit number contained in bits 16-31 of AC and bits 16-31 of AC+1 left a

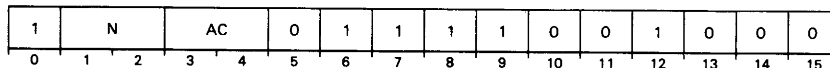
number of hex digits depending upon the immediate field *N*. The number of digits shifted is equal to *N*+1. Bits shifted out are lost and the vacated bit positions are filled with zeroes. Carry remains unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

> **NOTES:** *If AC is specified as AC3, then AC+1 is AC0.*
>
> *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits that he wishes to shift.*
>
> *If* N *is equal to 3, the contents of AC+1 are placed in AC and AC+1 is filled with zeroes.*

Double Hex Shift Right

DHXR *n,ac*



1	N		AC		0	1	1	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

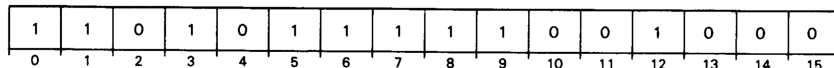
Shifts the 32-bit number contained in bits 16-31 of AC and bits 16-31 of AC+1 right a number of hex digits depending upon the immediate field *N*. The number of digits shifted is equal to *N*+1. Bits shifted out are lost and the vacated bit positions are filled with zeroes. Carry remains unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

> **NOTES:** *If AC is specified as AC3, then AC+1 is AC0.*
>
> *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits that he wishes to shift.*
>
> *If* N *is equal to 3, the contents of AC are placed in AC+1 and AC is filled with zeroes.*

Unsigned Divide

DIV



1	1	0	1	0	1	1	1	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Divides the unsigned 32-bit integer in bits 16-31 of two accumulators by the unsigned contents of a third accumulator. The quotient and remainder each occupy one accumulator.

Divides the unsigned 32-bit number contained in bits 16-31 of AC0 and bits 16-31 of AC1 by the unsigned, 16-bit number in bits 16-31 of AC2. The quotient and remainder are unsigned, 16-bit numbers and are placed in bits 16-31 of AC1 and AC0, respectively. Carry is set to 0. The contents of AC2 remain unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

> **NOTE:** *Before the divide operation takes place, the number in bits 16-31 of AC0 is compared to the number in bits 16-31 of AC2. If the contents of bits 16-31 of AC0 are greater than or equal to the contents of bits 16-31 of AC2, an overflow condition is indicated. Carry is set to*

1, and the operation is terminated. All operands remain unchanged.

Signed Divide
DIVS

1	1	0	1	1	1	1	1	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Divides the signed 32-bit integer in bits 16-31 of two accumulators by the signed contents of a third accumulator. The quotient and remainder each occupy one accumulator.

The signed, 32-bit two's complement number contained in bits 16-31 of AC0 and bits 16-31 of AC1 is divided by the signed, 16-bit two's complement number in bits 16-31 of AC2. The quotient and remainder are signed, 16-bit numbers and occupy bits 16-31 of AC1 and AC0, respectively. The sign of the quotient is determined by the rules of algebra. The sign of the remainder is always the same as the sign of the dividend, except that a zero quotient or a zero remainder is always positive. Carry is set to 0. The contents of AC2 remain unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

> NOTE: *If the magnitude of the quotient is such that it will not fit into bits 16-31 of AC1, an overflow condition is indicated. Carry is set to 1, and the operation is terminated. The contents of AC0 and AC1 are unpredictable.*

Sign Extend and Divide
DIVX

1	0	1	1	1	1	1	1	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Extends the sign of one accumulator into a second accumulator and performs a *Signed Divide* on the result.

Extends the sign of the 16-bit number in bits 16-31 of AC1 into bits 16-31 of AC0 by placing a copy of bit 16 of AC1 in bits 16-31 of of AC0. After extending the sign, the instruction performs a *Signed Divide* operation. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Double Logical Shift
DLSH *acs,acd*

1	ACS		ACD		0	1	0	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Shifts the 32-bit number contained in bits 16-31 of ACD and bits 16-31 of ACD+1 either left or right depending on the number contained in bits 24-31 of ACS. The signed, 8-bit two's complement number contained in bits 24–31 of ACS determines the direction

of the shift and the number of bits to be shifted. If the number in bits 24–31 of ACS is positive, shifting is to the left; if the number in bits 24–31 of ACS is negative, shifting is to the right. If the number in bits 24–31 of ACS is zero, no shifting is performed. Bits 0–23 of ACS are ignored.

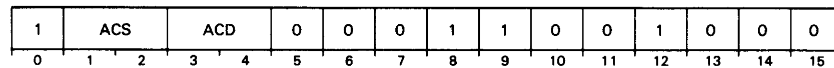
AC3+1 is AC0. The number of bits shifted is equal to the magnitude of the number in bits 24–31 of ACS. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. Carry and the contents of ACS remain unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *If the magnitude of the number in bits 24–31 of ACS is greater than 31_{10}, bits 16-31 of ACD are set to 0. Carry and the contents of ACS remain unchanged.*

Decimal Subtract

DSB *acs,acd*

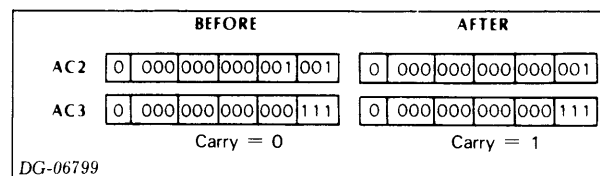


Performs decimal subtraction on 4-bit binary coded decimal (BCD) numbers and uses carry as a decimal borrow.

Subtracts the unsigned decimal digit contained in ACS bits 28–31 from the unsigned decimal digit contained in ACD bits 28–31. Subtracts the complement of carry from this result. Places the decimal units' position of the final result in ACD bits 28–31 and the complement of the decimal borrow in carry. In other words, if the final result is negative, the instruction indicates a borrow and sets carry to 0. If the final result is positive, the instruction indicates no borrow and sets carry to 1. The contents of ACS and bits 0–27 of ACD remain unchanged. *Overflow* is 0.

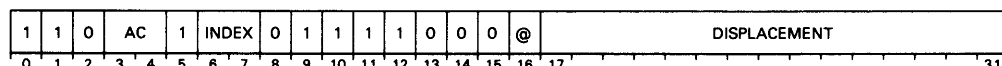
Example

Assume that bits 28–31 of AC2 contain 9; bits 28–31 of AC3 contain 7; and carry contains 0. After the instruction **DSB 3,2** is executed, AC3 remains the same; bits 28–31 of AC2 contain 1; and carry is set to 1, indicating no borrow from this *Decimal Subtract.*



Dispatch

DSPA *ac,[@]displacement[,index]*



Conditionally transfers control to an address selected from a table.

Computes the effective address E . This is the address of a *dispatch table.* The dispatch table consists of a table of addresses. Immediately before the table are two 16-bit, signed, two's complement limit words, L and H. The last word of the table is in location $E+H\text{-}L$.

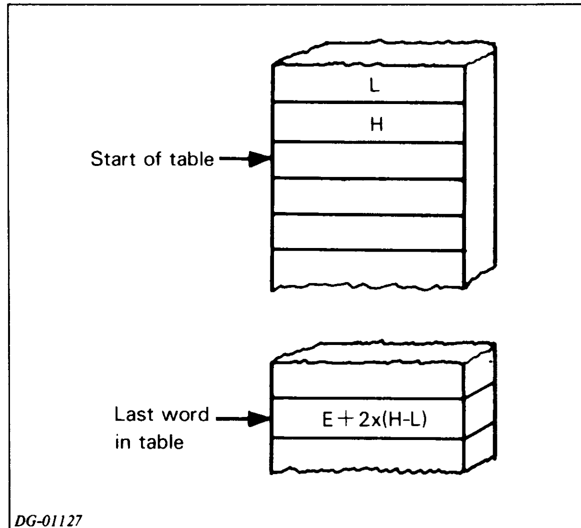


Figure 16.5

Compares the signed, two's complement number contained in bits 16-31 of the specified accumulator to the limit words. If the number in the accumulator is less than L or greater than H, sequential operation continues with the instruction immediately after the *Dispatch* instruction.

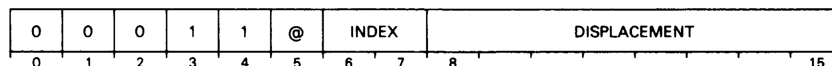
If the number in bits 16-31 of the specified accumulator is greater than or equal to L and less than or equal to H, the instruction fetches the word at location $E\text{-}L+number$. If the fetched word is equal to 177777_8, sequential operation continues with the instruction immediately after the *Dispatch* instruction. If the fetched word is not equal to 177777_8, the instruction treats this word as the intermediate address in the effective address calculation. After the indirection chain, if any, has been followed, the instruction places the effective address in the program counter and sequential operation continues with the word addressed by the updated value of the program counter.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

This instruction sets *overflow* to 0 and carry to 0.

Decrement And Skip If Zero

DSZ *[@]displacement[,index]*



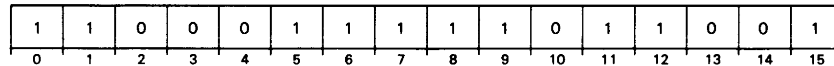
Decrements the addressed word, then skips if the decremented value is zero.

Decrements by one the word addressed by *E* and writes the result back into that location. If the updated value of the location is zero, the instruction skips the next sequential word. *Overflow* is 0 and carry remains unchanged.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Decrement the Word Addressed by WSP and Skip if Zero

DSZTS



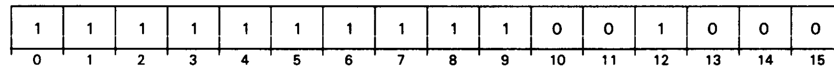
1	1	0	0	0	1	1	1	1	1	0	1	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Uses the contents of WSP (the wide stack pointer) as the address of a double word. Decrements the contents of the word addressed by WSP. If the decremented value is equal to zero, the instruction skips the next word. Carry is unchanged and *overflow* is 0.

NOTE: *The operation performed by this instruction is not indivisible.*

Load CPU Identification

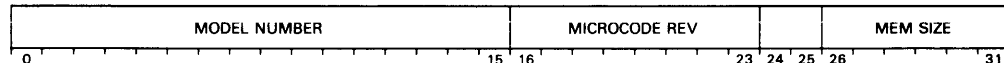
ECLID



1	1	1	1	1	1	1	1	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Loads a double word into AC0.

The double word has the format:



MODEL NUMBER	MICROCODE REV		MEM SIZE
0–15	16–23	24–25	26–31

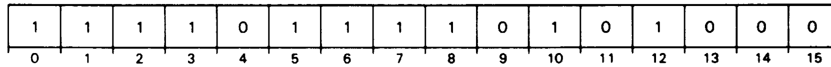
where

model # is the binary representation of the machine's model number,
microcode rev indicates the microcode revision currently in use on this machine,
mem size indicates the amount of physical memory on this machine. A zero in this field indicates 256 Kbytes of memory; a one indicates 512 Kbytes, and so on.

This instruction leaves carry unchanged. *Overflow* is 0.

NOTE: *When the C/350 MAP is enabled on the MV/8000, this instruction is used to identify the machine. The processor assumes AC0 to be 32 bits long for this instruction. If an interrupt occurs while* **ECLID** *is executing, however, the processor saves only bits 16–31 of AC0.*

Edit
EDIT



Converts a decimal number from either packed or unpacked form to a string of bytes under the control of an edit sub-program. This sub-program can perform many different operations on the number and its destination field, including leading zero suppression, leading or trailing signs, floating fill characters, punctuation control, and insertion of text into the destination field. The instruction also performs operations on alphanumeric data if data type 4 is specified.

The instruction maintains two flags and three indicators or pointers.

The flags are the significance Trigger (*T*) and the Sign flag (*S*). *T* is set to 1 when the first non-zero digit is processed unless otherwise specified by an edit op-code. At the beginning of an *Edit* instruction, *T* is set to 0. *S* is set to reflect the sign of the number being processed. If the number is positive, *S* is set to 0. If the number is negative, *S* is set to 1.

The three indicators are the Source Indicator (SI), the Destination Indicator (DI), and the op-code Pointer (P). Each is 16 bits wide and contains a byte pointer to the *current* byte in each respective area. At the beginning of an *Edit* instruction, SI is set to the value contained in bits 16-31 of AC3. DI is set to the value contained in bits 16-31 of AC2, and P is set to the value contained in bits 16-31 of AC0. Also at this time the sign of the source number is checked for validity.

The sub-program is made up of 8-bit op-codes followed by one or more 8-bit operands. P, a byte pointer, acts as the *program counter* for the *Edit* sub-program. The sub-program proceeds sequentially until a branching operation occurs - much the same way programs are processed. Unless instructed to do otherwise, the *Edit* instruction updates P after each operation to point to the next sequential op-code. The instruction continues to process 8-bit op-codes until directed to stop by the **DEND** op-code.

The sub-program can test and modify *S* and *T* as well as modify SI, DI and P.

Upon entry to **EDIT** bits 16-31 of AC0 contain a byte pointer to the first op-code of the *Edit* sub-program.

Bits 16-31 of AC1 contain the data-type indicator describing the number to be processed.

Bits 16-31 of AC2 contain a byte pointer to the the first byte of the destination field.

Bits 16-31 of AC3 contain a byte pointer to the first byte of the source field.

The fields may overlap in any way. However the instruction processes characters one at a time, so unusual side effects may be produced by certain types of overlap.

Upon successful termination, carry contains the significance Trigger; bits 16-31 of AC0 contain a byte pointer (P) to the next op-code to be processed; AC1 is undefined; bits 16-31 of AC2 contain a byte pointer (DI) to the next destination byte; and bits 16-31 of AC3 contain a byte pointer (SI) to the next source byte. *Overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

NOTES: *If SI is moved outside the area occupied by the source number, zeros will be supplied for numeric moves, even if SI is later moved back inside the source area.*

Some op-codes perform movement of characters from one string to another. For those op-codes which move numeric data, special actions may be performed. For those which move non-numeric data, characters are copied exactly to the destination.

The Edit *instruction places information on the stack. Therefore, the stack must be set up and have at least 9 words available for use.*

If the Edit *instruction is interrupted, it places restart information on the stack and places* 177777_8 *in AC0.*

If the initial contents of AC0 are equal to 177777_8 *the* Edit *instruction assumes it is restarting from an interrupt; therefore do not allow this to occur under any other circumstances.*

In the description of some of the *Edit* op-codes we use the symbol j to denote how many characters a certain operation should process. When the high order bit of j is 1, j has a different meaning, it is a pointer into the stack to a word that denotes the number of characters the instruction should process. So, in those cases where the high order bit of j is 1, the instructions interpret j as an 8-bit two's complement number pointing into the stack. The number on the stack is at the address:

stack pointer + 1 + j.

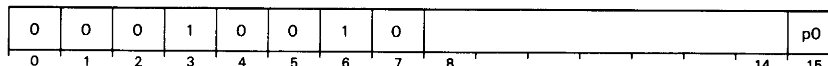
The operation uses the number at this address as a character count instead of j.

An *Edit* operation that processes numeric data (e.g., **DMVN**) skips a leading or trailing sign code it encounters; similarly, such an operation converts a high-order or low-order sign to its correct numeric equivalent.

The edit operations are as follows.

Add To DI

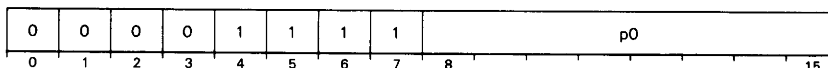
DADI *p0*



Adds the 8-bit two's complement integer specified by *p0* to the Destination Indicator (DI).

Add To P Depending On S

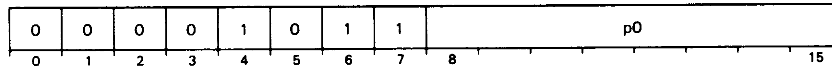
DAPS *p0*



If *S* is 0, the instruction adds the 8-bit two's complement integer specified by *p0* to the op-code Pointer (P). Before the add is performed, P is pointing to the byte containing the **DAPS** op-code.

Add To P Depending On T

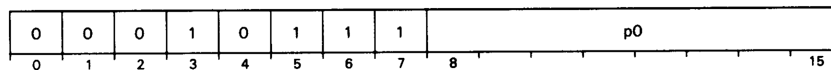
DAPT *p0*



If *T* is one, the instruction adds the 8-bit two's complement integer specified by *p0* to the op-code Pointer (P). Before the add is performed, P is pointing to the byte containing the **DAPT** op-code.

Add To P

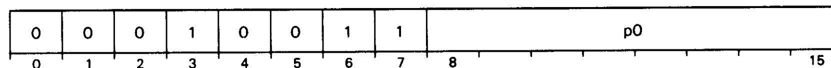
DAPU *p0*



Adds the 8-bit two's complement integer specified by *p0* to the op-code Pointer (P). Before the add is performed, P is pointing to the byte containing the **DAPU** op-code.

Add To SI

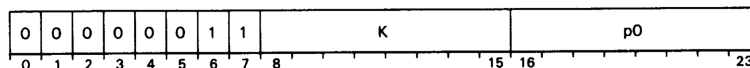
DASI *p0*



Adds the 8-bit two's complement integer specified by *p0* to the Source Indicator (SI).

Decrement and Jump If Non-Zero

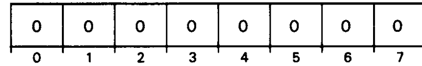
DDTK *k,p0*



Decrements a word in the stack by one. If the decremented value of the word is non-zero, the instruction adds the 8-bit two's complement integer specified by *p0* to the op-code Pointer (P). Before the add is performed, P is pointing to the byte containing the **DDTK** op-code. If the 8-bit two's complement integer specified by *k* is negative, the word decremented is at the address *stack pointer*$+1+k$. If *k* is positive, the word decremented is at the address *frame pointer*$+1+k$.

End Edit

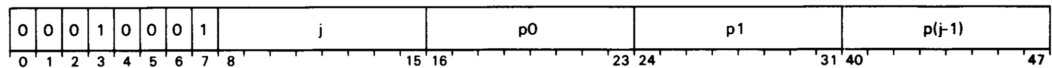
DEND



Terminates the EDIT sub-program.

Insert Characters Immediate

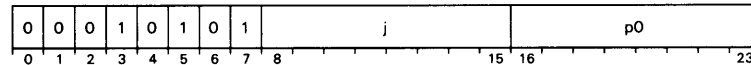
DICI *j,p0,p1,...,p(j-1)*



Inserts *j* characters from the op-code stream into the destination field beginning at the position specified by DI. Increases P by $j+2$, and increases DI by *j*.

Insert Character J Times

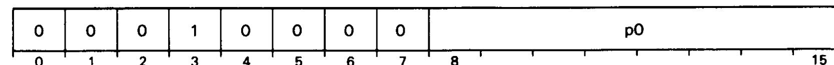
DIMC *j,p0*



Inserts the character specified by *p0* into the destination field a number of times equal to *j* beginning at the position specified by DI. Increases DI by *j*.

Insert Character Once

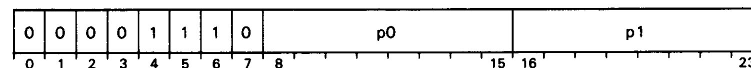
DINC *p0*



Inserts the character specified by *p0* in the destination field at the position specified by DI. Increments DI by 1.

Insert Sign

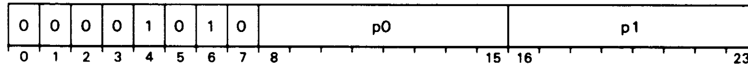
DINS *p0,p1*



If the Sign flag (*S*) is 0, the instruction inserts the character specified by *p0* in the destination field at the position specified by DI. If *S* is 1, the instruction inserts the character specified by *p1* in the destination field at the position specified by DI. Increments DI by one.

Insert Character Suppress

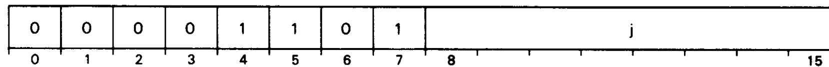
DINT *p0,p1*



If the significance Trigger (*T*) is 0, the instruction inserts the character specified by *p0* in the destination field at the position specified by DI. If *T* is 1, the instruction inserts the character specified by *p1* in the destination field at the position specified by DI. Increments DI by one.

Move Alphabetics

DMVA *j*

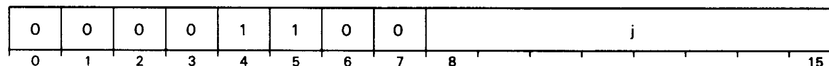


Moves *j* characters from the source field (beginning at the position specified by SI) to the destination field (beginning at the position specified by DI). Increases both SI and DI by *j*. Sets *T* to 1.

Initiates a commercial fault if the attribute specifier word indicates that the source field is data type 5 (packed). Initiates a commercial fault if any of the characters moved is not an alphabetic (A-Z, a-z, or space).

Move Characters

DMVC *j*

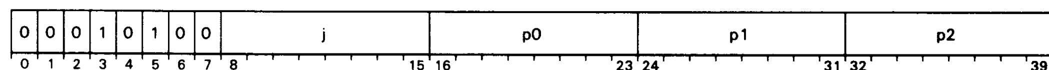


Increments SI if the source data type is 3 and $j>0$. The instruction then moves *j* characters from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. Increases both SI and DI by *j*. Sets *T* to 1.

Initiates a commercial fault if the attribute specifier word indicates that the source is data type 5 (packed). Performs no validation of the characters.

Move Float

DMVF *j,p0,p1,p2*



If the source data type is 3, $j>0$, and SI points to the sign of the source number, the instruction increments SI. Then for *j* characters, the instruction either places a digit substitute in the destination field beginning at the position specified by DI, or it moves a

digit from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. When *T* changes from 0 to 1, the instruction places both the digit substitute and the digit in the destination field, and increases SI by *j*. If *T* does not change from 0 to 1, the instruction increases DI by *j*. If *T* does change from 0 to 1, the instruction increases DI by $j+1$.

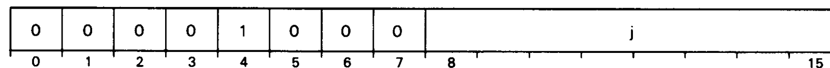
If the source data type is 2, the state of SI is undefined after the least significant digit has been processed.

If *T* is 1, the instruction moves each digit processed from the source field to the destination field. If *T* is 0 and the digit is a zero or space, the instruction places *p0* in the destination field. If *T* is 0 and the digit is a non-zero, the instruction sets *T* to 1 and the characters placed in the destination field depend on *S*. If *S* is 0, the instruction places *p1* in the destination field followed by the digit. If *S* is 1, the instruction places *p2* in the destination field followed by the digit.

The instruction initiates a commercial fault if any of the digits processed is not valid for the specified data type.

Move Numerics

DMVN *j*



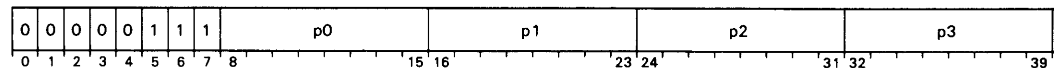
Increments SI if the source data type is 3 and $j>0$. The instruction then moves *j* characters from the source field beginning at the position specified by SI to the destination field beginning at the position specifed by DI. Increases both SI and DI by *j*. Sets *T* to 1.

Initiates a commercial fault if any of the characters moved is not valid for the specified data type.

For data type 2, the state of SI is undefined after the least significant digit has been processed.

Move Digit With Overpunch

DMVO *p0,p1,p2,p3*



Increments SI if the source data type is 3 and SI points to the sign of the source number. The instruction then either places a digit substitute in the destination field (at the position specified by DI), or it moves a digit plus overpunch from the source field (at the position specified by SI) to the destination field (at the position specified by DI). Increases both SI and DI by 1.

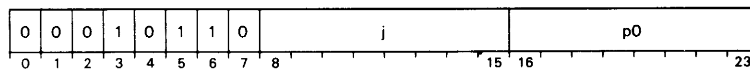
If the source data type is 2, the state of the SI is undefined after the least significant digit has been processed.

If the digit is a zero or space and *S* is 0, then the instruction places *p0* in the destination field. If the digit is a zero or space and *S* is 1, then the instruction places *p1* in the destination field. If the digit is a non-zero and *S* is 0, the instruction adds *p2* to the digit and places the result in the destination field. If the digit is a non-zero and *S* is 1, the instruction adds *p3* to the digit and places the result in the destination field. If the digit is a non-zero, the instruction sets *T* to 1. The instructions assumes *p2* and *p3* are ASCII characters.

The instruction initiates a commercial fault if the character is not valid for the specified data type.

Move Numeric With Zero Suppression

DMVS *j,p0*



Increments SI if the source data type is 3, $j>0$, and SI points to the sign of the source number. The instruction then moves *j* characters from the source field (beginning at the position specified by SI) to the destination field (beginning at the position specified by DI). Moves the digit from the source to the destination if *T* is 1. Replaces all zeros and spaces with *p0* as long as *T* is 0. Sets *T* to 1 when the first non-zero digit is encountered. Increases both SI and DI by *j*.

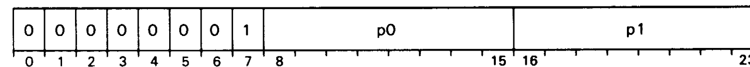
If the source data type is 2, the state of the SI is undefined after the least significant digit has been processed.

This op-code destroys the data type specifier.

Initiates a commercial fault if any of the characters moved is not a numeric (0-9 or space).

End Float

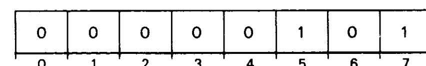
DNDF *p0,p1*



If *T* is 1, the instruction places nothing in the destination field and leaves DI unchanged. If *T* is 0 and *S* is 0, the instruction places *p0* in the destination field at the position specified by DI. If *T* is 0 and *S* is 1, the instruction places *p1* in the destination field at the position specified by DI. It increases DI by 1, and sets *T* to one.

Set S To One

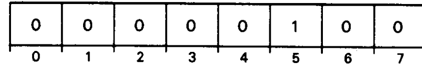
DSSO



Sets the Sign flag (S) to 1.

Set S To Zero

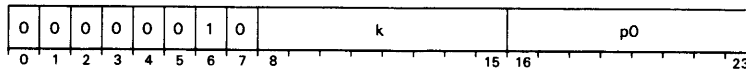
DSSZ



Sets the Sign flag (S) to 0.

Store In Stack

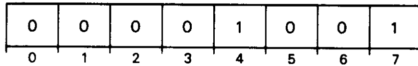
DSTK *k,p0*



Stores the byte specified by *p0* in bits 8–15 of a word in the stack. Sets bits 0–7 of the word that receives *p0* to 0. If the 8-bit two's complement integer specified by *k* is negative, the instruction addresses the word receiving *p0* by *stack pointer*+1+*k*. If *k* is positive, then the instruction stores *p0* at the address *frame pointer*+1+*k*.

Set T To One

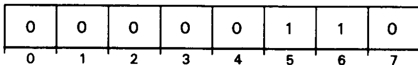
DSTO



Sets the significance Trigger (T) to 1.

Set T To Zero

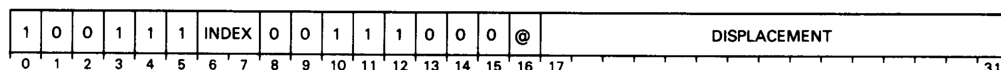
DSTZ



Sets the significance Trigger (T) to 0.

Extended Decrement and Skip if Zero

EDSZ *[@]displacement[,index]*



Decrements the addressed word, then skips if the decremented value is zero.

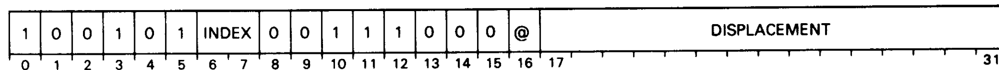
Computes the effective address, *E*. Decrements by one the contents of the location addressed by *E* and writes the result back into that location. If the updated value of the word is zero, the instruction skips the next sequential word.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

This instruction leaves carry unchanged. *Overflow* is 0.

Extended Increment And Skip If Zero

EISZ *[@]displacement[,index]*



Increments the addressed word, then skips if the incremented value is zero.

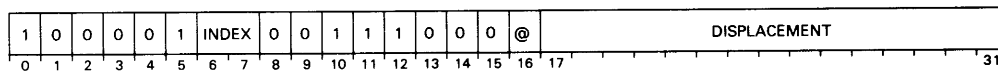
Computes the effective address, *E* . Increments by one the contents of the location specified by *E*, and writes the new value back into memory at the same address. If the updated value of the location is zero, the instruction skips the next sequential word.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

This instruction leaves carry unchanged. *Overflow* is 0.

Extended Jump

EJMP *[@]displacement[,index]*



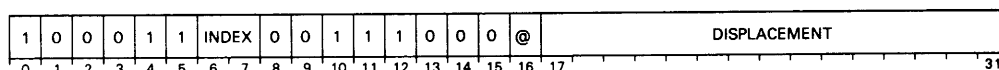
Computes the effective address, *E*, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Carry is unchanged and *overflow* is 0.

Extended Jump To Subroutine

EJSR *[@]displacement[,index]*



Increments and stores the value of the program counter in AC3, then places a new address in the program counter.

Computes the effective address, *E*. The instruction then places the address of the next sequential instruction (the instruction following the **EJSR** instruction) in AC3. Places *E* in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

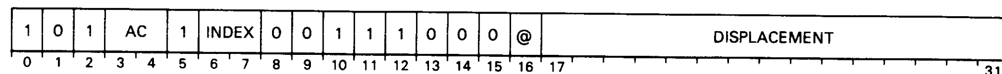
The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Overflow is 0 and carry is unchanged.

> **NOTE:** *The instruction computes* E *before it places the incremented program counter in AC3.*

Extended Load Accumulator

ELDA *ac,[@]displacement[,index]*



Moves a copy of the contents of a memory word into bits 16-31 of the specified accumulator.

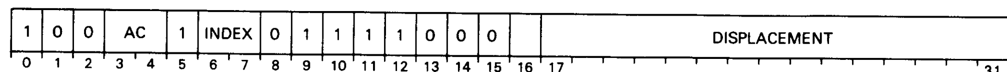
Calculates the effective address, *E*. Places the contents of the location addressed by *E* in bits 16-31 of the specified accumulator. The contents of the location addressed by *E* remain unchanged.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Carry remains unchanged and *overflow* is 0.

Extended Load Byte

ELDB *ac,displacement[,index]*



Copies a byte from memory into an accumulator.

Forms a byte pointer from the displacement in the following way: shifts the 16-bit number contained in the displacement field to the right one bit, producing a 15-bit address and a 1-bit byte indicator. Uses the value of the index bits to determine an offset value. Adds the offset value to the 15-bit address produced from the displacement to give a memory address. The byte indicator designates which byte of the addressed word will be loaded into bits 24–31 of the specified accumulator. The instruction sets bits 16–23 of the specified accumulator to 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

Carry is unchanged and *overflow* is 0.

The instruction destroys the previous contents of bits 16-31 of the specified accumulator, but it does not alter either the index value or the displacement.

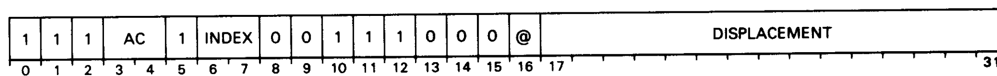
The argument *index* selects the source of the index value. It may have values in the range of 0–3. The meaning of each value is shown below:

Index Bits	Index Value
00	0
01	Address of the displacement field (Word 2 of this instruction)
10	Contents of bits 16-31 of AC2
11	Contents of bits 16-31 of AC3

This instruction sets *overflow* to 0 and carry to 0.

Load Effective Address

ELEF *ac,[@]displacement[,index]*

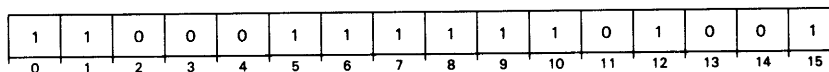


Places a 32-bit effective address constrained to be with the first 32 Kword of the current segment in an accumulator.

Sets bit 0 of the accumulator to 0. *Overflow* is 0 and carry is unchanged.

Enqueue Towards the Head

ENQH



Enqueues a data element.

AC0 contains the effective address of a queue descriptor.

AC1 contains an effective address of a data element in the queue defined by AC0.

AC2 contains the effective address of the data element to be added to the queue.

The instruction checks the page or pages that contain the element for valid read and write access privileges. If the privileges are invalid, the appropriate protection fault occurs and the queue remains unchanged.

If the privileges are valid, the instruction checks the queue descriptor addressed by AC0. If the queue descriptor indicates an empty queue, the instruction ignores the contents of AC1, places the data element addressed by AC2 in the queue, and updates the queue descriptor. The next sequential word is executed.

If the descriptor indicates a queue that contains data elements, the instruction prepares to enqueue a new data element; the instruction enqueues the data element addressed by AC2 *before* the data element addressed by AC1. If the new data element becomes the head of the queue, the instruction updates the queue descriptor appropriately. The next sequential word is skipped.

> NOTE: *If the processor references a page containing a link, the instruction completely updates that link before the processor references another link or descriptor. This means the instruction executes correctly even if only one page is resident in memory.*

The instruction checks all reads and writes of links in data elements and queue descriptors against the current ring. Ring numbers of the link addresses must be greater than or equal to the current ring.

The enqueue operation is not interruptable. The entire operation completes before any interrupts are enabled.

The instruction leaves the contents of AC0, AC1, AC2, and AC3 unchanged. Carry is unchanged and *overflow* is 0.

Enqueue Towards the Tail

ENQT

1	1	0	0	0	1	1	1	1	1	1	1	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Enqueues a data element.

AC0 contains the effective address of a queue descriptor.

AC1 contains an effective address of a data element in the queue defined by AC0.

AC2 contains the effective address of the data element to be added to the queue.

The instruction checks the page or pages that contain the element for valid read and write access privileges. If the privileges are invalid, the appropriate protection fault occurs and the queue remains unchanged.

If the privileges are valid, the instruction checks the queue descriptor addressed by AC0. If the queue descriptor indicates an empty queue, the instruction ignores the contents of AC1 and enqueues the data element addressed by AC2. The instruction updates the queue descriptor if necessary, then the next sequential word is executed.

If the descriptor indicates a queue that contains data elements, the instruction prepares to enqueue a new data element; the instruction enqueues the data element addressed by AC2 *after* the data element addressed by AC1. If the new data element becomes the tail of the queue, then the instruction updates the queue descriptor appropriately. The next sequential word is skipped.

> NOTE: *If the processor references a page containing a link, the instruction completely updates that link before the processor references another link or descriptor. This means that this instruction will execute correctly even if only one page is resident in memory.*

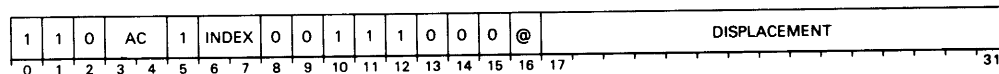
The instruction checks all reads and writes of links in data elements and queue descriptors against the current ring. Ring numbers of the link addresses must be greater than or equal to the current ring.

The enqueue operation is not interruptable. The entire operation completes before any interrupts are enabled.

The instruction leaves the contents of AC0, AC1, AC2, and AC3 unchanged. Carry is unchanged and *overflow* is 0.

Extended Store Accumulator

ESTA *ac,[@]displacement[,index]*



Stores the contents of bits 16-31 of an accumulator into a memory location.

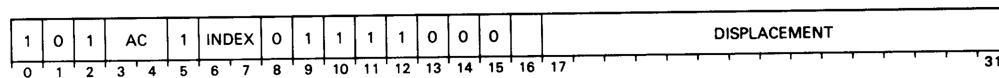
The contents of bits 16-31 of the specified accumulator are placed in the word addressed by the effective address, *E*. The previous contents of the location addressed by *E* are lost. The contents of the specified accumulator and carry remain unchanged.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kwords of the current segment.

Overflow is 0.

Extended Store Byte

ESTB *ac,displacement[,index]*



Copies into memory the byte contained in bits 24-31 of an accumulator.

Forms a byte pointer from the displacement as follows: shifts the 16-bit number contained in the displacement field to the right one bit, producing a 15-bit address and a 1-bit byte indicator. Uses the value of the index bits to determine an offset value. Adds the offset value to the 15-bit address produced from the displacement field to give a memory address. The byte indicator determines which byte of the addressed location will receive bits 24–31 of the specified accumulator.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

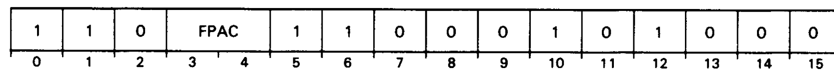
The argument *index* selects the source of the index value. It may have values in the range of 0–3; the meaning of each value is shown below:

Index Bits	Index Value
00	0
01	Address of the displacement field (Word 2 of this instruction)
10	Contents of bits 16-31 of AC2
11	Contents of bits 16-31 of AC3

This instruction leaves carry unchanged; *overflow* is 0.

Absolute Value

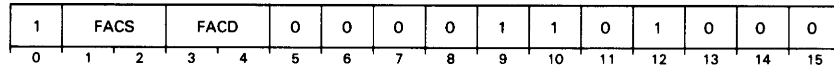
FAB *fpac*



Sets the sign bit of FPAC to 0. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

Add Double (FPAC to FPAC)

FAD *facs,facd*



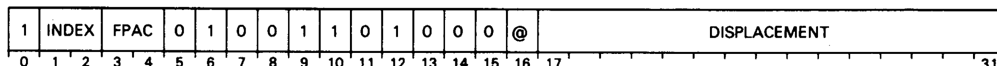
Adds the 64-bit floating point number in FACS to the 64-bit floating point number in FACD.

Adds the 64-bit floating point number in FACS to the 64-bit floating point number in FACD. Places the normalized result in FACD. Leaves the contents of FACS unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FACD.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Add Double (Memory to FPAC)

FAMD *fpac,[@]displacement[,index]*



Adds the 64-bit floating point number in the source location to the 64-bit floating point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Adds this 64-bit floating point number to the floating point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point

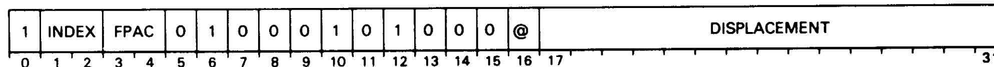
status register to reflect the new contents of FPAC.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kwords of the current segment.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Add Single (Memory to FPAC)

FAMS *fpac,[@]displacement[,index]*



Adds the 32-bit floating point number in the source location to the 32-bit floating point number in FPAC and places the normalized result in FPAC.

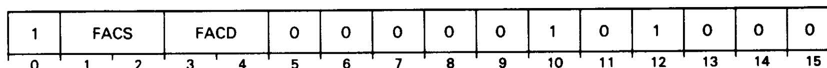
Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Adds this 32-bit floating point number to the floating point number in bits 0-31 of the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kwords of the current segment.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Add Single (FPAC to FPAC)

FAS *facs,facd*



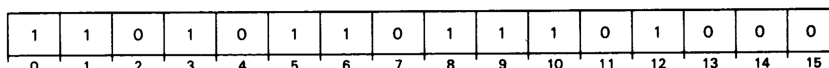
Adds the 32-bit floating point number in bits 0-31 of FACS to the 32-bit floating point number in bits 0-31 of FACD.

Adds the 32-bit floating point number in bits 0-31 of ACS to the 32-bit floating point number in bits 0-31 of FACD. Places the normalized result in FACD. Leaves the contents of FACS unchanged. Sets bits 32-63 of FACD to 0 and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FACD.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Clear Errors

FCLE



Sets bits 0–4 of the floating point status register to 0.

NOTES: *Since this instruction sets the* ANY *bit of the FPSR to 0, the FPPC field is undefined.*

The IORST *instruction and the system reset function will also set these bits to 0.*

Compare Floating Point

FCMP *facs,facd*

1	FACS	FACD	1	1	1	0	0	1	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Compares two floating point numbers and sets the *Z* and *N* flags in the floating point status register accordingly.

Algebraically compares the floating point numbers in FACS and FACD to each other. Updates the *Z* and *N* flags in the floating point status register to reflect the result. The contents of FACS and FACD remain unchanged. The results of the compare and the corresponding flag settings are shown in the table below.

Z	N	Result
1	0	FACS=FACD
0	1	FACS>FACD
0	0	FACS<FACD

Divide Double (FPAC by FPAC)

FDD *facs,facd*

1	FACS	FACD	0	0	1	1	1	1	0	1	0	0	0
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Divides the floating point number in FACD by the floating point number in FACS and places the normalized result in FACD. Destroys the previous contents of FACD. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FACD. The contents of FACS remain unchanged.

Checks the floating point number contained in FACS for a zero mantissa. If the mantissa is zero, sets the *DVZ* bit in the floating point status register and terminates. The number in FACD remains unchanged.

If the mantissa is nonzero, compares the mantissas of the numbers contained in FACS and FACD. If the mantissa of the number in FACD is greater than or equal to the mantissa of the number in FACS, the instruction shifts the mantissa of the number in FACD right one hex digit and adds one to the exponent of the number in FACD.

After aligning the mantissas, the instruction divides the mantissa in FACD by the mantissa in FACS. The mantissa of the quotient becomes the mantissa of the intermediate result. The two operands and the rules of algebra determine the sign of the intermediate result. To calculate the exponent of the intermediate result in excess 64 representation,

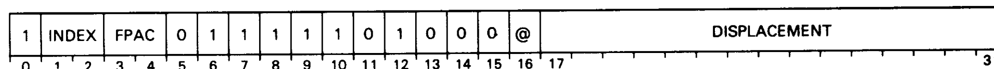
the instruction subtracts the exponent in FACS from the exponent in FACD and adds 64 to this result. The result is already normalized by the shifting algorithm described in the paragraph above, so the instruction places the result in FACD unaltered. Updates the *Z* and *N* flags to reflect the new contents of FACD.

If the exponent processing produces either overflow or underflow, the instruction sets the corresponding bit in the floating point status register. If overflow occurs, the sign and the mantissa in FACD are correct but the exponent is 128 too small. If underflow occurs, the sign and the mantissa in FACD are correct but the exponent is 128 too large.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Divide Double (FPAC by Memory)

FDMD *fpac,[@]displacement[,index]*



Divides the 64-bit floating point number in FPAC by the 64-bit floating point number in the source location and places the normalized result in FPAC.

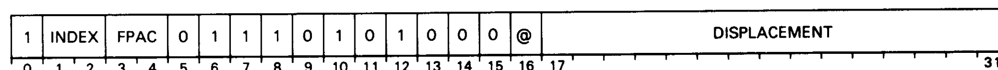
Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Divides the floating point number in the specified FPAC by this 64-bit floating point number. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Divide Single (FPAC by Memory)

FDMS *fpac,[@]displacement[,index]*



Divides the 32-bit floating point number in bits 0-31 of FPAC by the 32-bit floating point number in the source location and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Divides the floating point number in bits 0-31 of the specified FPAC by this 32-bit floating point number. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Divide Single (FPAC by FPAC)

FDS *facs,facd*

1	FACS	FACD	0	0	1	1	0	1	0	1	0	0	0
0	1–2	3–4	5	6	7	8	9	10	11	12	13	14	15

Divides the floating point number in bits 0–31 of FACD by the floating point number in bits 0–31 of FACS and places the normalized result in FACD. Destroys the previous contents of FACD. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FACD. The contents of FACS remain unchanged.

Checks the floating point number contained in FACS for a zero mantissa. If the mantissa is zero, sets the *DVZ* bit in the floating point status register and terminates. The number in FACD remains unchanged.

If the mantissa is nonzero, compares the mantissas of the numbers contained in FACS and FACD. If the mantissa of the number in FACD is greater than or equal to the mantissa of the number in FACS, the instruction shifts the mantissa of the number in FACD right one hex digit and adds one to the exponent of the number in FACD.

After aligning the mantissas, the instruction divides the mantissa in FACD by the mantissa in FACS. The mantissa of the quotient becomes the mantissa of the intermediate result. The two operands and the rules of algebra determine the sign of the intermediate result. To calculate the exponent of the intermediate result in excess 64 representation, the instruction subtracts the exponent in FACS from the exponent in FACD and adds 64 to this result. The result is already normalized by the shifting algorithm described in the paragraph above, so the instruction places the result in FACD unaltered. Updates the *Z* and *N* flags to reflect the new contents of FACD.

If the exponent processing produces either overflow or underflow, the instruction sets the corresponding bit in the floating point status register. If overflow occurs, the sign and the mantissa in FACD are correct but the exponent is 128 too small. If underflow occurs, the sign and the mantissa in FACD are correct but the exponent is 128 too large.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Load Exponent

FEXP *fpac*

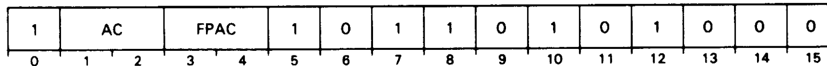
1	0	1	FPAC	1	1	0	0	1	1	0	1	0	0	0
0	1	2	3–4	5	6	7	8	9	10	11	12	13	14	15

Loads an exponent into bits 1–7 of an accumulator.

The instruction places bits 17–23 of AC0 in bits 1–7 of the specified FPAC. Ignores bits 0–16 and 24–31 of AC0. Changes the *Z* and *N* flags in the floating point status register to reflect the contents of FPAC. AC0 and bits 0 and 8–63 of FPAC remain unchanged. If FPAC contains true zero, the instruction does not load bits 1–7 of FPAC.

Fix To AC

FFAS *ac,fpac*



Converts the integer portion of the floating point number contained in the specified FPAC to a signed two's complement integer. Places the result in the specified accumulator.

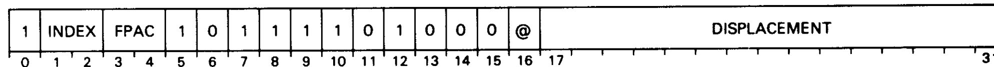
If the integer portion of the number contained in FPAC is less than -32,769 or greater than +32,768, the instruction sets *MOF* in the FPSR to 1. Takes the absolute value of the integer portion of the number contained in the FPAC. Takes the 15 least significant bits of the absolute value and appends a 0 onto the leftmost bit to giver a 16-bit number. If the sign of the number is negative, formes the two's complement of the 16-bit result. Places the 16-bit integer in bits 16-31 of the specified accumulator.

If the integer portion is within the range of -32,768 to +32,767 inclusive, the instruction places the 16-bit, two's complement of the integer portion of the number contained in the FPAC in bits 16-31 of the specified accumulator.

The instruction leaves the FPAC and the *Z* and *N* flags of the FPSR unchanged.

Fix To Memory

FFMD *fpac,[@]displacement[,index]*



Converts the integer portion of the floating point number contained in the specified FPAC to a signed two's complement integer. Places the result in a memory location.

Calculates the effective address, *E*. If the integer portion of the number contained in FPAC is less than -2,147,483,649 or greater than +2,147,483,648, the instruction sets *MOF* in the FPSR to 1. Takes the absolute value of the integer portion of the number contained in the FPAC. Takes the 31 least significant bits of the absolute value and appends a 0 onto the leftmost bit to give a 32-bit number. If the sign of the number is negative, forms the two's complement of the 32-bit result. Places the 32-bit integer in the memory locations specified by *E*.

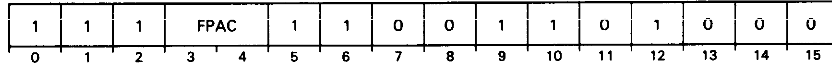
If the integer portion is within the range of -2,147,483,648 to +2,147,483,647 inclusive, the instruction places the 32-bit, two's complement of the integer portion of the number contained in the FPAC in the memory locations specified by *E*.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

The instruction leaves the FPAC and the *Z* and *N* flags of the FPSR unchanged.

Halve

FHLV *fpac*



Divides the floating point number in FPAC by 2.

Shifts the mantissa contained in FPAC right one bit position. Fills the vacated bit position with a zero and places the bit shifted out in the guard digit. Normalizes the number and places the result in FPAC. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

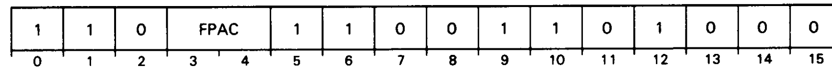
If underflow occurs, sets the *UNF* flag in the floating point status register to 1. In this case, the mantissa and sign in FPAC are correct, but the exponent is 128 too large.

This instruction does rounding.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Integerize

FINT



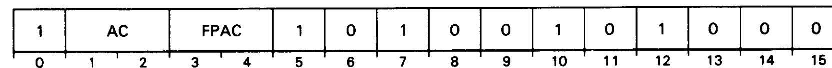
Zeros the fractional portion (if any) of the number contained in the specified FPAC. Normalizes the result. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of the specified FPAC.

> **NOTE:** *If the absolute value of the number contained in the specified FPAC is less than 1, the specified FPAC is set to true zero.*
>
> *This instruction truncates towards zero, and does not do rounding.*

See Chapter 8 and Appendix G for more information about floating point manipulation.

Float From AC

FLAS *ac,fpac*

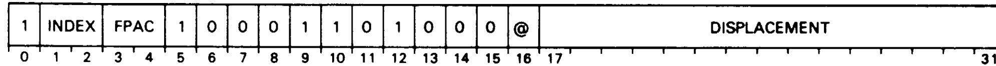


Converts a two's complement number in the range of -32,768 to +32,767 inclusive to floating point format.

Converts the signed two's complement number contained in bits 16–31 of the specified accumulator to a single precision floating point number. Places the result in the high-order 32 bits of the specified FPAC. Sets the low-order 32 bits of the FPAC to 0. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC. The contents of the specified accumulator remain unchanged.

Load Floating Point Double

FLDD *fpac,[@]displacement[,index]*



Moves four words out of memory and into a specified FPAC.

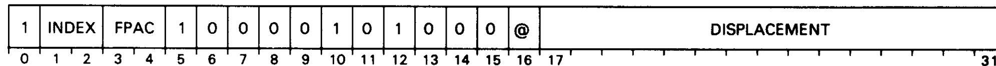
Computes the effective address, *E*. Fetches the double precision floating point number at the address specified by *E* and places it in FPAC. Updates the *Z* and *N* flags in the FPSR to reflect the new contents of FPAC.

The 32-bit effective address generated by this instruction is contrained to be within the first 32 Kword of the current segment.

NOTE: *This instruction will move unnormalized data without change.*

Load Floating Point Single

FLDS *fpac,[@]displacement[,index]*



Moves two words out of memory into a specified FPAC.

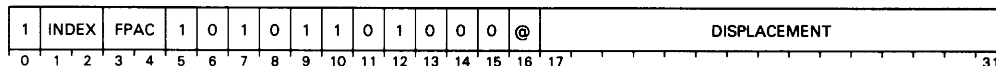
Computes the effective address, *E*. Fetches the single precision floating point number at the address specified by *E*. Places the number in the high-order bits of FPAC. Sets the low-order 32 bits of FPAC to 0. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

NOTE: *This instruction will move unnormalized or illegal data without change.*

Float From Memory

FLMD *fpac,[@]displacement[,index]*



Converts the contents of two 16-bit memory locations to floating point format and places the result in a specified FPAC.

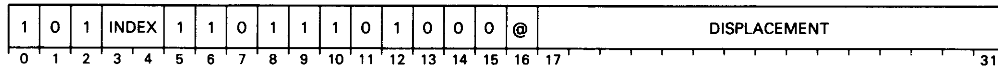
Computes the effective address, *E*. Converts the 32-bit, signed, two's complement number addressed by *E* to a double precision floating point number. Places the result in the specified FPAC. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of the FPAC.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

The range of numbers that you can convert is -2,147,483,648 to +2,147,483,647 inclusive.

Load Floating Point Status

FLST *[@]displacement[,index]*



Moves two words out of memory into the floating point status register.

Computes the effective address, *E*. Places the 32-bit operand addressed by *E* in the floating point status register as follows:

- Places bits 0-15 of the operand in bits 0-15 of the FPSR. Sets bits 16-32 of the FPSR to 0.
- If *ANY* is 0, bits 33-63 of the FPSR (the FPPC) are undefined.
- If *ANY* is 1, the instruction places the value of the current segment in bits 33-35 of the FPSR, zeroes in bits 36-48, and bits 17-31 of the operand in bits 49-63 of the FPSR.

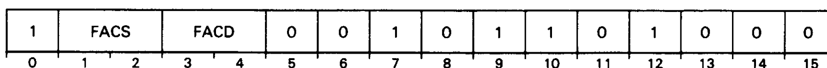
The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

> NOTES: *This instruction does not set the* ANY *flag from memory. If any of bits 1-4 are loaded as 1,* ANY *is set to 1; otherwise,* ANY *is 0.*
>
> *Bits 12-15 of the FPSR are not set from memory. These bits are the floating point identification code an are read protected. In the MV/8000 they are set to 0111.*
>
> *This instruction initiates a floating point trap if* ANY *and* TE *are both 1 after the FPPC is loaded.*

See Chapter 8 and Appendix G for more information about floating point manipulation.

Multiply Double (FPAC by FPAC)

FMD *facs,facd*



Multiplies the floating point number in FACD by the floating point number in FACS and places the normalized result in FACD. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FACD. The contents of FACS remain unchanged.

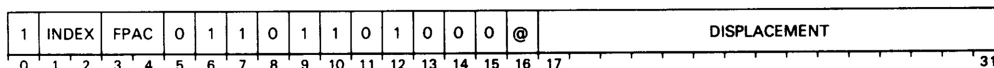
The instruction multiplies the mantissas of the two numbers together. The result is the mantissa of the intermediate result. The two operands and the rules of algebra determine the sign of the intermediate result. Adds the exponents of the two numbers together and subtracts 64 from the result to maintain excess 64 notation. This value becomes the exponent of the intermediate result. Normalizes the intermediate result if necessary and loads the result into FACD. Updates the *Z* and *N* flags in the floating point status register.

If the exponent processing produces either overflow or underflow, the result is held until normalization, as that procedure may correct the condition. If normalization does not correct the condition, the instruction sets the corresponding flag in the floating point status register to 1. The mantissa and sign of the number will be correct, but the exponent will be 128 too small if overflow occurred; or 128 too large if underflow occurred.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Multiply Double (FPAC by Memory)

FMMD *fpac,[@]displacement[,index]*



Multiplies the 64-bit floating point number in the source location by the 64-bit floating point number in FPAC and places the normalized result in FPAC.

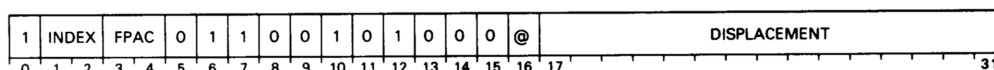
Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Multiplies this 64-bit floating point number by the floating point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kword of the current segment.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Multiply Single (FPAC by Memory)

FMMS *fpac,[@]displacement[,index]*



Multiplies the 32-bit floating point number in the source location by the 32-bit floating point number in bits 0-31 of FPAC and places the normalized result in FPAC.

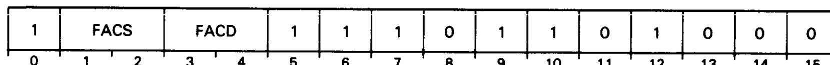
Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Multiplies this 32-bit floating point number by the floating point number in bits 0-31 of the specified FPAC. Places the normalized result in bits 0-31 of the specified FPAC. Sets bits 32-63 of FPAC to 0. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Move Floating Point

FMOV *facs,facd*

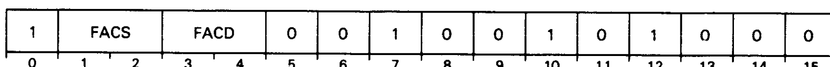


Moves the contents of one FPAC to another FPAC.

Places the contents of FACS in FACD. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FACD. The contents of FACS remain unchanged.

Multiply Single (FPAC by FPAC)

FMS *facs,facd*



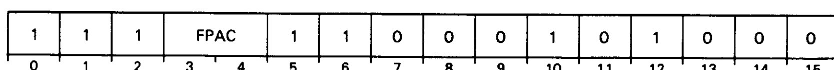
Multiplies the 32-bit floating point number in bits 0-31 of FACS by the 32-bit floating point number in bits 0-31 of FACD.

Multiplies the 32-bit floating point number in bits 0-31 of ACS by the 32-bit floating point number in bits 0-31 of FACD. Places the normalized result in FACD. Leaves the contents of FACS unchanged. Sets bits 32-63 of FACD to 0 and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FACD.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Negate

FNEG *fpac*

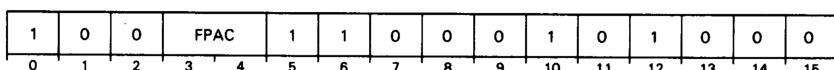


Inverts the sign bit of FPAC. Leaves bits 1–63 of FPAC unchanged. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

If FPAC contains true zero, leaves the sign bit unchanged.

Normalize

FNOM *fpac*



Normalizes the floating point numbers in FPAC. Sets a true zero in FPAC if all the bits of the mantissa are zero. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

If an exponent underflow occurs, sets the *UNF* flag in the floating point status register. In this case, the mantissa and the sign of the number in FPAC are correct, but the exponent is 128 too large.

NOTE: *This instruction does not do rounding.*

See Chapter 8 and Appendix G for more information about floating point manipulation.

No Skip

FNS

1	0	0	0	0	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The next sequential word is executed.

Pop Floating Point State

FPOP

1	1	1	0	1	1	1	0	1	1	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pops the state of the floating point unit off the narrow stack.

Pops an 18-word block off the narrow stack and loads the contents into the FPSR and the four FPACs. The format of the 18-word block is shown below.

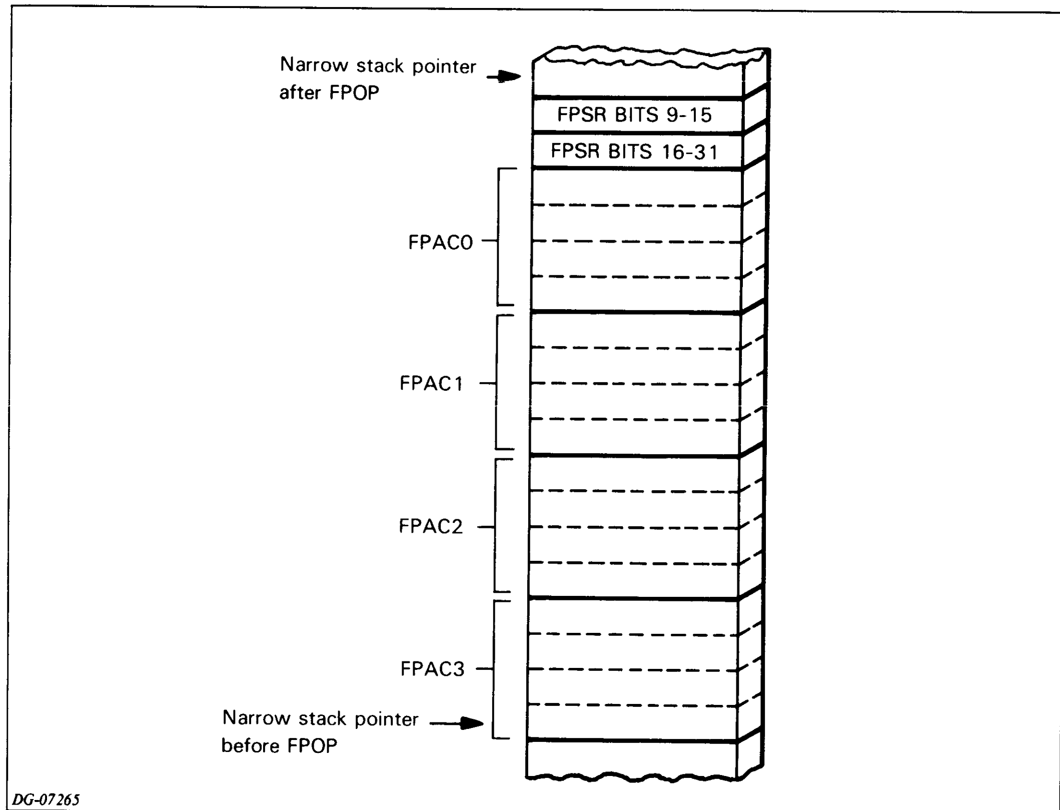


Figure 16.6

The instruction pops the first 32-bit operand on the stack and places it in the FPSR as follows:

- Places bits 0-15 of the operand in bits 0-15 of the FPSR.
- If *ANY* is 0, bits 16-31 of the FPSR are undefined.
- If *ANY* is 1, the instruction places bits 16-31 of the popped operand into bits 16-31 of the FPSR.

The rest of the stack words are popped in the usual way. See Chapter 8 for more information.

The 32-bit effective address generated by this instruction is constrainted to be within the first 32 Kword of the current segment.

NOTES: *This instruction moves unnormalized data without change.*

This instruction does not set the ANY *flag from memory. If any of bits 1-4 are loaded as 1,* ANY *is set to 1; otherwise,* ANY *is 0.*

Bits 12-15 of the FPSR are not set from memory. These bits are the floating point identification code an are read protected. In the MV/8000 they are set to 0111.

This instruction does not initiates a floating point trap under any conditions of the FPSR.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Push Floating Point State
FPSH

1	1	1	0	0	1	1	0	1	1	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pushes an 18-word floating point return block onto the narrow stack, leaving the contents of the floating point accumulators and the floating point status register unchanged. The format of the 18 words pushed is as follows:

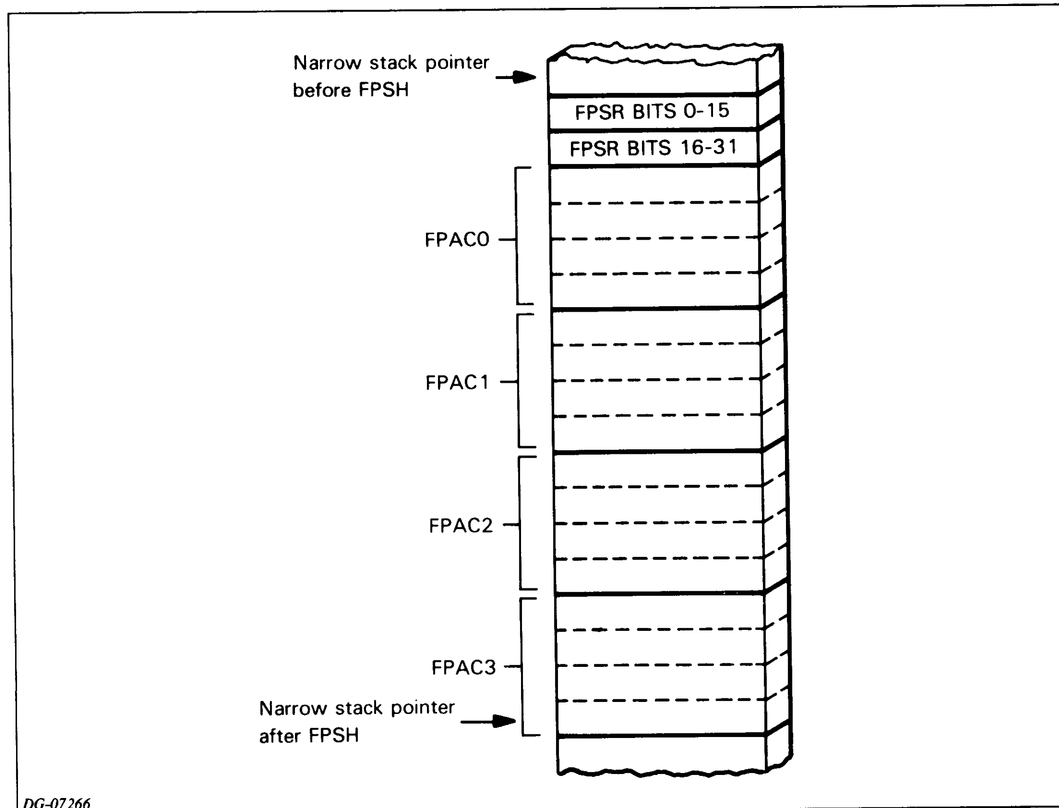


Figure 16.7

The instruction pushes the contents of the FPSR as follows:

- Stores bits 0-15 of the FPSR in the first memory word.
- If *ANY* is 0, the contents of the second memory word are undefined.
- If *ANY* is 1, the instruction stores bits 16-31 of the FPSR into the second memory word.

The rest of the block is pushed after the FPSR has been pushed.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

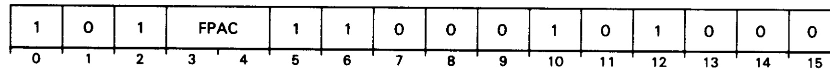
NOTES: *This instruction moves unnormalized data without change.*

This instruction does not initiate a floating point trap under any conditions of the FPSR.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Read High Word

FRH *fpac*

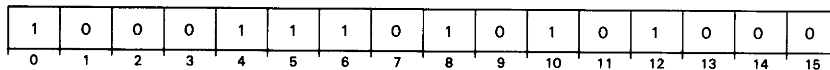


Places the high-order 16 bits of FPAC in bits 16–31 of AC0. FPAC and the *Z* and *N* flags in the floating point status register remain unchanged.

NOTE: *This instruction moves unnormalized data without change.*

Skip Always

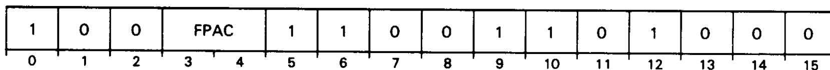
FSA



Skips the next sequential word.

Scale

FSCAL *fpac*



Shifts the mantissa of the floating point number in FPAC either right or left, depending upon the contents of bits 17–23 of AC0. Leaves the contents of AC0 unchanged.

Bits 17–23 of AC0 contain an exponent.

The instruction subtracts the exponent of the number contained in FPAC from the exponent in AC0. The difference between the exponents specifies *D*, a number of hex digits.

If *D* is zero, the instruction updates the *Z* and *N* flags, and stops.

If *D* is positive, the instruction shifts the mantissa of the number contained in FPAC to the right by *D* digits.

If *D* is negative, the instruction shifts the mantissa of the number contained in FPAC to the left by *D* digits. Sets the *MOF* flag in the floating point status register.

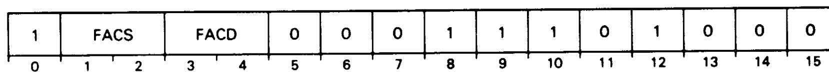
After the right or left shift, the instruction loads the contents of bits 17–23 of AC0 into the exponent field of FPAC. Bits shifted out of either end of the mantissa are lost. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

NOTE: *This instruction does not do rounding.*

See Chapter 8 and Appendix G for more information about floating point manipulation.

Subtract Double (FPAC from FPAC)

FSD *facs,facd*

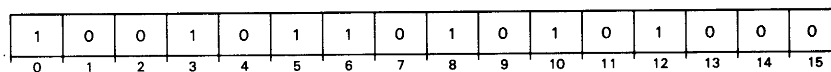


Subtracts the 64-bit floating point number in FACS from the 64-bit floating point number in FACD. Places the normalized result in FACD. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FACD. The contents of FACS remain unchanged.

Refer to Chapter 8 and Appendix G for more information about floating point manipulation.

Skip On Zero

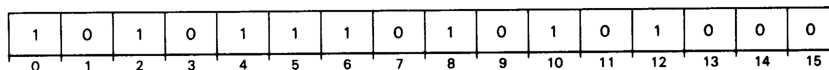
FSEQ



Skips the next sequential word if the *Z* flag of the floating point status register is 1.

Skip On Greater Than Or Equal To Zero

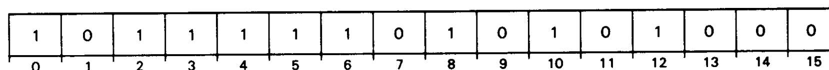
FSGE



Skips the next sequential word if the *N* flag of the floating point status register is 0.

Skip On Greater Than Zero

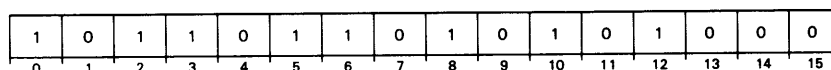
FSGT



Skips the next sequential word if both the *Z* and *N* flags of the floating point status register are 0.

Skip On Less Than Or Equal To Zero

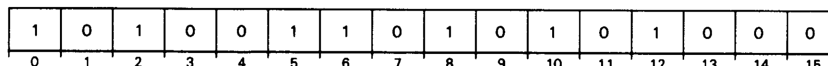
FSLE



Skips the next sequential instruction if either the *Z* flag or the *N* flag of the floating point status register is 1.

Skip On Less Than Zero

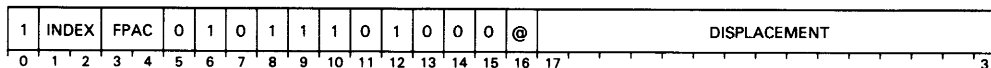
FSLT



Skips the next sequential word if the *N* flag of the floating point status register is 1.

Subtract Double (Memory from FPAC)

FSMD *fpac,[@]displacement[,index]*



Subtracts the 64-bit floating point number in the source location from the 64-bit floating point number in FPAC and places the normalized result in FPAC.

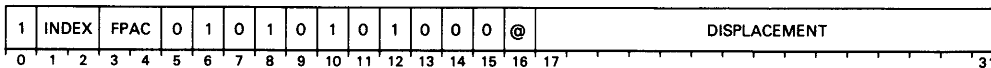
Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Subtracts this 64-bit floating point number from the floating point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Subtract Single (Memory from FPAC)

FSMS *fpac,[@]displacement[,index]*



Subtracts the 32-bit floating point number in the source location from the 32-bit floating point number in bits 0-31 of FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Subtracts this 32-bit floating point number from the floating point number in bits 0-31 of the specified FPAC. Places the normalized result in the specified FPAC. Sets bits 32-63 of FPAC to 0. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Skip On No Zero Divide

FSND

1	1	0	0	1	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word if the *DVZ* flag of the floating point status register is 0.

Skip On Non-Zero

FSNE

1	0	0	1	1	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word if the *Z* flag of the floating point status register is 0.

Skip On No Error

FSNER

1	1	1	1	1	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word if bits 1–4 of the floating point status register are all 0.

Skip On No Mantissa Overflow

FSNM

1	1	0	0	0	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word if the *MOF* flag of the floating point status register is 0.

Skip On No Overflow

FSNO

1	1	1	0	0	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word if the *OVF* flag of the floating point status register is 0.

Skip On No Overflow and No Zero Divide
FSNOD

1	1	1	0	1	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word if both the *OVF* flag and the *DVZ* flag of the floating point status register are 0.

Skip On No Underflow
FSNU

1	1	0	1	0	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word if the *UNF* flag of the floating point status register is 0.

Skip On No Underflow And No Zero Divide
FSNUD

1	1	0	1	1	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word if both the *UNF* flag and the *DVZ* flag of the floating point status register are 0.

Skip On No Underflow And No Overflow
FSNUO

1	1	1	1	0	1	1	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Skips the next sequential word if both the *UNF* flag and the *OVF* flag of the floating point status register are 0.

Subtract Single (FPAC from FPAC)
FSS *facs,facd*

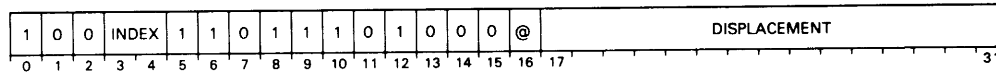
1	FACS		FACD		0	0	0	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Subtracts the 32-bit floating point number in bits 0-31 of FACS from the 32-bit floating point number in bits 0-31 of FACD. Places the normalized result in bits 0-31 of FACD. Sets bits 32-63 of FACD to 0. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FACD. The contents of FACS remain unchanged.

Refer to Chapter 8 and Appendix G for more information about floating point manipulation.

Store Floating Point Status

FSST *[@]displacement[,index]*



Moves the contents of the narrow FPSR into memory.

Computes the effective address, *E*, of two sequential, 16-bit locations in memory. Stores the contents of the FPSR in these locations as follows:

- Stores bits 0-15 of the FPSR in the first memory word.
- If *ANY* is 0, the contents of the second memory word are undefined.
- If *ANY* is 1, the instruction stores bits 48-63 of the FPSR into the second memory word.

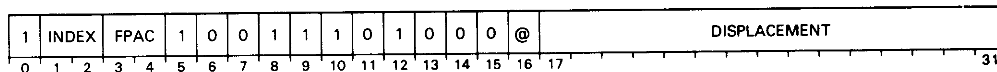
The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

> **NOTE:** *This instruction does not initiate a floating point trap under any conditions of the FPSR.*

See Chapter 8 and Appendix G for more information about floating point manipulation.

Store Floating Point Double

FSTD *fpac,[@]displacement[,index]*



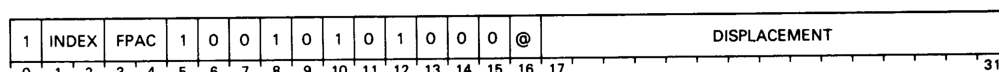
Stores the contents of a specified FPAC into a memory location.

Computes the effective address, *E*. Places the floating point number contained in FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location. The contents of FPAC and the condition codes in the FPSR remain unchanged.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Store Floating Point Single

FSTS *fpac,[@]displacement[,index]*



Stores the contents of a specified FPAC into a memory location.

Computes the effective address *E*. Places the 32 high-order bits of FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location. The contents of FPAC and the condition codes in the FPSR remain unchanged.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Trap Disable
FTD

1	1	0	0	1	1	1	0	1	1	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sets the trap enable (*TE*) bit of the FPSR to 0.

NOTE: *The I/O RESET instruction will also set this bit to 0.*

Trap Enable
FTE

1	1	0	0	0	1	1	0	1	1	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Sets the trap enable *TE* bit of the FPSR to 1. If *ANY* is 1 before execution of this instruction, signals a floating point trap. If *ANY* is 0 before execution of this instruction, execution continues normally at the end of this instruction.

NOTES: *When this instruction is used to cause a floating point trap, the FPPC portion of the FPSR will contain the address of the first instruction to cause a fault. Even if another instruction causes a second fault that occurs before the* FTE *instruction executes, the FPPC will still contain the address of the first instruction that caused a fault.*

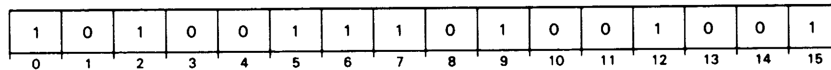
When a floating point fault occurs and TE *is 1, the processor sets* TE *to 0 before transferring control to the floating point error handler.* TE *should be set to 1 before resuming normal processing.*

Fixed Point Trap Disable
FXTD

1	0	1	0	0	1	1	1	0	1	1	1	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

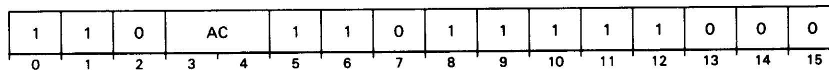
Unconditionally sets the **OVK** flag to zero. This disables fixed point overflow traps. Carry is unchanged.

Fixed Point Trap Enable
FXTE



Unconditionally sets **OVK** to 1 and **OVR** to 0. This enables fixed point overflow traps. Carry is unchanged.

Halve
HLV *ac*



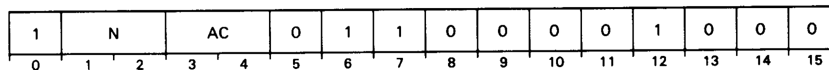
Divides the contents of an accumulator by 2 and rounds the result toward zero.

The signed, 16-bit two's complement number contained in bits 16-31 of the specified accumulator is divided by 2 and rounded toward 0. The result is placed in bits 16-31 of the specified accumulator.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

This instruction leaves carry unchanged; *overflow* is 0.

Hex Shift Left
HXL *n,ac*

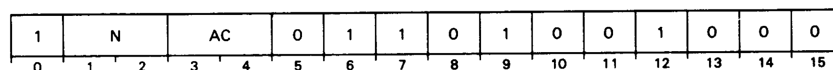


Shifts the contents of bits 16-31 of the specified accumulator left a number of hex digits depending upon the immediate field *N*. The number of digits shifted is equal to $N+1$. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. If *N* is equal to 3, then bits 16-31 of the specified accumulator are shifted out and are set to 0. Leaves carry unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits that he wishes to shift.*

Hex Shift Right
HXR *n,ac*



Shifts the contents of bits 16-31 of the specified accumulator right a number of hex

digits depending upon the immediate field, *N*. The number of digits shifted is equal to *N*+1. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. If *N* is equal to 3, then bits 16-31 of the specified accumulator are shifted out and are set to 0. Leaves carry unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

> **NOTE:** *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact number of hex digits that he wishes to shift.*

Increment

INC*[c][sh][#]* *acs,acd[,skip]*

1	ACS	ACD	0	1	1	SH	C	#	SKIP
0	1-2	3-4	5	6	7	8-9	10-11	12	13-15

Increments the contents of bits 16-31 of an accumulator.

Initializes carry to the specified value. Increments the unsigned, 16-bit number in bits 16-31 of ACS by one and places the result in the shifter. If the incrementation produces a carry of 1 out of the high order bit, the instruction complements carry. Performs the specified shift operation, and loads the result of the shift into bits 16-31 of ACD if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

If the load option is specified, bits 0-15 of ACD are undefined.

> **NOTE:** *If the number in ACS is* 177777_8 *the instruction complements carry.*

For this instruction, *overflow* is 0.

Inclusive OR

IOR *acs,acd*

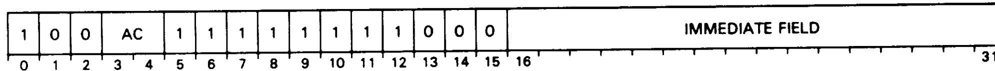
1	ACS	ACD	0	0	1	0	0	0	0	1	0	0	0
0	1-2	3-4	5	6	7	8	9	10	11	12	13	14	15

Forms the logical inclusive OR of the contents of bits 16-31 of ACS and the contents of bits 16-31 of ACD and places the result in bits 16-31 of ACD. Sets a bit position in the result to 1 if the corresponding bit position in one or both operands contains a 1; otherwise, the instruction sets the result bit to 0. The contents of ACS remain unchanged. Carry remains unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Inclusive OR Immediate

IORI *i,ac*

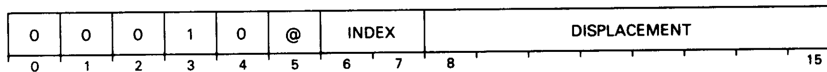


Forms the logical inclusive OR of the contents of the immediate field and the contents of bits 16-31 of the specified accumulator and places the result in bits 16-31 of the specified accumulator. Carry remains unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Increment And Skip If Zero

ISZ *[@]displacement[,index]*



Increments the addressed word, then skips if the incremented value is zero.

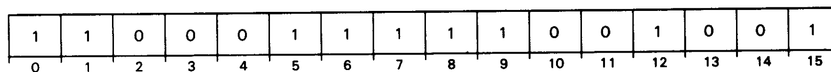
Increments the word addressed by *E* and writes the result back into memory at that location. If the updated value of the location is zero, the instruction skips the next sequential word.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Carry remains unchanged and *overflow* is 0.

Increment the Word Addressed by WSP and Skip if Zero

ISZTS

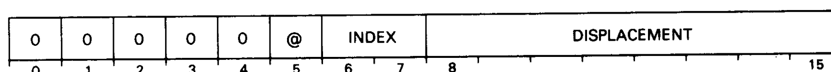


Uses the contents of WSP (the wide stack pointer) as the address of a double word. Increments the contents of the word addressed by WSP. If the incremented value is equal to zero, then the next sequential word is skipped. Carry is unchanged and *overflow* is 0.

NOTE: *The operation performed by this instruction is not indivisible.*

Jump

JMP



Computes the effective address, *E*, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program

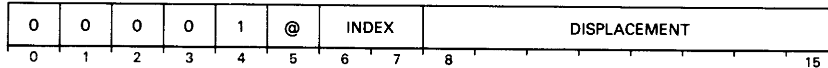
counter.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Carry remains unchanged and *overflow* is 0.

Jump To Subroutine

JSR *[@]displacement[,index]*



Increments and stores the value of the program counter in AC3, and then places a new address in the program counter.

Computes the effective address, *E*; then places the address of the next sequential instruction in bits 16-31 of AC3. Places *E* in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

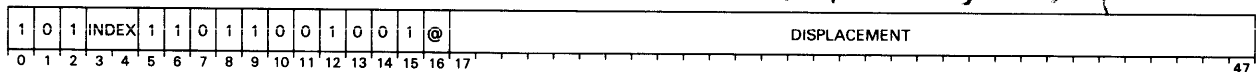
The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

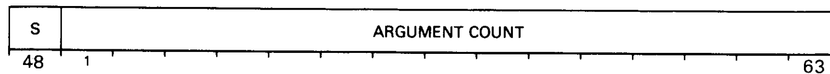
Carry remains unchanged and *overflow* is 0.

NOTE: *The instruction computes* E *before it places the incremented program counter in AC3.*

Call Subroutine (Long Displacement)

LCALL *opcode,argument count,displacement*

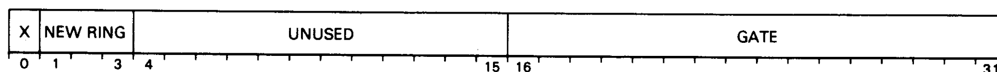




Evaluates the address of a subroutine call.

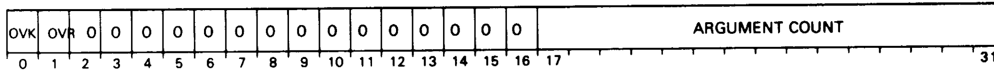
If the target address specifies an outward ring crossing, a protection fault (code=7 in AC1) occurs. Note that the contents of the PC in the return block are undefined.

If the target address specifies an inward ring call, then the instruction assumes the target address has the following format:

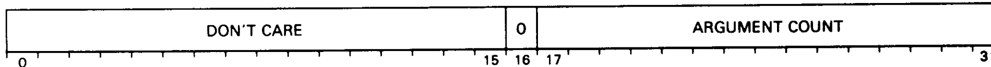


The instruction checks the gate field of the above format for a legal gate. If the specified gate is illegal, a protection fault (code=6 in AC1) occurs and no subroutine call is made. Note that the value of the PC in the return block is undefined.

If the specified gate is legal, or if the target address specifies an intra ring crossing, the instruction loads the contents of the PC, plus four, into AC3. The contents of AC3 always references the current ring. If bit 0 of the argument count is 0, the instruction creates a word with the following format:



The instruction pushes this word onto the wide stack. If a stack overflow occurs after this push, a stack fault occurs and no subroutine call is made. Note that the value of the PC in the return block is undefined. If bit 0 of the argument count is 1, then the instruction assumes the top word of the wide stack has the following format:

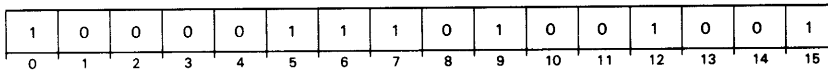


The instruction modifies this word to include the correct settings of **OVK** and **OVR** in bits 0 and 1.

Regardless of the setting of bit 0 of the argument count, the instruction next unconditionally sets **OVR** to 0 and loads the PC with the target address. Control then transfers to the word referenced by the PC.

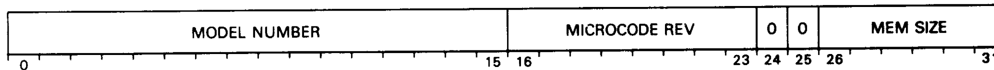
Load CPU Identification

LCPID



Loads a double word into AC0. Carry is unchanged and *overflow* is 0.

The double word has the format:

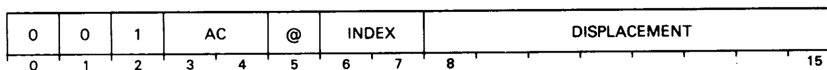


where

model # is the binary representation of the machine's model number,
microcode rev indicates the microcode revision currently in use on this machine,
mem size indicates the amount of physical memory on this machine. A zero in this field indicates 256 Kbytes of memory; a one indicates 512 Kbytes, and so on.

Load Accumulator

LDA *ac,[@]displacement[,index]*



Copies a word from memory to an accumulator.

Places the word addressed by the effective address, *E*, in bits 16-31 of the specified accumulator.

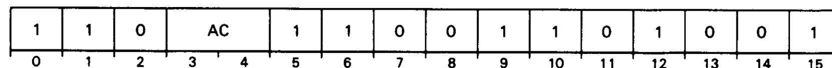
Bits 0-14 are undefined.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

The previous contents of the location addressed by *E* and carry remain unchanged. *Overflow* is 0.

Load Accumulator with WFP

LDAFP *ac*

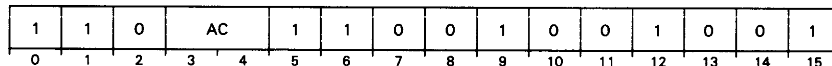


1	1	0	AC	1	1	0	0	1	1	0	1	0	0	1
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15

Loads the 32-bit contents of WFP (the wide frame pointer) into the specified 32-bit accumulator. Carry is unchanged and *overflow* is 0.

Load Accumulator with WSB

LDASB *ac*

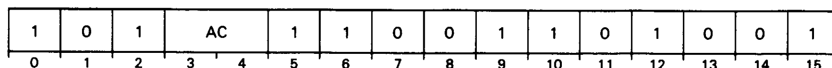


1	1	0	AC	1	1	0	0	1	0	0	1	0	0	1
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15

Loads the 32-bit contents of WSB (the wide stack base) into the specified 32-bit accumulator. Carry is unchanged and *overflow* is 0.

Load Accumulator with WSL

LDASL *ac*

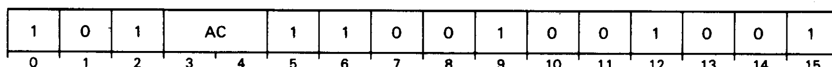


1	0	1	AC	1	1	0	0	1	1	0	1	0	0	1
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15

Loads the 32-bit contents of WSL (the wide stack limit) into the specified 32-bit accumulator. Carry is unchanged and *overflow* is 0.

Load Accumulator with WSP

LDASP *ac*

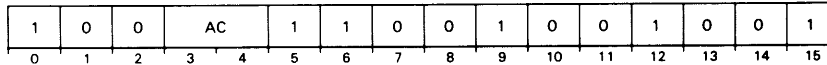


1	0	1	AC	1	1	0	0	1	0	0	1	0	0	1
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15

Loads the contents of WSP (the wide stack pointer) into the specified accumulator. Carry is unchanged and *overflow* is 0.

Load Accumulator with Double Word

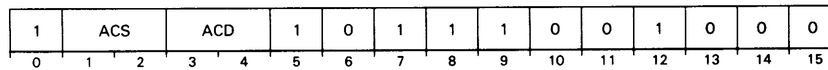
LDATS *ac*



Uses the contents of WSP (the wide stack pointer) as the address of a double word. Loads the contents of the addressed double word into the specified accumulator. Carry is unchanged and *overflow* is 0.

Load Byte

LDB *acs,acd*



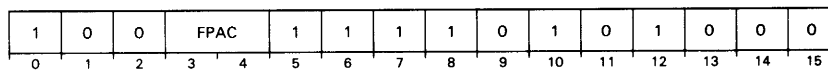
Moves a copy of the contents of a memory byte (as addressed by a byte pointer in one accumulator) into the second accumulator.

Places the 8-bit byte addressed by the byte pointer contained in bits 15-31 of ACS in bits 24–31 of ACD. Sets bits 16–23 of ACD to 0. The contents of ACS remain unchanged unless ACS and ACD are the same accumulator. Carry remains unchanged and *overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

Load Integer

LDI *fpac*



Translates a decimal integer from memory to (normalized) floating point format and places the result in a floating point accumulator.

Under the control of accumulators AC1 and AC3, converts a decimal integer to floating point form, normalizes it, and places it in the specified FPAC. The instruction updates the *Z* and *N* bits in the FPSR to describe the new contents of the specified FPAC. Leaves the decimal number unchanged in memory, and destroys the previous contents of the specified FPAC.

Bits 16-31 of AC1 must contain the data-type indicator describing the number.

Bits 16-31 of AC3 must contain a byte pointer which is the address of the high-order byte of the number in memory.

Numbers of data type 7 are not normalized after loading. By convention, the first byte of a number stored according to data type 7 must contain the sign and exponent of the floating point number. The exponent must be in "excess 64" representation. The instruction copies each byte (following the lead byte) directly to mantissa of the specified FPAC. It then sets to zero each low-order byte in the FPAC that does not receive data

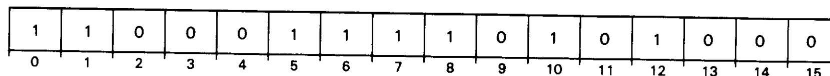
from memory.

Upon successful completion, the instruction leaves accumulators AC0 and AC1 unchanged. AC2 contains the original contents of AC3; the contents of AC3 are undefined. Carry remains unchanged and *overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

NOTE: *An attempt to load a minus 0 sets the specified FPAC to true zero.*

Load Integer Extended
LDIX



Distributes a decimal integer of data type 0, 1, 2, 3, 4, or 5 into the four FPACs.

Extends the integer with high-order zeros until it is 32 digits long. Divides the integer into four units of 8 digits each and converts each unit to a floating point number. Places the number obtained from the 8 high-order digits into FAC0, the number obtained from the next 8 digits into FAC1, the number obtained from the next 8 digits into FAC2, and the number obtained from the low-order 8 bits into FAC3. The instruction places the sign of the integer in each FPAC unless that FPAC has received 8 digits of zeros, in which case the instruction sets FPAC to true zero. The Z and N flags in the floating point status register are unpredictable.

Bits 16-31 of AC1 must contain the data-type indicator describing the integer.

Bits 16-31 of AC3 must contain a byte pointer which is the address of the high-order byte of the integer.

Upon successful termination, the contents of AC0 and AC3 are undefined; the contents of AC1 remain unchanged; and AC2 contains the original contents of AC3. Carry remains unchanged and *overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

Dispatch (Long Displacement)
LDSP *ac,index,displacement*

Dispatches through a table of 28-bit self-relative addresses indexed by the 31-bit PC.

Computes the effective address E . This is the address of a *dispatch table*. The dispatch table consists of a table of 28-bit self-relative addresses (bits 0-3 are ignored). Immediately before the table are two signed, two's complement limit words, L and H. The last word of the table is in location $E + 2 \times (H - L)$. The instruction adds the 28-bit self-relative address in the table entry to the address of the table entry. The ring field of

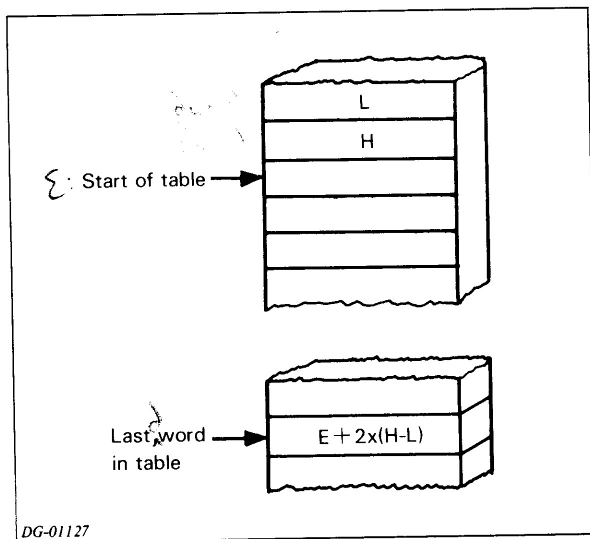
the fetched table entry is ignored.

Compares the signed, two's complement number contained in the accumulator to the limit words. If the number in the accumulator is less than L or greater than H, sequential operation continues with the instruction immediately after the *Wide Dispatch* instruction.

If the number in AC is greater than or equal to L and less than or equal to H, the instruction fetches the word at location E - 2 x (L - *number*). If the fetched word is equal to 37777777777_8 (all 1's), sequential operation continues with the instruction immediately after the *Wide Dispatch* instruction. If the fetched word is not equal to 37777777777_8, the instruction treats this word as the intermediate address in the effective address calculation. After the indirection chain, if any, has been followed, the instruction places the effective address in the program counter and sequential operation continues with the word addressed by the updated value of the program counter. Carry is unchanged and *overflow* is 0.

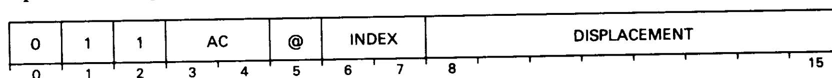
Wraparound occurs within the 28-bit offset. A ring crossing cannot occur. The effective address, E, references a table of self-relative addresses in the current segment. Thus, bits 1–3 of E and bits 1–3 of any levels of indirection are always interpreted as the current segment.

The structure of the dispatch table is shown in the figure below.



Load Effective Address

LEF *ac,[@]displacement[,index]*



Computes the effective address, E, within the current segment and places it in the specified accumulator. Sets bit 0 of the accumulator to 0. The previous contents of the AC are lost.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

> **NOTES:** *The* LEF *instruction can only be used in a mapped system while in the user mode. With the* LEF *mode bit set to 1, all I/O and* LEF *instructions will be interpreted as* LEF *instructions. With the* LEF *mode bit set to 0, all I/O and* LEF *instructions will be interpreted as I/O instructions.*
>
> *Be sure that I/O protection is enabled or the* LEF *mode bit is set to 1 before using the* LEF *instruction. If you issue a* LEF *instruction in the I/O mode, with protection disabled, the instruction will be interpreted and executed as an I/O instruction, with possibly undesirable results.*

Carry is unchanged and *overflow* is 0.

Add Double (Memory to FPAC) (Long Displacement)

LFAMD *fpac,[@]displacement[,index]*

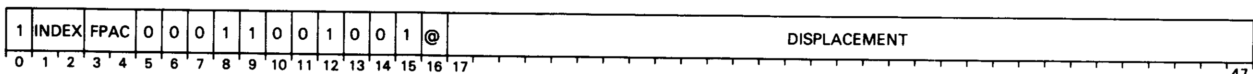
Adds the 64-bit floating point number in the source location to the 64-bit floating point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Adds this 64-bit floating point number to the floating point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Add Single (Memory to FPAC) (Long Displacement)

LFAMS *fpac,[@]displacement[,index]*



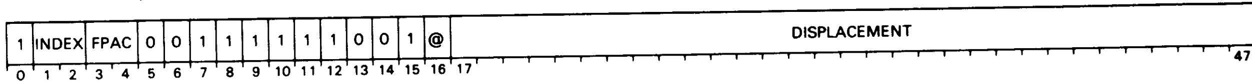
Adds the 32-bit floating point number in the source location to the 32-bit floating point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Adds this 32-bit floating point number to the floating point number in bits 0-31 of the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Divide Double (FPAC by Memory) (Long Displacement)

LFDMD *fpac,[@]displacement[,index]*



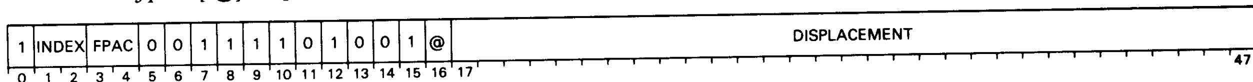
Divides the 64-bit floating point number in FPAC by the 64-bit floating point number in the source location and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Divides the floating point number in the specified FPAC by this 64-bit floating point number. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Divide Single (FPAC by Memory) (Long Displacement)

LFDMS *fpac,[@]displacement[,index]*



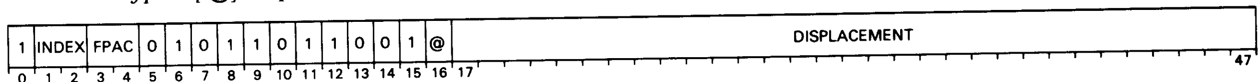
Divides the 32-bit floating point number in bits 32-63 of FPAC by the 32-bit floating point number in the source location and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Divides the floating point number in bits 0-31 of the specified FPAC by this 32-bit floating point number. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Load Floating Point Double (Long Displacement)

LFLDD *fpac,[@]displacement[,index]*



Moves four words out of memory and into a specified FPAC.

Computes the effective address, *E*. Fetches the double precision floating point number at the address specified by *E* and places it in FPAC. Updates the *Z* and *N* flags in the FPSR to reflect the new contents of FPAC.

NOTE: *This instruction will move unnormalized data without change.*

Floating Point Load Single (Long Displacement)

LFLDS *fpac,[@]displacement[,index]*

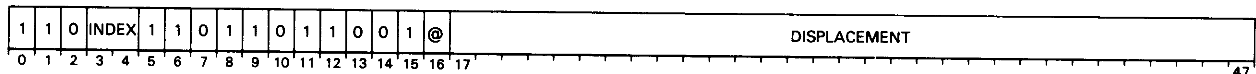
Moves two words out of memory into a specified FPAC.

Computes the effective address *E*. Fetches the single precision floating point number at the address specified by *E*. Places the number in the high-order bits of FPAC. Sets the low-order 32 bits of FPAC to 0. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

> **NOTE:** *This instruction will move unnormalized or illegal data without change, but the* Z *and* N *flags will be undefined.*

Load Floating Point Status (Long Displacement)

LFLST *[@]displacement[,index]*



Moves the contents of two specified memory locations to the floating point status register.

Computes the effective address, *E*. Places the 32-bit operand addressed by *E* in the floating point status register as follows:

- Places bits 0-15 of the operand in bits 0-15 of the FPSR. Sets bits 16-32 of the FPSR to 0.
- If *ANY* is 0, bits 33-63 of the FPSR are undefined.
- If *ANY* is 1, the instruction places the value of the current segment in bits 33-35 of the FPSR, zeroes in bits 36-48, and bits 17-31 of the operand in bits 49-63 of the FPSR.

> **NOTES:** *This instruction does not set the* ANY *flag from memory. If any of bits 1-4 are loaded as 1,* ANY *is set to 1; otherwise,* ANY *is 0.*
>
> *Bits 12-15 of the FPSR are not set from memory. These bits are the floating point identification code an are read protected. In the MV/8000 they are set to 0111.*
>
> *This instruction initiates a floating point trap if* ANY *and* TE *are both 1 after the FPPC is loaded.*

See Chapter 8 and Appendix G for more information about floating point manipulation.

Multiply Double (FPAC by Memory) (Long Displacement)

LFMMD *fpac,[@]displacement[,index]*

Multiplies the 64-bit floating point number in the source location by the 64-bit floating

point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Multiplies this 64-bit floating point number by the floating point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Multiply Single (FPAC by Memory) (Long Displacement)

LFMMS *fpac,[@]displacement[,index]*

Multiplies the 32-bit floating point number in the source location by the 32-bit floating point number in bits 0-31 of FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Multiplies this 32-bit floating point number by the floating point number in bits 0-31 of the specified FPAC. Places the normalized result in bits 0-31 of the specified FPAC. Sets bits 32-63 of FPAC to 0. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Subtract Double (Memory from FPAC) (Long Displacement)

LFSMD *fpac,[@]displacement[,index]*

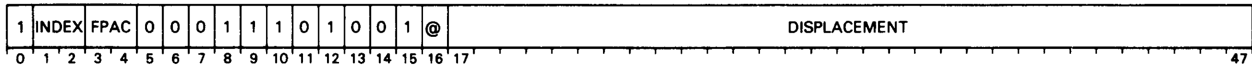
Subtracts the 64-bit floating point number in the source location from the 64-bit floating point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Subtracts this 64-bit floating point number from the floating point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Subtract Single (Memory from FPAC) (Long Displacement)

LFSMS *fpac,[@]displacement[,index]*



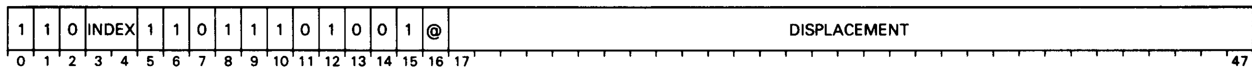
Subtracts the 32-bit floating point number in the source location from the 32-bit floating point number in bits 0-31 of FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Subtracts this 32-bit floating point number from the floating point number in bits 0-31 of the specified FPAC. Places the normalized result in the specified FPAC. Sets bits 32-63 of FPAC to 0. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Store Floating Point Status (Long Displacement)

LFSST *[@]displacement[,index]*



Moves the contents of the FPSR to four specified memory locations.

Computes the effective address, *E*, of two sequential, 32-bit locations in memory. Stores the contents of the FPSR in these locations as follows:

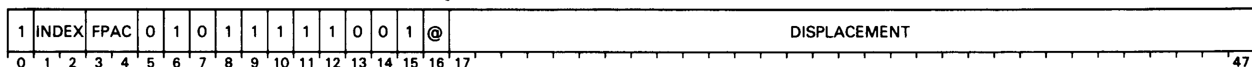
- Stores bits 0-15 of the FPSR in the first memory word.
- Sets bits 16-31 of the first memory double word and bit 0 of the second memory double word to 0.
- If *ANY* is 0, the contents of bits 1-31 of the second memory double word are undefined.
- If *ANY* is 1, the instruction stores bits 33-63 of the FPSR into bits 1-31 of the second memory double word.

NOTE: *This instruction does not initiate a floating point trap under any conditions of the FPSR.*

See Chapter 8 and Appendix G for more information about floating point manipulation.

Store Floating Point Double (Long Displacement)

LFSTD *fpac,[@]displacement[,index]*



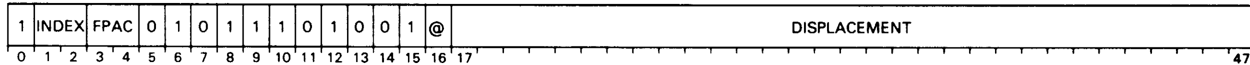
Stores the contents of a specified FPAC into a memory location.

Computes the effective address, *E*. Places the floating point number contained in FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location. The contents of FPAC and the condition codes in the FPSR remain unchanged.

NOTE: *This instruction moves unnormalized or illegal data without change.*

Store Floating Point Single (Long Displacement)

LFSTS *fpac,[@]displacement[,index]*



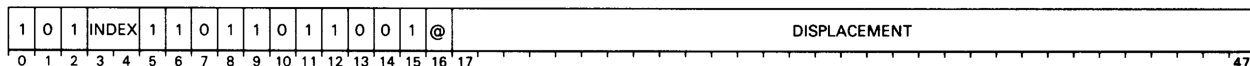
Stores the contents of a specified FPAC into a memory location.

Computes the effective address *E*. Places the 32 high-order bits of FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location. The contents of FPAC and the condition codes in the FPSR remain unchanged.

NOTE: *This instruction moves unnormalized or illegal data without change.*

Jump (Long Displacement)

LJMP *index,displacement*

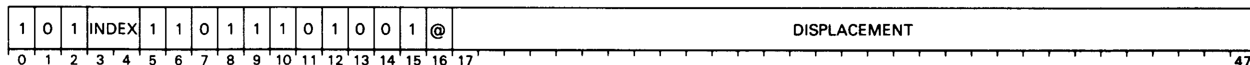


Calculates the effective address *E*. Loads *E* into the PC. Carry is unchanged and *overflow* is 0.

NOTE: *The calculation of* E *is forced to remain within the current segment of execution.*

Jump to Subroutine (Long Displacement)

LJSR *index,displacement*



Loads AC3 with the current 31-bit value of the program counter plus three. Loads the PC with the effective address. Carry is unchanged and *overflow* is 0.

NOTE: *The calculation of* E *is forced to remain within the current segment of execution.*

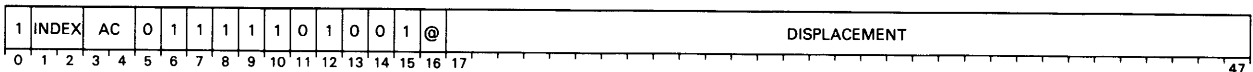
Load Byte (Long Displacement)

LLDB *ac,index,displacement*

Calculates the effective byte address. Uses the byte address to reference a byte in memory. Loads the addressed byte into bits 24–31 of the specified accumulator, then zero extends the value to 32 bits. Carry is unchanged and *overflow* is 0.

Load Effective Address (Long Displacement)

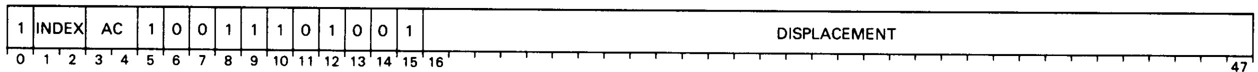
LLEF *ac,index,displacement*



Calculates the effective address, *E*. Checks for segment crossing violation. If no violation occurs, loads *E* into the specified accumulator. If a violation occurs, issues a protection fault. Carry is unchanged and *overflow* is 0.

Load Effective Byte Address (Long Displacement)

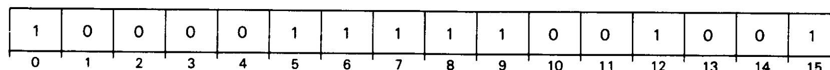
LLEFB *ac,index,displacement*



Calculates a byte address. Checks for segment crossing violation. If no violation occurs, loads the byte address into the specified accumulator. If a violation occurs, issues a protection fault. Carry is unchanged and *overflow* is 0.

Load Modified and Referenced Bits

LMRF



Loads the modified and referenced bits of a pageframe into AC0.

AC1 contains a pageframe number in bits 13–31.

The instruction loads the modified and referenced bits of the pageframe specified by AC1 into AC0. The bits are loaded right justified and zero filled. The instruction then resets the referenced bit just accessed to 0. Carry is unchanged and *overflow* is 0.

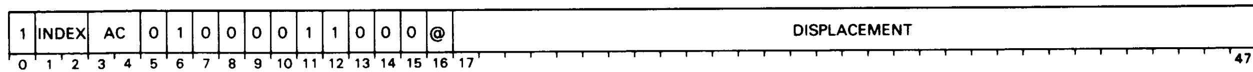
If the ATU is not enabled, undefined results will occur.

Specification of a non-existant pageframe results in an indeterminate data.

NOTE: *This is a privileged instruction.*

Narrow Add Memory Word to Accumulator (Long Displacement)

LNADD *ac,index,displacement*

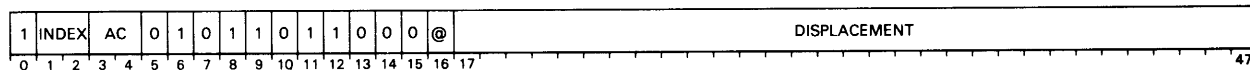


Adds an integer in memory to an integer contained in an accumulator.

The instruction calculates the effective address, *E*. Adds the 16-bit integer contained in the location specified by *E* to the integer contained in bits 16–31 of the specified accumulator. Sign extends the 16-bit result to 32 bits and loads it into the specified accumulator. Sets carry to the value of ALU carry, and *overflow* to 1 if there is an ALU overflow. The contents of the referenced memory location remain unchanged.

Narrow Divide Memory Word (Long Displacement)

LNDIV *ac,index,displacement*



Divides an integer contained in an accumulator by an integer in memory.

The instruction calculates the effective address, *E*. Sign extends the integer contained in bits 16–31 of the specified accumulator to 32 bits and divides it by the 16-bit integer contained in the location specified by *E*. If the quotient is within the range -32,768 to +32,767 inclusive, sign extends the result to 32 bits and loads it into the specified accumulator. If the quotient is outside of this range, or the memory word is zero, the instruction sets *overflow* to 1 and leaves the specified accumulator unchanged. Otherwise, *overflow* is 0. The contents of the referenced memory location and carry remain unchanged.

Narrow Decrement and Skip if Zero (Long Displacement)

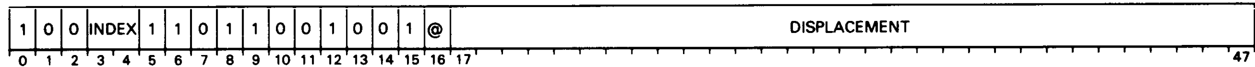
LNDSZ *index,displacement*

Calculates the effective address *E*. Decrements by one the contents of the 16-bit memory location addressed by *E*. If the result is equal to zero, then the next sequential word is skipped. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction is indivisible.*

Narrow Increment and Skip if Zero (Long Displacement)

LNISZ *index,displacement*

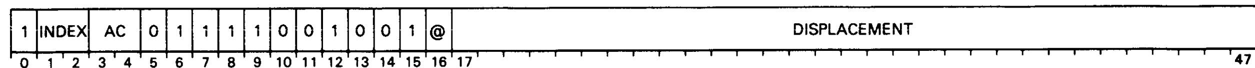


Calculates the effective address, *E*. Increments by one the contents of the 16-bit memory location addressed by *E*. If the result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction is indivisible.*

Narrow Load Accumulator (Long Displacement)

LNLDA *ac,index,displacement*



Calculates the effective address, *E*. Fetches the 16-bit fixed point integer contained in the location specified by *E*. Sign extends this integer to 32 bits and loads it into the specified accumulator. Carry is unchanged and *overflow* is 0.

Narrow Multiply Memory Word (Long Displacement)

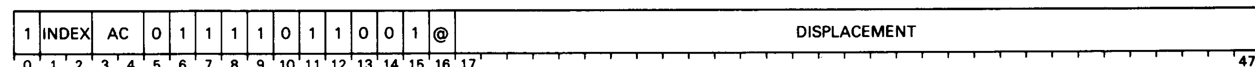
LNMUL *ac,index,displacement*

Multiplies an integer in memory by an integer in an accumulator.

Calculates the effective address, *E*. Multiplies the 16-bit, signed integer contained in the location referenced by *E* by the signed integer contained in bits 16–31 of the specified accumulator. If the result is outside the range of -32,768 to +32,767 inclusive, sets *overflow* to 1; otherwise, *overflow* is 0. Sign extends the result to 32 bits and places the result in the specified accumulator. The contents of the referenced memory location and carry remain unchanged.

Narrow Store Accumulator (Long Displacement)

LNSTA *ac,index,displacement*



Computes the effective address, *E*. Stores the low-order 16 bits of the specified accumulator into the location specified by *E*. Carry is unchanged and *overflow* is 0.

Narrow Subtract Memory Word (Long Displacement)

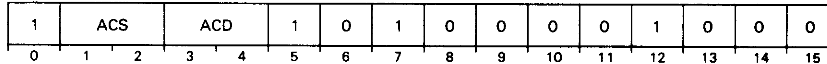
LNSUB *ac,index,displacement*

Subtracts an integer in memory from an integer in an accumulator.

Calculates the effective address, *E*. Subtracts the 16-bit integer contained in the location referenced by *E* from the integer contained in bits 16–31 of the specified accumulator. Sign extends the result to 32 bits and stores it in the specified accumulator. Sets carry to the value of ALU carry, and *overflow* to 1 if there is an ALU overflow. The contents of the specified memory location remain unchanged.

Locate Lead Bit

LOB *acs,acd*



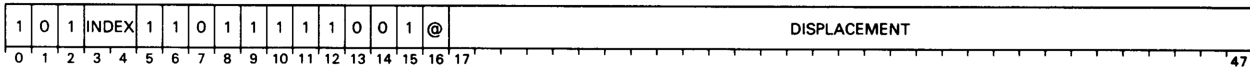
Adds a number equal to the number of high-order zeroes in the contents of bits 16-31 of ACS to the signed, 16-bit, two's complement number contained in bits 16-31 of ACD. The contents of ACS and the state of carry remain unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

> NOTE: *If ACS and ACD are specified as the same accumulator, the instruction functions as described above, except that since ACS and ACD are the same accumulator, the contents of ACS will be changed.*

Push Address (Long Displacement)

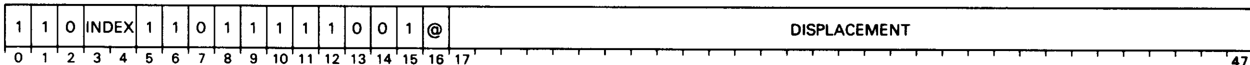
LPEF *index, displacement*



Pushes an effective address onto the wide stack. Carry is unchanged and *overflow* is 0.

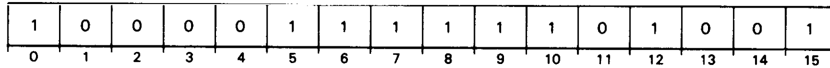
Push Byte Address (Long Displacement)

LPEFB *index, displacement*



Calculates an effective byte address. Pushes this byte address onto the wide stack. Carry is unchanged and *overflow* is 0.

Load Physical
LPHY



Translates the logical address contained in AC1 to a physical address.

AC1 contains a logical word address.

If the ATU is disabled, this instruction does nothing. The next word is executed.

If the ATU is enabled, then the actions described below occur.

The instruction compares the ring field of AC1 to the current ring. If AC1's ring field is less than or equal to the current ring field, then a protection fault (AC1 = 4) occurs.

If AC1's ring field is greater than the current ring, then the instruction references the SBR specified by AC1. If the SBR contents are invalid, then the instruction ends and the next instruction is executed. The contents of AC0 will be unchanged.

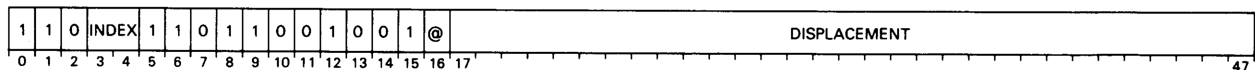
If the contents of the SBR are valid, the instruction loads AC0 with the last resident PTE. If the PTE indicates no page or validity faults, the instruction loads AC2 with the 32-bit physical word address of the logical address contained in AC1. The next sequential word is skipped.

If the PTE signals a page or validity fault, the contents of AC2 remain unchanged. The next sequential word is executed.

The instruction leaves carry unchanged; *overflow* is 0.

Push Jump (Long Displacement)

LPSHJ *index,displacement*



Saves a return address on the wide stack and jumps to a specified location.

Pushes the current 31-bit value of the program counter + 3 onto the wide stack. Calculates the effective address, *E*. Loads the PC with *E*. Sequential operation continues with the word addressed by the updated value of the program counter. Carry is unchanged and *overflow* is 0.

> **NOTE:** *The value pushed onto the wide stack will always point to a location in the current ring.*

Load Processor Status Register into AC0

LPSR

1	0	1	0	0	1	1	1	1	0	0	1	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Loads the contents of the PSR into AC0.

Loads the contents of **OVK, OVR,** and **IRES** into bits 0, 1, and 2 of AC0, respectively. Fills the rest of AC0 with zeroes. The contents of the PSR remain unchanged. Carry is unchanged and *overflow* is 0.

Locate and Reset Lead Bit

LRB *acs,acd*

1	ACS		ACD		1	0	1	0	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Performs a *Locate Lead Bit* instruction, and sets the lead bit to 0.

Adds a number equal to the number of high-order zeroes in the contents of bits 16-31 of ACS to the signed, 16-bit, two's complement number contained in bits 16-31 of ACD. Sets the leading 1 in bits 16-31 of ACS to 0. Carry remains unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *If ACS and ACD are specified to be the same accumulator, then the instruction sets the leading 1 in that accumulator to 0, and no count is taken.*

Load All Segment Base Registers

LSBRA

1	1	0	0	0	1	1	1	1	0	1	1	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Loads the SBRs with new values.

AC0 contains the starting address of an 8 double word block.

The instruction loads a copy of the contents of these words into the SBRs as shown in the table below:

Double Word in Block	Destination	Order Moved
1	SBR0	First
2	SBR1	Second
3	SBR2	Third
4	SBR3	Fourth
5	SBR4	Fifth
6	SBR5	Sixth
7	SBR6	Seventh
8	SBR7	Eighth

After loading the SBRs, the instruction purges the ATU. If the ATU was disabled at the beginning of this instruction cycle, the processor enables it now.

If an invalid address is loaded into SBR0, the processor disables the ATU and a protection fault occurs (code = 3 in AC1). This means that logical addresses are identical to physical addresses, and the fault is processed in physical address space.

The instruction leaves AC0 and carry unchanged; *overflow* is 0.

NOTE: *This is a privileged instruction.*

Load Segment Base Registers 1-7
LSBRS

1	1	1	0	0	1	1	1	1	0	0	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Loads SBR1 through SBR7 with new values.

AC0 contains the starting address of a block of seven double words. The instruction loads a copy of the contents of these words into the SBRs as shown in the table below:

Double Word in Block	Destination	Order Moved
1	SBR1	First
2	SBR2	Second
3	SBR3	Third
4	SBR4	Fourth
5	SBR5	Fifth
6	SBR6	Sixth
7	SBR7	Seventh

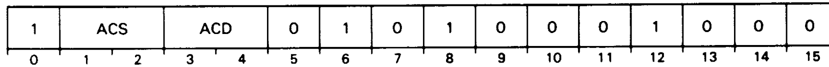
After loading the SBRs, the instruction purges the ATU. If the ATU was disabled at the beginning of this instruction cycle, the processor enables it now.

If SBR0 contains invalid information, then the processor disables the ATU and a protection fault occurs (code = 3 in AC1). This means that logical addresses are identical to physical addresses, and the fault is processed in physical address space.

The instruction leaves AC0 and carry unchanged; *overflow* is 0.

NOTE: *This is a privileged instruction.*

Logical Shift
LSH *acs,acd*



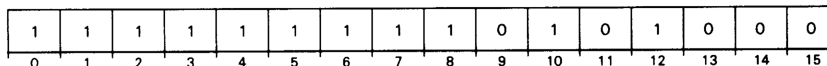
Shifts the contents of bits 16-31 of ACD either left or right depending on the number contained in bits 24–31 of ACS. The signed, 8-bit two's complement number contained in bits 24–31 of ACS determines the direction of the shift and the number of bits to be shifted. If the number in bits 24–31 of ACS is positive, shifting is to the left; if the number in bits 24–31 of ACS is negative, shifting is to the right. If the number in bits 24–31 of ACS is zero, no shifting is performed. Bits 16–23 of ACS are ignored.

The number of bits shifted is equal to the magnitude of the number in bits 24–31 of ACS. Bits shifted out are lost, and the vacated bit positions are filled with zeroes. Carry and the contents of ACS remain unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

> NOTE: *If the magnitude of the number in bits 24–31 of ACS is greater than 15, all bits of ACD are set to 0. Carry and the contents of ACS remain unchanged.*

Load Sign
LSN



Under the control of accumulators AC1 and AC3, evaluates a decimal number in memory and returns in AC1 a code that classifies the number as zero or nonzero and identifies its sign. The meaning of the returned code is as follows:

Value of Number	Code
Positive non-zero	+1
Negative non-zero	-1
Positive zero	0
Negative zero	-2

Bits 16-31 of AC1 must contain the data type indicator describing the number.

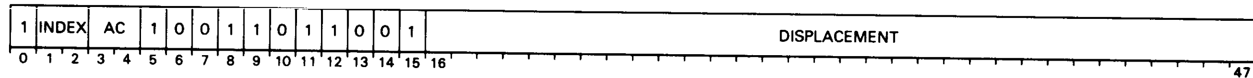
Bits 16-31 of AC3 must contain a byte pointer which is the address of the high-order byte of the number.

Upon successful termination, the contents of AC0 remain unchanged; AC1 contains the value code; AC2 contains the original contents of AC3; and the contents of AC3 are unpredictable. Carry remains unchanged. The contents of the addressed memory locations remain unchanged. *Overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

Store Byte (Long Displacement)

LSTB *ac,index,displacement*

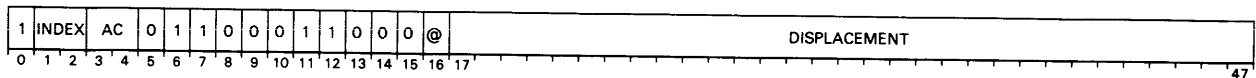


Stores the low-order byte of the specified accumulator in memory.

Calculates the effective byte address. Moves a copy of the contents of bits 24–31 of the specified accumulator into memory at the location specified by the byte address. Carry is unchanged and *overflow* is 0.

Wide Add Memory Word to Accumulator (Long Displacement)

LWADD *ac,index,displacement*

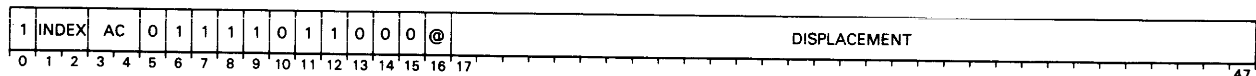


Adds an integer in memory to an integer in an accumulator.

The instruction calculates the effective address, *E*. Adds the 32-bit integer contained in the location specified by *E* to the 32-bit integer contained in the specified accumulator. Loads the result into the specified accumulator. Sets carry to the value of ALU carry, and *overflow* to 1 if there is an ALU overflow. The contents of the referenced memory location remain unchanged.

Wide Divide From Memory (Long Displacement)

LWDIV *ac,index,displacement*



Divides an integer in an accumulator by an integer in memory.

The instruction calculates the effective address, *E*. Sign extends the 32-bit integer contained in the specified accumulator to 64 bits and divides it by the 32-bit integer contained in the location specified by *E*.

If the quotient is within the range of -2,147,483,648 to +2,147,483,647 inclusive, the instruction loads the quotient into the specified accumulator. *Overflow* is 0.

If the quotient is outside this range, or if the word in memory is zero, the instruction sets *overflow* to 1 and leaves the specified accumulator unchanged.

The contents of the referenced memory location and carry remain unchanged.

Wide Decrement and Skip if Zero (Long Displacement)

LWDSZ *index,displacement*

Decrements the contents of a 32-bit memory location by one. If the result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction executes in one indivisible memory cycle if the instruction is located on a double word boundary.*

Wide Increment and Skip if Zero (Long Displacement)

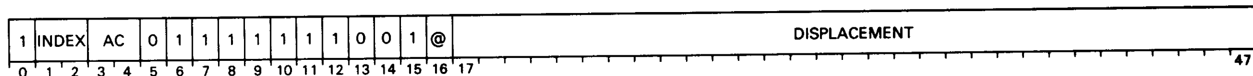
LWISZ *index,displacement*

Increments the contents of a 32-bit memory location by one. If the result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction executes in one indivisible memory cycle if the instruction is located on a double word boundary.*

Wide Load Accumulator (Long Displacement)

LWLDA *ac,index,displacement*



Loads the contents of a memory location into an accumulator.

Calculates the effective address, *E*. Fetches the 32-bit fixed point integer contained in the location specified by *E*. Loads this integer into the specified accumulator. Carry is unchanged and *overflow* is 0.

Wide Multiply From Memory (Long Displacement)

LWMUL *ac,index,displacement*

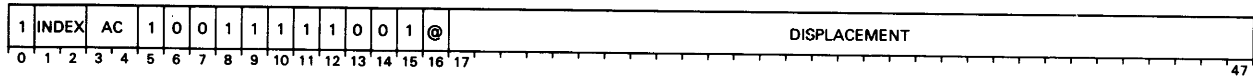
Multiplies an integer in an accumulator by an integer in memory.

The instruction calculates the effective address, *E*. Multiplies the 32-bit, signed integer contained in the location referenced by *E* by the 32-bit, signed integer contained in the specified accumulator. Places the 32 least significant bits of the result in the specified accumulator. The contents of the memory location and carry remain unchanged.

If the result is outside the range of -2,147,483,648 to +2,147,483,647 inclusive, sets *overflow* to 1; otherwise, *overflow* is 1. The specified accumulator will contain the 32 least significant bits of the result.

Wide Store Accumulator (Long Displacement)

LWSTA *ac,index,displacement*



Calculates the effective address, *E*. Stores the 32-bit contents of the specified accumulator in the location specified by *E*. Carry is unchanged and *overflow* is 0.

Wide Subtract Memory Word (Long Displacement)

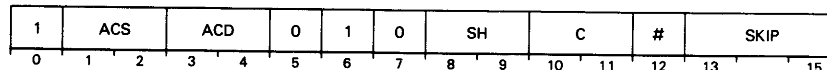
LWSUB *ac,index,displacement*

Subtracts an integer in memory from an integer in an accumulator.

The instruction calculates the effective address, *E*. Subtracts the 32-bit integer contained in the location referenced by *E* from the 32-bit integer contained in the specified accumulator. Loads the result into the specified accumulator. Sets carry to the value of ALU carry, and *overflow* to 1 if there is an ALU overflow. The contents of the specified memory location remain unchanged.

Move

MOV*[c][sh][#]* *acs,acd[,skip]*



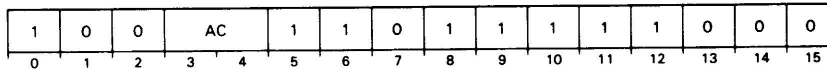
Moves the contents of bits 16-31 of an accumulator into another accumulator.

Initializes carry to the specified value. Places the contents of bits 16-31 of ACS in the shifter. Performs the specified shift operation and loads the result of the shift into bits 16-31 of ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word. *Overflow* is 0.

If the load option is specified, bits 0-15 of ACD are undefined.

Modify Stack Pointer

MSP *ac*



Changes the value of the stack pointer and tests for potential overflow.

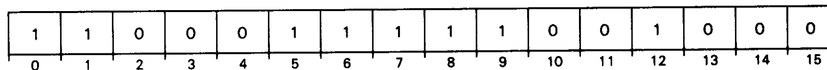
Adds the signed two's-complement number in bits 16-31 of the specified accumulator to the value of the stack pointer and places the result in location 40. The instruction then checks for overflow by comparing the result in location 40 with the value of the stack limit. If the result in location 40 is less than the stack limit, then the instruction ends.

If the result is greater than the stack limit, the instruction changes the value of location 40 back to its original contents before the add. The instruction pushes a standard return block. The program counter in the return block contains the address of the *Modify Stack Pointer* instruction.

After pushing the return block, the program counter contains the address of the stack fault routine. The stack pointer is updated with the value used to push the return block, and control transfers to the stack fault routine. Carry remains unchanged and *overflow* is 0.

Unsigned Multiply

MUL



Multiplies the unsigned contents of two accumulators and adds the result to the unsigned contents of a third accumulator. The result is an unsigned 32-bit integer in two accumulators.

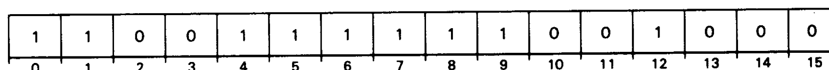
The unsigned, 16-bit number in bits 16-31 of AC1 is multiplied by the unsigned, 16-bit number in bits 16-31 of AC2 to yield an unsigned, 32-bit intermediate result. The unsigned, 16-bit number in bits 16-31 of AC0 is added to the intermediate result to produce the final result. The final result is an unsigned, 32-bit number and occupies bits 16-31 of both AC0 and AC1. Bit 16 of AC0 is the high-order bit of the result and bit 31 of AC1 is the low-order bit. The contents of AC2 remain unchanged.

Because the result is a double-length number, overflow cannot occur. Carry remains unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Signed Multiply

MULS



Multiplies the signed contents of two accumulators and adds the result to the signed

contents of a third accumulator. The result is a signed 32-bit integer in two accumulators.

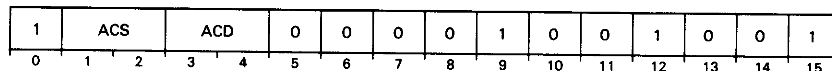
The signed, 16-bit two's complement number in bits 16-31 of AC1 is multiplied by the signed, 16-bit two's complement number in bits 16-31 of AC2 to yield a signed, 32-bit two's complement intermediate result. The signed, 16-bit two's complement number in bits 16-31 of AC0 is added to the intermediate result to produce the final result. The final result is a signed, 32-bit two's complement number which occupies bits 16-31 of both AC0 and AC1. Bit 16 of AC0 is the sign bit of the result and bit 31 of AC1 is the low-order bit. The contents of AC2 remain unchanged.

Because the result is a double-length number, overflow cannot occur. Carry remains unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Narrow Add

NADD *acs,acd*

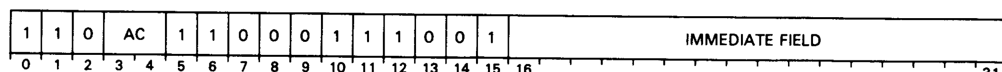


Adds two integers contained in accumulators.

The instruction adds the 16-bit integer contained in bits 16–31 of ACS to the 16-bit integer contained in bits 16–31 of ACD. Stores the result in bits 16–31 of ACD. Sign extends ACD to 32 bits. Sets carry to the value of ALU carry, and sets *overflow* to 1 if there is an ALU overflow.

Narrow Extended Add Immediate

NADDI *n,ac*

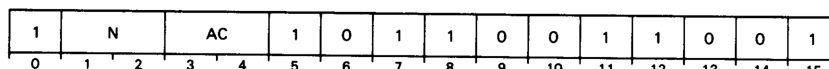


Adds an integer contained in an immediate field to an integer in an accumulator.

Adds the 16-bit value contained in the immediate field to bits 16-31 of the specified accumulator. Stores the result in the lower 16 bits of ACD. Sign extends ACD to 32 bits. Sets carry to ALU carry (16 bit operation). Sets *overflow* to 1 if there is an ALU overflow (16 bit operation).

Narrow Add Immediate

NADI *n,ac*



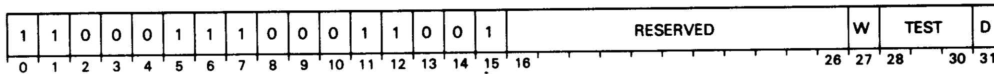
Adds an integer in the range of 1 to 4 to an integer contained in an accumulator.

The instruction adds the value $n+1$ to the 16-bit contents of the specified accumulator, where n is an integer in the range of 0 to 3. Stores the result in the lower 16 bits of the specified accumulator. Sign extends the specified accumulator to 32 bits. Sets carry to the value of ALU carry (16-bit operation). Sets *overflow* to 1 if there is an ALU overflow (16 bit operation).

NOTE: *The assembler takes the coded value of* n *and subtracts 1 from it before placing it in the immediate field. Therefore, the programmer should code the exact value that he wishes to add.*

Search Queue

*<width><direction>*S*<test condition>*



Searches a queue for the first data element containing information that meets a specified condition.

AC1 contains the effective address of the first queue data element to search.

AC3 contains a two's complement word offset. The instruction adds the offset to the address of the forward link of each data element to get the address of the location to test (called the test location).

The double word on the top of the wide stack contains a mask word.

Bits 11–15 of the second word of the search instruction specify the conditions of the search. The table below explains the meanings of these bits.

Bits	Name of Field	Encoding	Mnemonic	Meaning
11	Width	0	N	Search field is 16 bits wide.
		1	W	Search field is 32 bits wide.
12–14	Test	000	SS	Some of the bits specified by the mask in the test condition are one.
		001	SC	Some of the bits specified by the mask in the test condition are zero.
		010	AS	All of the bits specified by the mask in the test condition are one.
		011	AC	All of the bits specified by the mask in the test condition are zero.
		100	E*	The mask and test location are equal.
		101	GE*	The mask is greater than or equal to the test location.
		110	LE*	The mask is less than or equal to the test location.
		111	NE*	The mask and the test location are not equal.
15	Direction	0	F	Search forward in the queue.
		1	B	Search backward in the queue.

NOTE: *The instruction treats the values contained in the mask and the test location as unsigned integers for these test conditions.*

The size of the field to search (bit 11) determines the size of the mask and the size of the offset. If you specify a narrow search, then bits 16–31 of the wide stack word contain the mask. AC3 specifies a relative word offset to the 16-bit test location. If you specify a wide search, then bits 0–31 of the wide stack word contain the mask. AC3 specifies the relative word offset to the 32-bit test location.

The instruction searches each data element in the queue from the element specified by AC1 to the head or tail of the queue (depending on the direction of the search). To perform the search on each element, the instruction adds the offset contained in AC3 to the address contained in AC1 to obtain an address of a location contained in some data element in the queue. Compares the mask field to the value contained in the calculated address.

If the search fails, AC1 contains the effective address of the last data element searched. Execution continues with the next sequential instruction. Interrupts are honored between the time the search fails and the time the next word executes.

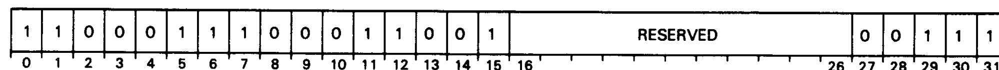
If the search is interrupted, AC1 contains the effective address of the next data element to be examined. The next sequential word is skipped and execution continues with the second word. Interrupts are honored between the time the interrupt occurs and the time the second instruction executes.

If the search is successful, AC1 contains the address of the data element that met the specified condition. The next two sequential words are skipped and execution continues with the third word. Note that interrupts cannot occur between the time the search is successful and the time the third word executes.

For all returns, the contents of carry, WSP, AC0, AC2, and AC3 remain unchanged. *Overflow* is 0.

Narrow Search Queue Backward

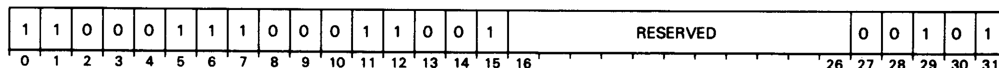
NBSAC



See instruction entry “Search Queue”.

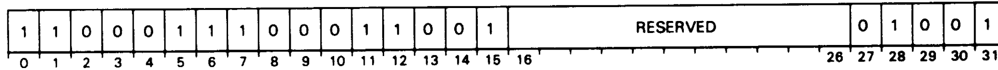
Narrow Search Queue Backward

NBSAS



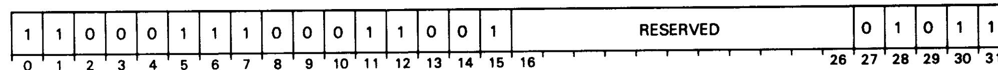
See instruction entry “Search Queue”.

Narrow Search Queue Backward
NBSE



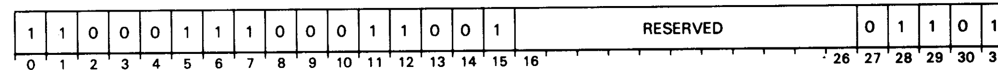
See instruction entry "Search Queue".

Narrow Search Queue Backward
NBSGE



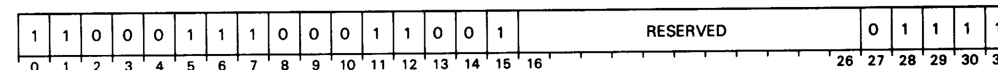
See instruction entry "Search Queue".

Narrow Search Queue Backward
NBSLE



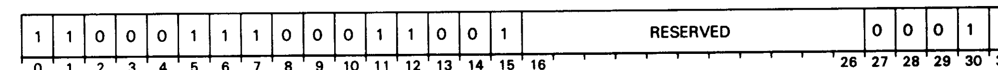
See instruction entry "Search Queue".

Narrow Search Queue Backward
NBSNE



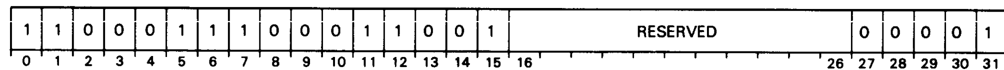
See instruction entry "Search Queue".

Narrow Search Queue Backward
NBSSC



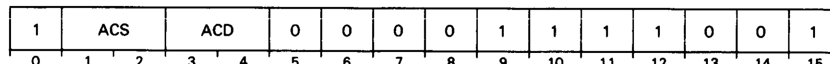
See instruction entry "Search Queue".

Narrow Search Queue Backward
NBSSS



See instruction entry "Search Queue".

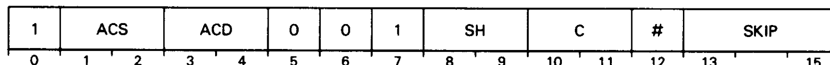
Narrow Divide
NDIV *acs,acd*



Divides an integer in an accumulator by an integer in another accumulator.

The instruction sign extends bits 16–31 of ACD to 32 bits. Divides this signed integer by the 16-bit signed integer contained in bits 16–31 of ACS. If the quotient is within the range -32,768 to +32,767 inclusive, sign extends the lower 16 bits of the result to 32 bits and places these 16 bits in ACD. If the quotient is outside of this range, or if ACS is zero, the instruction sets *overflow* to 1 and leaves ACD unchanged. Otherwise, *overflow* is 0. The contents of ACS and carry always remain unchanged.

Negate
NEG*[c][sh][#]* *acs,acd[,skip]*



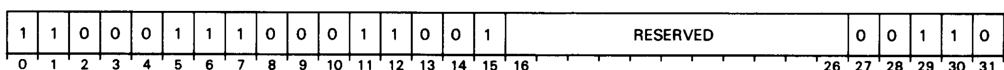
Forms the two's complement of the contents of bits 16-31 of an accumulator.

Initializes carry to the specified value. Places the two's complement of the unsigned, 16-bit number in bits 16-31 of ACS in the shifter. If the negate operation produces a carry of 1 out of the high-order bit, the instruction complements carry. Performs the specified shift operation and places the result in bits 16-31 of ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word. *Overflow* is 0.

If the load option is specified, bits 0-15 of ACD are undefined.

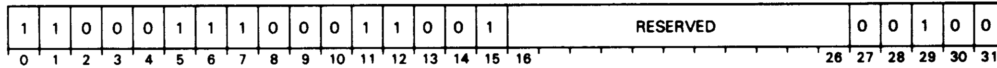
NOTE: *If ACS contains 0, the instruction complements carry.*

Narrow Search Queue Forward
NFSAC



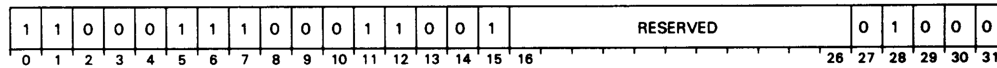
See instruction entry "Search Queue".

Narrow Search Queue Forward
NFSAS



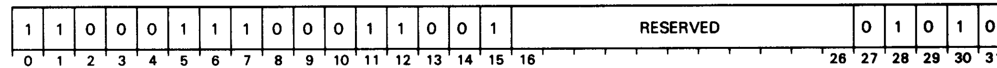
See instruction entry "Search Queue".

Narrow Search Queue Forward
NFSE



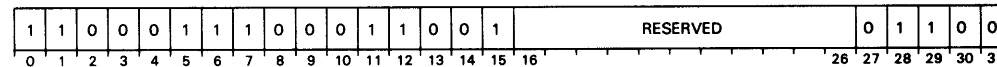
See instruction entry "Search Queue".

Narrow Search Queue Forward
NFSGE



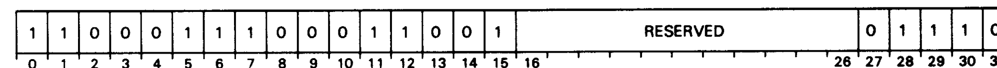
See instruction entry "Search Queue".

Narrow Search Queue Forward
NFSLE



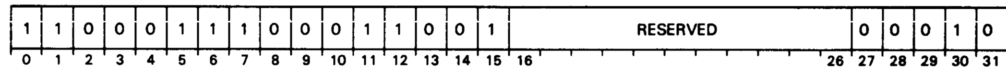
See instruction entry "Search Queue".

Narrow Search Queue Forward
NFSNE



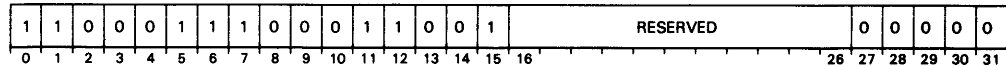
See instruction entry "Search Queue".

Narrow Search Queue Forward
NFSSC



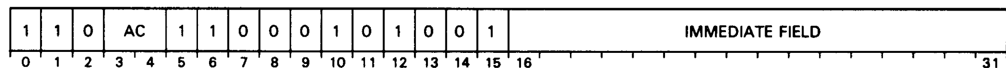
See instruction entry "Search Queue".

Narrow Search Queue Forward
NFSSS



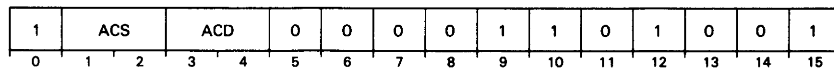
See instruction entry "Search Queue".

Narrow Load Immediate
NLDAI *ac,immediate*



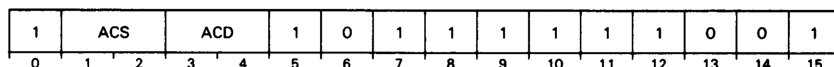
Sign extends the 16-bit, two's complement literal value contained in the immediate field to 32 bits. Loads the result of the sign extension into the specified accumulator. Carry is unchanged and *overflow* is 0.

Narrow Multiply
NMUL *acs,acd*



Multiplies the signed integer contained in bits 16–31 of ACD by the signed integer contained in bits 16–31 of ACS. If the result is outside the range of -32,768 to +32,767 inclusive, sets *overflow* to 1; otherwise, *overflow* is 0. Sign extends the lower 16 bits of the result to 32 bits and places these 32 bits in ACD. The contents of ACS and carry remain unchanged.

Narrow Negate
NNEG *acs,acd*

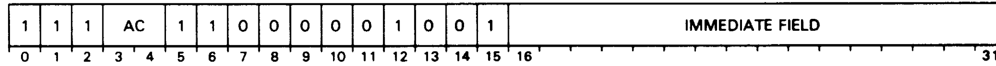


Negates the 16 least significant bits of ACS by performing a two's complement subtract from zero. Sign extends these 16 bits to 32 bits and loads the result in ACD. Sets carry to the value of ALU carry.

NOTE: *Negating the largest negative 16-bit integer, (100000_8) sets overflow to 1.*

Narrow Skip on All Bits Set in Accumulator

NSALA *ac,immediate*

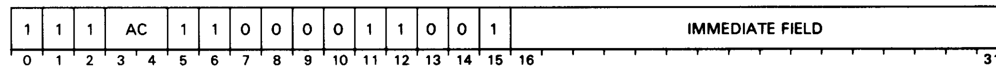


Logically ANDs the value in the immediate field with the complement of the contents of an accumulator and skips depending on the result of the AND.

The instruction performs a logical AND on the contents of the immediate field and the complement of the least significant 16 bits contained in the specified accumulator. If the result of the AND is zero, then the next sequential word is skipped. If the result of the AND is nonzero, the next sequential word is executed. The contents of the specified accumulator remain unchanged. Carry is unchanged and *overflow* is 0.

Narrow Skip on All Bits Set in Memory Location

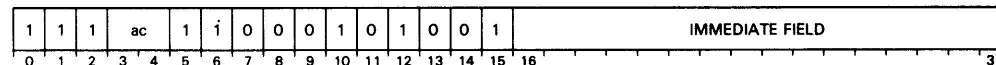
NSALM *ac,immediate*



Performs a logical AND on the contents of the immediate field and the complement of the word addressed by the specified accumulator. If the result of the AND is zero, then execution skips the next sequential word before continuing. If the result of the AND is nonzero, then execution continues with the next sequential word. The contents of the specified accumulator and memory location remain unchanged. Carry is unchanged and *overflow* is 0.

Narrow Skip on Any Bit Set in Accumulator

NSANA *ac,immediate*

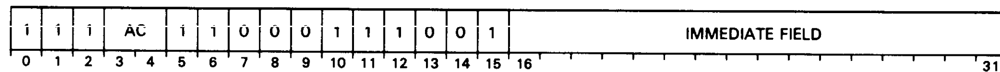


Logically ANDS the contents of an immediate field with the contents of an accumulator and skips, depending on the result.

The instruction performs a logical AND on the contents of the immediate field and the least significant 16 bits contained in the specified accumulator. If the result of the AND is nonzero, the next sequential word is skipped. If the result of the AND is zero, the next sequential word is executed. The contents of the specified accumulator remain unchanged. Carry is unchanged and *overflow* is 0.

Narrow Skip on Any Bit Set in Memory Location

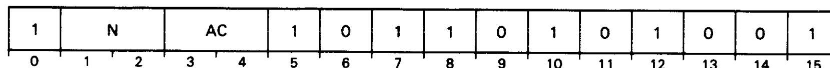
NSANM *ac,immediate*



The instruction performs a logical AND on the contents of the immediate field and the contents of the word addressed by the specified accumulator. If the result of the AND is nonzero, then the next sequential word is skipped. If the result of the AND is zero, the next sequential word is executed. The contents of the specified accumulator and memory location remain unchanged. Carry is unchanged and *overflow* is 0.

Narrow Subtract Immediate

NSBI *n,ac*



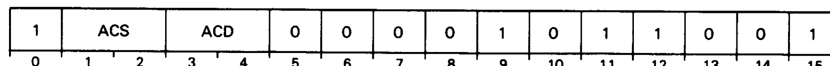
Subtracts a value in the range of 1 to 4 from the value contained in an accumulator.

The instruction subtracts the value $n+1$ from the 16-bit value contained in the specified accumulator, where n is an integer in the range of 0 to 3. Stores the result in bits 16–31 of the specified accumulator. Sign extends the specified accumulator to 32 bits. Sets carry to the value of ALU carry. Sets *overflow* to 1 if there is an ALU overflow.

NOTE: *The assembler takes the coded value of* n *and subtracts 1 from it before placing it in the immediate field. Therefore, the programmer should code the exact value that he wishes to subtract.*

Narrow Subtract

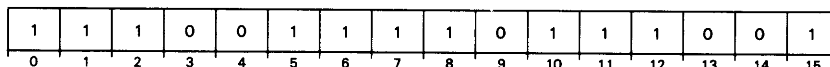
NSUB *acs,acd*



Subtracts the 16-bit integer contained in bits 16–31 of ACS from the 16-bit integer contained in bits 16–31 of ACD. Stores the result in bits 16–31 of ACD. Sign extends ACD to 32 bits. Sets carry to the value of ALU carry, and *overflow* to 1 if there is an ALU overflow.

OR Referenced Bits

ORFB



Performs an inclusive OR on the referenced bits and the contents of a word string.

Bits 13–31 of AC1 contain a pageframe number. Bits 28-31 of AC1 are 0 so that the initial page frame number is a multiple of 16.

AC0 specifies the number of words to be ORed.

AC2 contains the starting address of a word string. The instruction will inclusively OR the contents of this word string with the referenced bits.

The instruction fetches the referenced bits of 16 consecutive pageframes, beginning with the pageframe specified by AC1. Exclusively ORs these 16 bits with the 16-bit word specified by AC2. Stores the result of the OR in the location specified by AC2. Resets the 16 referenced bits to 0, decrements AC0 by 1, increments AC1 by 16, and increments AC2 by 1.

If the contents of AC0 are 0, the instruction ends. If AC0 does not contain 0, the instruction continues the ORing process with the next 16 referenced bits specified by AC1 and the word specified by AC2. Carry is unchanged and *overflow* is 0.

> NOTE: *If AC1 contains a nonexistent pageframe number, or if the ATU is not enabled when this instruction executes, the result of the instruction is undefined.*
>
> *This is a privileged instruction.*

Purge the ATU
PATU

1	1	1	0	0	1	1	1	1	0	0	1	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Purges the entire ATU of all entries. Carry is unchanged and *overflow* is 0.

> NOTE: *This is a privileged instruction.*

Pop Block and Execute
PBX

1	0	0	0	0	1	1	1	0	1	0	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Saves a 16-bit instruction, pops a wide return block off the stack, and executes the saved instruction. Carry and *overflow* are indeterminate.

Bits 16–31 of AC0 contain a 16-bit instruction.

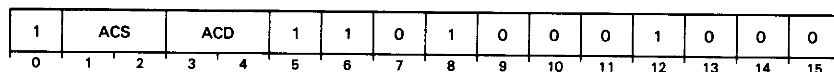
The instruction temporarily saves the instruction contained in bits 16–31 of AC0. Executes a **WPOPB** instruction, except that execution does not continue with the value loaded into the PC. After the wide return block is popped, the instruction executes the instruction that was temporarily saved. The executed instruction determines the value of the processor flags. The next instruction to be executed is addressed by the popped value of the PC + 1.

Note that the value popped off the stack and loaded into the PC must reference a **BKPT** instruction. If it does not, undefined results occur. If it does, then the instruction effectively substitutes the 16-bit instruction in AC0 for the **BKPT** instruction referenced

by the PC after the pop.

Pop Multiple Accumulators

POP *acs,acd*



Pops 1 to 4 words off the stack and places them in the indicated accumulators.

The set of accumulators from ACS through ACD, bits 16-31, is filled with words popped from the stack. Bits 16-31 of the accumulators are filled in descending order, starting with bits 16-31 of the accumulator specified by ACS and continuing down through bits 16-31 of the accumulator specified by ACD, wrapping around if necessary, with AC3 following AC0. If ACS is equal to ACD, only one word is popped and it is placed in ACS.

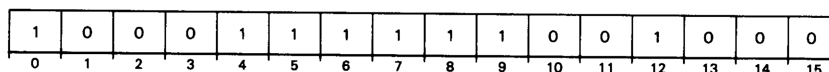
The stack pointer is decremented by the number of accumulators popped and the frame pointer is unchanged. A check for underflow is made only after the entire pop operation is done.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

This instruction leaves carry unchanged; *overflow* is 0.

Pop Block

POPB



Returns control from a *System Call* routine or an I/O interrupt handler that does not use the stack change facility of the *Vector* instruction.

Five words are popped off the stack and placed in predetermined locations. The words popped and their destinations are as follows:

Word Popped	Destination
1	Bit 0 is loaded into carry
	Bits 1-15 are loaded into the PC
2	AC3
3	AC2
4	AC1
5	AC0

Sequential operation is continued with the word addressed by the updated value of the program counter. Carry remains unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

NOTE: *If the I/O handler uses the stack change facility of the* Vector on Interrupting Device Code *instruction, do not use the* Pop Block *instruction. Use the* Restore *instruction instead.*

Pop PC And Jump
POPJ

1	0	0	1	1	1	1	1	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pops the top word off the stack and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

The stack pointer is decremented by one and the frame pointer is unchanged. A check for underflow occurs after the pop operation. Carry remains unchanged and *overflow* is 0.

Push Multiple Accumulators

PSH *acs,acd*

1	ACS		ACD		1	1	0	0	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pushes the contents of 1 to 4 accumulators onto the stack.

Bits 16-31 of the set of accumulators from ACS through ACD are pushed onto the stack. The contents of bits 16-31 of the accumulators are pushed in ascending order, starting with bits 16-31 of the AC specified by ACS and continuing up through bits 16-31 of the AC specified by ACD, wrapping around if necessary, with AC0 following AC3. The contents of the accumulators remain unchanged. If ACS equals ACD, only ACS is pushed. Carry remains unchanged and *overflow* is 0.

The stack pointer is incremented by the number of accumulators pushed and the frame pointer is unchanged. A check for overflow is made only after the entire push operation finishes.

Push Jump

PSHJ *[@]displacement[,index]*

1	0	0	0	0	1	INDEX		1	0	1	1	1	0	0	0	@	DISPLACEMENT	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	31

Pushes the address of the next sequential instruction onto the stack, computes the effective address, *E*, and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Carry remains unchanged and *overflow* is 0.

Push Return Address

PSHR

1	0	0	0	0	1	1	1	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pushes the address of this instruction + 2 onto the narrow stack. Carry remains unchanged and *overflow* is 0.

Reset Referenced Bit

RRFB

1	1	1	0	0	1	1	1	1	0	1	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Resets the specified referenced bits.

AC1 contains a pageframe number in bits 13–31. Bits 28–31 are cleared to 0 so that the initial pageframe number is a multiple of 16.

AC0 contains an origin 0 pageframe count that specifies the number of groups of 16 referenced bits to reset. A count of 0 means that the instruction resets 16 pageframes.

The instruction sets to 0 the referenced bits of 16 contiguous pageframes, starting with the pageframe specified by the contents of AC1. Decrements the contents of AC0 by 1 and increments the contents of AC1 by 16.

If AC0 contains a nonnegative number after the decrement, the instruction repeats the operation with the next 16 pageframes. If AC0 contains a negative number, the instruction ends. Carry is unchanged and *overflow* is 0.

> **NOTE:** *If AC0 specifies a nonexistent pageframe, or if the ATU is not enabled when this instruction executes, the result of the instruction is undefined.*
>
> *This is a privileged instruction.*

Restore

RSTR

1	1	1	0	1	1	1	1	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Returns control from certain types of I/O interrupts.

Pops nine words off the stack and places them in predetermined locations. The words popped and their destinations are as follows:

Word Popped	Destination
1	Bit 0 is loaded into carry
	Bits 1-15 are loaded into the PC
2	AC3
3	AC2
4	AC1
5	AC0
6	Stack fault address
7	Stack limit
8	Frame pointer
9	Stack pointer

Sequential operation continues with the word addressed by the updated value of the program counter.

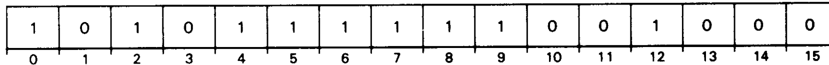
Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Carry remains unchanged and *overflow* is 0.

NOTES: *Use the* Restore *instruction to return control to the program only if the I/O interrupt handler uses the stack change facility of the* Vector on Interrupting Device Code *instruction.*

The Restore *instruction does not check for stack underflow.*

Return
RTN



Returns control from subroutines that issue a *Save* instruction at their entry points.

The *Save* instruction loads the current value of the stack pointer into the frame pointer. The *Return* instructions uses this value of the frame pointer to pop a standard return block off of the stack. The format of the return block is:

Word Popped	Destination
1	Bit 0 is loaded into carry
	Bits 1-15 are loaded into the PC
2	AC3
3	AC2
4	AC1
5	AC0

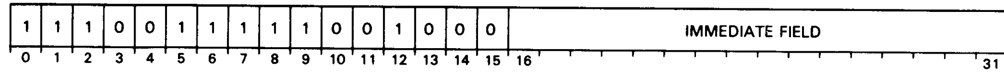
After popping the return block, the *Return* instruction loads the decremented value of the frame pointer into the stack pointer and the popped value of AC3 into the frame pointer.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Carry remains unchanged and *overflow* is 0.

Save

SAVE *i*



Saves the information required by the *Return* instruction.

Saves the current value of the stack pointer in a temporary location. Adds 5 plus the unsigned, 16-bit integer contained in the immediate field to the current value of the stack pointer and loads the result into location 40. Compares this new value of the stack pointer to the stack limit to check for overflow. If no overflow condition exists, then the instruction places the current value of the frame pointer in bits 16-31 of AC3. Fetches the contents of the temporary location and loads them into the frame pointer. The instruction uses the value in the frame pointer to push a five-word return block. The formats and contents of the five-word return block is as follows:

Word Pushed	Contents
1	Bits 16-31 of AC0
2	Bits 16-31 of AC1
3	Bits 16-31 of AC2
4	Frame pointer before the *Save*
5	Bit 0 = carry Bits 1-15 = bits 16-31 of AC3

After pushing the return block on the narrow stack, the instruction places the value of the frame pointer (which now contains the old value of the stack pointer + 5) in bits 16-31 of AC3. Carry remains unchanged and *overflow* is 0.

If an overflow condition exists, the *Save* instruction transfers control to the stack fault routine. The program counter in the fault return block contains the address of the *Save* instruction.

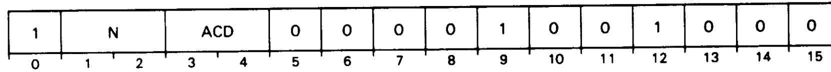
The *Save* instruction allocates a portion of the stack for use by the procedure which executed the *Save*. The value of the *frame size*, contained in the immediate field, determines the number of words in this stack area. This portion of the stack will not normally be accessed by push and pop operations, but will be used by the procedure for temporary storage of variables, counters, etc. The frame pointer acts as the reference point for this storage area.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

Use the *Save* instruction with the *Jump to Subroutine* instruction. The *Jump to Subroutine* instruction places the return value of the program counter in bits 16-31 of AC3. *Save* then pushes the return value (contents of bits 16-31 of AC3) into bits 1–15 of the fifth word pushed.

Subtract Immediate

SBI *n,ac*



Subtracts an unsigned integer in the range 1 to 4 from the contents of an accumulator.

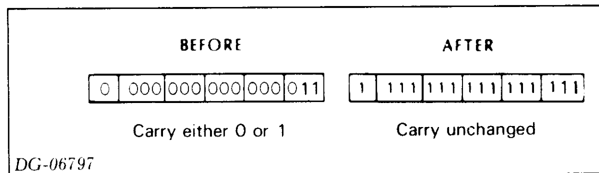
The instruction subtracts the value $N+1$ from the unsigned 16-bit number contained in bits 16-31 of the specified accumulator and the result is placed in bits 16-31 of ACD. Carry remains unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

> NOTE: *The assembler takes the coded value of* **n** *and subtracts 1 from it before placing it in the immediate field. Therefore the programmer should code the exact value he wishes to subtract.*

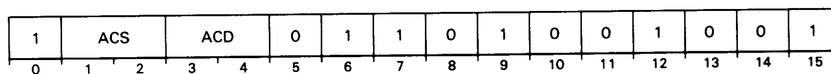
Example

Assume that bits 16-31 of AC2 contains 000003_8. After the instruction **SBI 4,2** is executed, bits 16-31 of AC2 contains 177777_8 and carry remains unchanged.



Sign Extend

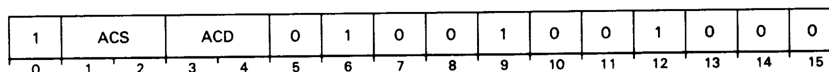
SEX *acs,acd*



Sign extends the 16-bit integer contained in ACS to 32 bits and loads the result into ACD. The contents of ACS remain unchanged, unless ACS and ACD are specified to be the same accumulator. Carry is unchanged and *overflow* is 0.

Skip If ACS Greater Than Or Equal to ACD

SGE *acs,acd*



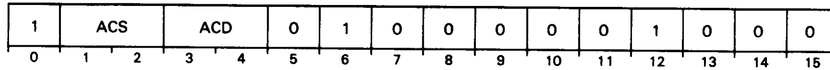
Compares two signed integers in two accumulators and skips if the first is greater than or equal to the second.

The signed two's complement numbers in bits 16-31 of ACS and ACD are algebraically compared. If the number in bits 16-31 of ACS is greater than or equal to the number in bits 16-31 of ACD, the next sequential word is skipped. The contents of ACS, ACD, and

carry remain unchanged. *Overflow* is 0.

> NOTE: *The* Skip If ACS Greater Than ACD *and* Skip If ACS Greater Than Or Equal To ACD *instructions treat the contents of the specified accumulators as signed, two's complement integers. To compare unsigned integers, use the* Subtract and Add Complement *instruction.*

Skip If ACS Greater Than ACD
SGT *acs,acd*

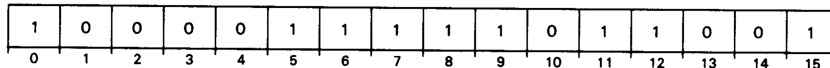


Compares two signed integers in two accumulators and skips if the first is greater than the second.

The signed, two's complement numbers in bits 16-31 of ACS and ACD are algebraically compared. If the number in bits 16-31 of ACS is greater than the number in bits 16-31 of ACD, the next sequential word is skipped. The contents of ACS, ACD, and carry remain unchanged.

> NOTE: *The* Skip If ACS Greater Than ACD *and* Skip If ACS Greater Than Or Equal To ACD *instructions treat the contents of the specified accumulators as signed, two's complement integers. To compare unsigned integers, use the* Subtract and Add Complement *instruction.*

Store Modified and Referenced Bits
SMRF



Stores new values into the modified and referenced bits of a pageframe.

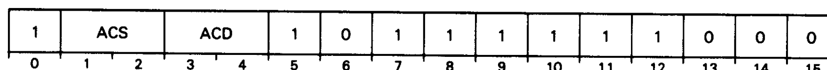
AC1 contains a pageframe number in bits 13 –31.

The instruction fetches the contents of the two least significant bits of AC0. Stores these values in the modified and referenced bits of the pageframe specified by AC1. Carry is unchanged and *overflow* is 0.

If the ATU is not enabled, undefined results will occur. If a nonexistent pageframe is specified, the instruction loads the appropriate modified and referenced bits with indeterminate data.

> NOTE: *This is a privileged instruction.*

Skip On Non-Zero Bit
SNB *acs,acd*



The two accumulators form a bit pointer. If the addressed bit is 1, the next sequential

word is skipped.

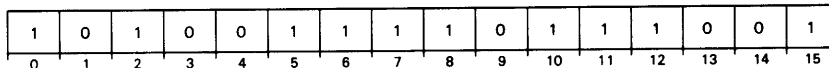
Forms a 32-bit bit pointer from the contents of bits 16-31 of both ACS and ACD. Bits 16-31 of ACS contains the high-order 16 bits and bits 16-31 of ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16 bits of the bit pointer and assumes the high-order 16 bits are 0.

If the addressed bit in memory is 1, the next sequential word is skipped. The contents of ACS, ACD, and carry remain unchanged. *Overflow* is 0. The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

> **NOTE:** *The bit pointer formed by the two accumulators cannot make indirect memory references.*

Skip on OVR Reset

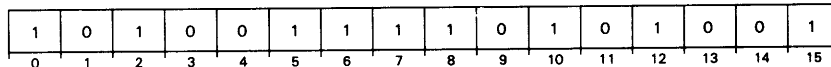
SNOVR



Tests the value of **OVR**. If the flag has the value 0, the next sequential word is skipped. If the flag has the value 1, the next sequential word is executed. Carry is unchanged and *overflow* is 0.

Store Processor Status Register From AC0

SPSR

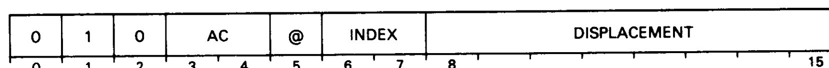


Stores the contents of AC0 in the PSR.

Loads the contents of AC0 bits 0, 1, and 2 into **OVK, OVR,** and **IRES**, respectively. The contents of AC0 remain unchanged. Carry is unchanged and *overflow* is 0.

Store Accumulator

STA *ac,[@]displacement[,index]*



Stores the contents of bits 16-31 of an accumulator into a memory location.

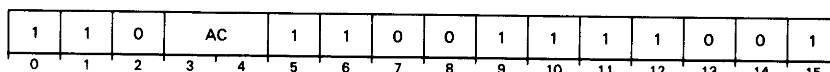
Places the contents of bits 16-31 of the specified accumulator in the word addressed by the effective address, *E*. The previous contents of the location addressed by *E* are lost.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

The contents of carry and the specified accumulator remain unchanged. *Overflow* is 0.

Store Accumulator in WFP

STAFP *ac*

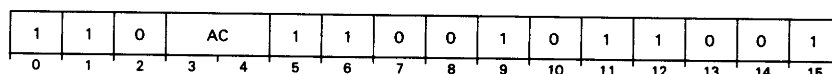


1	1	0	AC	1	1	0	0	1	1	1	1	0	0	1
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15

Stores a copy of the contents of the specified accumulator into WFP (the wide frame pointer). Carry is unchanged and *overflow* is 0.

Store Accumulator in WSB

STASB *ac*

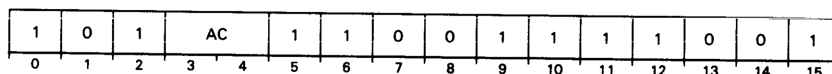


1	1	0	AC	1	1	0	0	1	0	1	1	0	0	1
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15

Stores a copy of the contents of the specified accumulator into WSB (the wide stack base) as well as locations $26–27_8$ of the current segment. Carry is unchanged and *overflow* is 0.

Store Accumulator in WSL

STASL *ac*

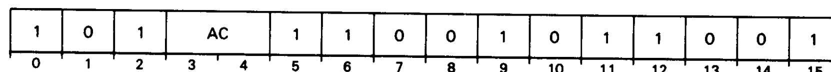


1	0	1	AC	1	1	0	0	1	1	1	1	0	0	1
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15

Stores a copy of the contents of the specified accumulator into WSP (the wide stack pointer) as well as locations $24–25_8$ of the current segment. Carry is unchanged and *overflow* is 0.

Store Accumulator in WSP

STASP *ac*

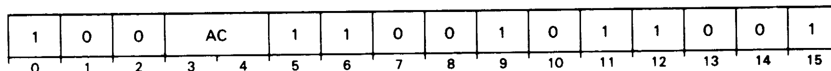


1	0	1	AC	1	1	0	0	1	0	1	1	0	0	1
0	1	2	3 4	5	6	7	8	9	10	11	12	13	14	15

Stores a copy of the contents of the specified accumulator into WSP (the wide stack pointer). Carry is unchanged and *overflow* is 0.

Store Accumulator into Stack Pointer Contents

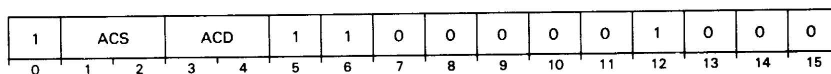
STATS *ac*



Uses the contents of WSP (the wide stack pointer) as the address of a double word. Stores a copy of the contents of the specified accumulator at the address contained in WSP. Carry is unchanged and *overflow* is 0.

Store Byte

STB *acs,acd*



Moves the rightmost byte of ACD to a byte in memory. ACS contains the byte pointer.

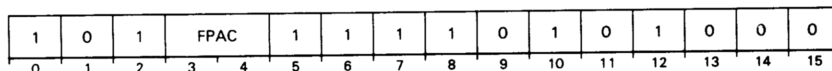
Places bits 24–31 of ACD in the byte addressed by the byte pointer contained in bits 16-31 of ACS.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

The contents of ACS, ACD, and carry remain unchanged. *Overflow* is 0.

Store Integer

STI *fpac*



Under the control of accumulators AC1 and AC3, translates the contents of the specified FPAC to an integer of the specified type and stores it, right-justified, in memory, beginning at the specified location. The instruction leaves the floating point number unchanged in the FPAC, and destroys the previous contents of memory at the specified location(s).

Bits 16-31 of AC1 must contain the data-type indicator describing the integer.

Bits 16-31 of AC3 must contain a byte pointer which is the address of the high-order byte of the number in memory.

Upon successful completion, the instruction leaves accumulators AC0 and AC1 unchanged. AC2 contains the original contents of AC3 and AC3 contains a byte pointer which is the address of the next byte after the destination field. *Overflow* is 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

NOTES: *If the number in the specified FPAC has any fractional part, the result of the instruction is undefined. Use the* Integerize *instruction to clear any fractional part.*

If the destination field cannot contain the entire number being stored, high-order digits are discarded until the number will fit into the destination. The remaining low-order digits are stored and carry is set to 1.

For data types 0, 1, 2, 3, 4, and 5, if the number being stored will not fill the destination field, the high-order bytes to the right of the sign are set to 0.

For data type 6, if the number being stored will not fill the destination field, the sign bit is extended to the left to fill the field.

For data type 7, if the number being stored will not fill the destination field, the low-order bytes are set to 0.

Store Integer Extended
STIX

1	1	0	0	1	1	1	1	1	0	1	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Converts the contents of the four FPAC's to integer form and uses the low-order 8 digits of each to form a 32-digit integer. The instruction stores this integer, right-justified, in memory beginning at the specified location. The sign of the integer is the logical OR of the signs of all four FPAC's. The previous contents of the addressed memory locations are lost. Sets carry to 0. The contents of the FPAC's remain unchanged. The condition codes in the FPSR are unpredictable.

Bits 16-31 of AC1 must contain the data-type indicator describing the form of the in memory.

Bits 16-31 of AC3 must contain a byte pointer which is the address of the high-order byte of the destination field in memory.

Upon successful termination, the contents of AC0 are undefined; the contents of AC1 remain unchanged; AC2 contains the original contents of AC3; and AC3 contains a byte pointer which is the address of the next byte after the destination field. *Overflow* is 0.

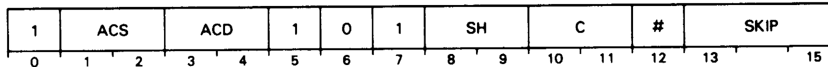
The 32-bit effective address generated by this instruction is constrained to be within the first 64 Kbyte of the current segment.

NOTES: *If the destination field is not large enough to contain the number being stored, the instruction disregards high-order digits until the number will fit in the destination. The instruction stores low-order digits remaining and sets carry to 1.*

For data types 0, 1, 2, 3, 4, and 5, if the number being stored will not fill the destination field, the instruction sets the high-order bytes to 0.

For data type 6, if the number being stored will not fill the destination field, the instruction extends the sign bit to the left to fill the field.

Subtract
SUB*[c][sh][#]* *acs,acd[,skip]*



Performs unsigned integer subtraction and complements carry if appropriate.

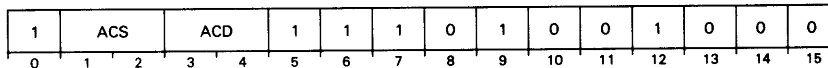
Initializes carry to its specified value. The instruction subtracts the unsigned, 16-bit number in bits 16-31 of ACS from the unsigned, 16-bit number in bits 16-31 of ACD by taking the two's complement of the number in ACS and adding it to the number in ACD. The instruction places the result of the addition in the shifter. If the operation produces a carry of 1 out of the high-order bit, the instruction complements carry. The instruction performs the specified shift operation and places the result of the shift in bits 16-31 of ACD if the no-load bit is 0. If the skip condition is true, the instruction skips the next sequential word.

If the load option is specified, bits 0-15 of ACD are undefined.

Overflow is 0 for this instruction.

NOTE: *If the number in ACS is less than or equal to the number in ACD, the instruction complements carry.*

System Call
SYC *acs,acd*



Pushes a return block and transfers control to the *system call handler.*

If a user map is enabled, the instruction disables it and pushes a return block onto the stack. The program counter in the return block points to the instruction immediately following the *System Call* instruction. After pushing the return block, the instruction executes a *Jump Indirect* to location 2, which contains the address of the *system call handler.*

If this instruction disables a user map, then I/O interrupts cannot occur between the time the *System Call* instruction is executed and the time the first instruction of the system call handler is executed.

If the ATU is enabled, a privileged instruction protection fault occurs.

This instruction leaves carry unchanged; *overflow* is 0.

NOTES: *If both accumulators are specified as AC0, the instruction does not push a return block onto the stack. The contents of AC0 remain unchanged.*

The assembler recognizes the mnemonic SCL *as equivalent to* SYC 1,1.

The assembler recognizes the mnemonic SVC *as equivalent to* SYC 0,0.

Skip On Zero Bit
SZB *acs,acd*

1	ACS		ACD		1	0	0	1	0	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The two accumulators form a bit pointer. If the addressed bit is zero, the next sequential word is skipped.

Forms a 32-bit bit pointer from the contents of bits 16-31 of both ACS and ACD. Bits 16-31 of ACS contains the high-order 16 bits and bits 16-31 of ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16 bits of the bit pointer and assumes the high-order 16 bits are 0.

If the addressed bit in memory is 0, the next sequential word is skipped. The contents of ACS and ACD remain unchanged.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

This instruction leaves carry unchanged; *overflow* is 0.

NOTE: *The bit pointer contained in ACS and ACD cannot make indirect memory references.*

Skip On Zero Bit And Set To One
SZBO *acs,acd*

1	ACS		ACD		1	0	0	1	1	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The two accumulators form a bit pointer. The instruction sets the addressed bit to 1. If the addressed bit was 0 before being set to 1, the instruction skips the next sequential word. The contents of ACS, ACD, and carry remain unchanged. *Overflow* is 0.

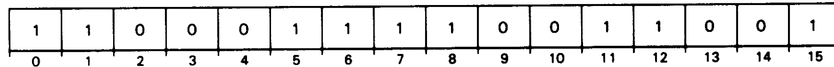
Forms a 32-bit bit pointer from the contents of bits 16-31 of ACS and ACD. Bits 16-31 of ACS contains the high-order 16 bits and bits 16-31 of ACD contains the low-order 16 bits of the bit pointer. If ACS and ACD are specified as the same accumulator, the instruction treats the accumulator contents as the low-order 16 bits of the bit pointer and assumes the high-order 16 bits are 0.

The 32-bit effective address generated by this instruction is constrained to be within the first 32 Kword of the current segment.

NOTES: *The bit pointer contained in ACS and ACD must not make indirect memory references*

This instruction facilitates the use of bit maps for such purposes as allocation of facilities (memory blocks, I/O devices, etc.) to several processes, or tasks, that may interrupt one another, or in a multiprocessor environment. The bit is tested and set to 1 in one memory cycle.

Skip on Valid Byte Pointer
VBP



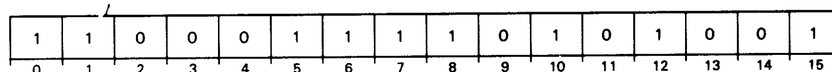
Checks a byte pointer for valid reference, and skips or does not skip the next word depending on the outcome of the check. Carry is unchanged and *overflow* is 0.

AC1 contains a ring number in bits 1–3; all other bits contain zeroes.

AC0 contains a 32-bit byte pointer.

The instruction compares the ring field of AC1 to the ring field of AC0. If AC1's ring field is greater than AC0's ring field, the next sequential word is executed; otherwise, the next sequential word is skipped.

Skip on Valid Word Pointer
VWP



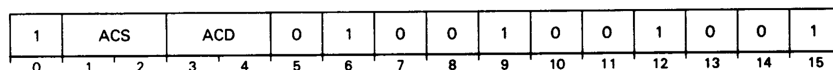
Checks a word pointer for valid reference, and skips or does not skip the next word depending on the outcome of the check.

AC1 contains a ring number in bits 1–3; all other bits contain zeroes.

AC0 contains a 31-bit word pointer (indirectable).

The instruction compares the ring field of AC1 to the ring field of AC0. If AC1's ring field is greater than AC0's ring field, the next sequential word is executed; otherwise, the next sequential word is skipped. Carry is unchanged and *overflow* is 0.

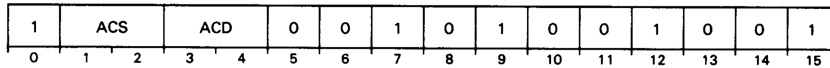
Wide Add Complement
WADC *acs,acd*



Forms the logical complement of the 32-bit integer contained in ACS and adds it to the 32-bit integer contained in ACD. Stores the result in ACD. Sets carry to the value of ALU carry. Sets *overflow* to 1 if there is an ALU overflow.

Wide Add

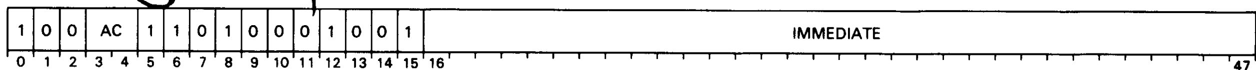
WADD *acs,acd*



Adds the 32-bit fixed point integer contained in ACS to the 32-bit fixed point integer contained in ACD. Stores the result in ACD. Sets carry to ALU carry. Sets *overflow* to 1 if there is an ALU overflow.

Wide Add With Wide Immediate

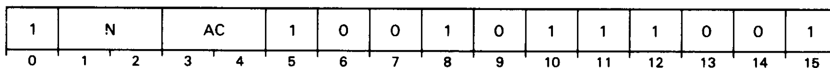
WADDI *ac,immediate*



Adds the 32-bit fixed point integer contained in the immediate field to the 32-bit fixed point integer contained in the specified accumulator. Stores the result in the specified accumulator. Sets *overflow* to 1 if there is an ALU overflow. Sets carry to the value of the ALU carry.

Wide Add Immediate

WADI *n,ac*

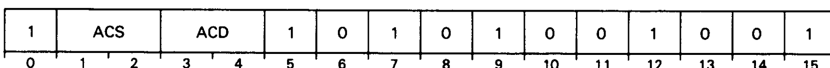


Adds the value $n+1$ to the 32-bit fixed point integer contained in the specified accumulator. Stores the result in the specified accumulator. Sets carry to the value of ALU carry. Sets *overflow* to 1 if there is an ALU overflow.

NOTE: *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact value that he wishes to add.*

Wide AND with Complemented Source

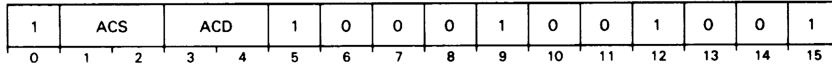
WANC *acs,acd*



Forms the one's complement of the 32 bits contained in ACS and logically ANDs it with the 32 bits contained in ACD. Stores the result in ACD. Carry is unchanged and *overflow* is 0.

Wide AND

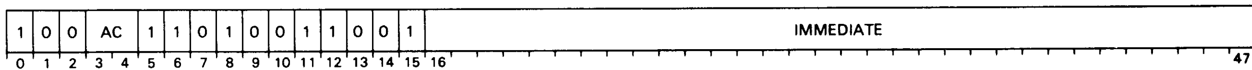
WAND *acs,acd*



Forms the logical AND between corresponding bits of ACS and ACD. Loads the 32-bit result into ACD. The contents of ACS remain unchanged. Carry is unchanged and *overflow* is 0.

Wide AND Immediate

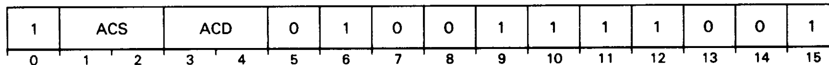
WANDI *ac, immediate*



Forms the logical AND between corresponding bits of the specified accumulator and the value contained in the literal field. The instruction places the 32-bit result of the logical AND in the specified accumulator. Carry is unchanged and *overflow* is 0.

Wide Arithmetic Shift

WASH *acs,acd*



Shifts the contents of ACD left or right.

Bits 24–31 of ACS specify the number of bits to shift and the direction of shifting.

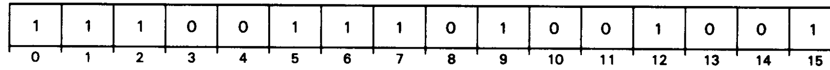
If ACS contains a positive number, the instruction shifts the contents of ACD left; zeroes fill the vacated bit positions. If ACS contains a negative number, the instruction shifts the contents of ACD right; the sign bit fills the vacated bit positions. If ACS contains zero, no shifting occurs. The instruction ignores bits 0–23 of ACS.

If the instruction is to shift the contents of ACD to the right, it truncates the contents one bit position for each shift.

In shifting negative numbers to the right, rounding towards zero is performed. For instance, -3 shifted one position to the right results in -1.

The value of ACS and carry remain unchanged. If, while performing a left shift, you shift out a bit whose value is the complement of ACD's sign bit, *overflow* is set to 1. Otherwise, *overflow* is 0.

Wide Block Move
WBLM



Moves words sequentially from one memory location to another, treating them as unsigned, 32-bit integers.

AC1 contains the two's complement of the number of words to be moved. If the contents of AC1 are positive, then data movement progresses from the lowest memory location to the highest (ascending). If the contents of AC1 are negative, then data movement progresses from the highest memory location to the lowest (descending).

Bits 1–31 of AC2 contain the address of the source location. Bits 1–31 of AC3 contain the address of the destination location. The address in bits 1–31 of AC2 or AC3 is an indirect address if bit 0 of that accumulator is 1. In that case, the instruction follows the indirection chain before placing the resultant effective address in the accumulator.

AC	Contents
0	Unused
1	Number of words to be moved
2	Source address
3	Destination address

For each word moved, the instruction decrements the count in AC1 by 1. If data movement is ascending, the instruction increments the source and destination addresses by 1 for each word moved. If data movement is descending, the instruction decrements the source and destination addresses by 1 for each word moved.

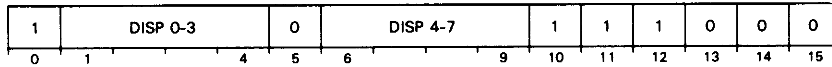
Upon completion of the instruction, AC1 contains zeroes, and AC2 and AC3 point to the word following (ascending) or preceding (descending) the last word in their respective fields. AC0 is unused. Carry is unchanged and *overflow* is 0.

> **NOTES:** *Since this instruction may require a long time to execute, it is interruptable. When this instruction is interrupted, the processor saves the address of the* **WBLM** *instruction. This instruction updates addresses and word count after storing each word, so any interrupt service routine returning control via the saved address will correctly restart the* **WBLM** *instruction.*
>
> *If data movement is descending and a ring crossing would occur, a protection trap occurs and this instruction does not execute. AC1 will contain the value 4.*

When updating the source and destination addresses, the *Wide Block Move* instruction forces bit 0 of the result to 0. This ensures that upon return from an interrupt, the *Wide Block Move* instruction will not try to resolve an indirect address in either AC2 or AC3.

Load PC

WBR *displacement*

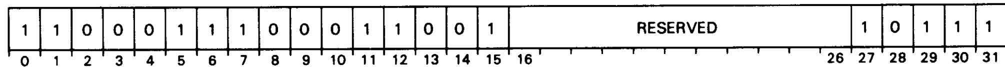


Adds the 31-bit value contained in the PC to the value of the displacement and places the result in the PC. Carry is unchanged and *overflow* is 0.

NOTE: *The processor always forces the value loaded into the PC to reference a location in the current segment of execution.*

Wide Search Queue Backward

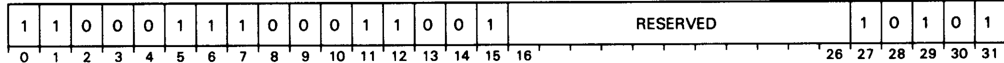
WBSAC



See instruction entry "Search Queue".

Wide Search Queue Backward

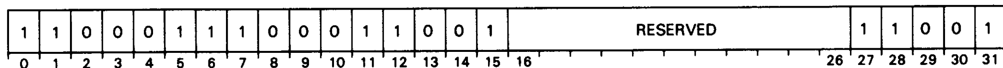
WBSAS



See instruction entry "Search Queue".

Wide Search Queue Backward

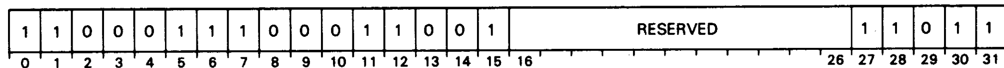
WBSE



See instruction entry "Search Queue".

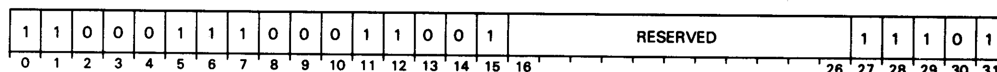
Wide Search Queue Backward

WBSGE



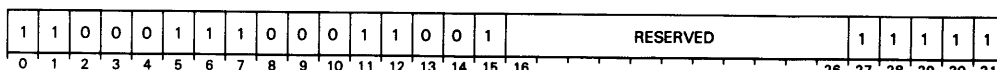
See instruction entry "Search Queue".

Wide Search Queue Backward
WBSLE



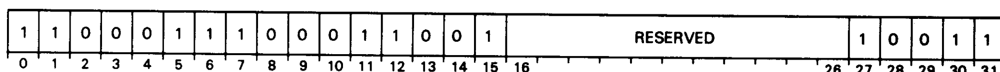
See instruction entry "Search Queue".

Wide Search Queue Backward
WBSNE



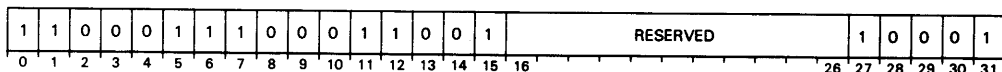
See instruction entry "Search Queue".

Wide Search Queue Backward
WBSSC



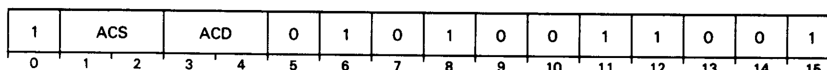
See instruction entry "Search Queue".

Wide Search Queue Backward
WBSSS



See instruction entry "Search Queue".

Wide Set Bit to One
WBTO *acs,acd*



Sets the specified bit to one. Carry is unchanged and *overflow* is 0.

ACS contains a 31-bit word address.

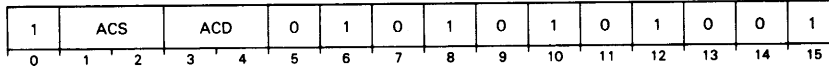
ACD contains a bit offset.

The instruction sets the bit specified by ACS and ACD to one. The contents of ACS and ACD remain unchanged.

If ACS and ACD are specified to be the same accumulator, then the processor assumes the word address is zero within the current segment. In this case, the specified accumulator contains a 32-bit bit pointer.

Wide Set Bit to Zero

WBTZ *acs,acd*



1	ACS	ACD	0	1	0	1	0	1	0	1	0	0	1
0	1–2	3–4	5	6	7	8	9	10	11	12	13	14	15

Sets the specified bit to zero. Carry is unchanged and *overflow* is 0.

ACS contains a 31-bit word address.

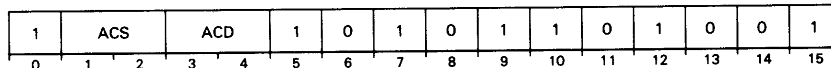
ACD contains a bit offset.

The instruction sets the bit specified by ACS and ACD to zero. The contents of ACS and ACD remain unchanged.

If ACS and ACD are specified to be the same accumulator, then the processor assumes the word address is zero within the current segment. In this case, the specified accumulator contains a 32-bit bit pointer.

Wide Compare to Limits

WCLM *acs,acd*



1	ACS	ACD	1	0	1	0	1	1	0	1	0	0	1
0	1–2	3–4	5	6	7	8	9	10	11	12	13	14	15

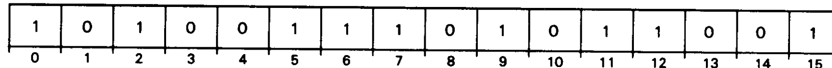
Compares a signed integer with two limit values and skips if the integer is between the limit values. The accumulators determine the location of the limit values. Carry is unchanged and *overflow* is 0.

Compares the signed, two's complement integer in ACS to two signed, two's complement integer limit values, *L* and *H*. If the number in ACS is greater than or equal to *L* and less than or equal to *H*, execution skips the next sequential word before continuing. If the number in ACS is less than *L* or greater than *H*, execution continues with the next sequential word.

If ACS and ACD are specified as different accumulators, bits 1–31 of ACD contain the address of the limit value *L*. The word following *L* contains the limit value *H*. Bit 0 of ACD is ignored.

If ACS and ACD are specified as the same accumulator, the integer to be compared must be in that accumulator and the limit values *L* and *H* must be in the two words following the instruction. The first word contains *L*, and the second contains *H*. The third word contains the next sequential word of the program.

Wide Character Compare
WCMP



Under control of the four accumulators, compares two strings of bytes and returns a code in AC1 reflecting the results of the comparison.

The instruction compares the strings one byte at a time. Each byte is treated as an unsigned 8-bit binary quantity in the range $0–255_{10}$. If two bytes are not equal, the string whose byte has the smaller numerical value is, by definition, the *lower valued* string. Both strings remain unchanged. The four accumulators contain parameters passed to the instruction. Two accumulators specify the starting address, the number of bytes, and the direction of processing (ascending or descending addresses) for each string.

AC0 specifies the length and direction of comparison for string 2. If the string is to be compared from its lowest memory location to the highest, AC0 contains the unsigned value of the number of bytes in string 2. If the string is to be compared from its highest memory location to the lowest, AC0 contains the two's complement of the number of bytes in string 2.

AC1 specifies the length and direction of comparison for string 1. If the string is to be compared from its lowest memory location to the highest, AC0 contains the unsigned value of the number of bytes in string 1. If the string is to be compared from its highest memory location to the lowest, AC1 contains the two's complement of the number of bytes in string 1.

AC2 contains a byte pointer to the first byte compared in string 2. When the string is compared in ascending order, AC2 points to the lowest byte. When the string is compared in descending order, AC2 points to the highest byte.

AC3 contains a byte pointer to the first byte compared in string 1. When the string is compared in ascending order, AC3 points to the lowest byte. When the string is compared in descending order, AC3 points to the highest byte.

Code	Comparison Result
-1	String 1 < String 2
0	String 1 = String 2
+1	String 1 > String 2

The strings may overlap in any way. Overlap will not effect the results of the comparison.

Upon completion, AC0 contains the number of bytes left to compare in string 2. AC1 contains the return code as shown in the table above. AC2 contains a byte pointer either to the failing byte in string 2 (if an inequality was found), or to the byte following string 2 (if string 2 was exhausted). AC3 contains a byte pointer either to the failing byte in string 1 (if an inequality was found), or to the byte following string 1 (if string 1 was exhausted). Carry is unchanged and *overflow* is 0.

If AC0 and AC1 both contain zero (both string 1 and string 2 have length zero), the instruction returns 0 in AC1.

If the two strings are of unequal length, the instruction *fakes* space characters $<040_8>$ in place of bytes from the exhausted string, and continues the comparison.

> **NOTE:** *The original contents of AC2 and AC3 must be valid byte pointers to an area in the user's address space. If they are invalid a protection fault occurs, even if no bytes are to be compared. AC1 contains the code 4.*

Wide Character Move Until True
WCMT

1	0	1	0	0	1	1	1	0	1	0	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Under control of the four accumulators, moves a string of bytes from one area of memory to another until either a table-specified delimiter character is moved or the source string is exhausted.

The instruction copies the string one byte at a time. Before it moves a byte, the instruction uses that byte's value to determine if it is a delimiter. It treats the byte as an unsigned 8-bit binary integer (in the range $0–255_{10}$) and uses it as a bit index into a 256-bit delimiter table. If the indexed bit in the delimiter table is zero, the byte pending is not a delimiter, and the instruction copies it from the source string to the destination string. If the indexed bit in the delimiter table is 1, the byte pending is a delimiter; the instruction does not copy it, and the instruction terminates.

The instruction processes both strings in the same direction, either from lowest memory locations to highest (*ascending order*), or from highest memory locations to lowest (*descending order*). Processing continues until there is a delimiter or the source string is exhausted. The four accumulators contain parameters passed to the instruction.

AC0 contains the address (word address), possibly indirect, of the start of the 256-bit (16-word) delimiter table.

AC1 specifies the length of the strings and the direction of processing. If the source string is to be moved to the destination field in ascending order, AC1 contains the unsigned value of the number of bytes in the source string. If the source string is to be moved to the destination field in descending order, AC1 contains the two's complement of the number of bytes in the source string.

AC2 contains a byte pointer to the first byte to be written in the destination field. When the process is performed in ascending order, AC2 points to the lowest byte in the destination field. When the process is performed in descending order, AC2 points to the highest byte in the destination field.

AC3 contains a byte pointer to the first byte to be processed in the source string. When the process is performed in ascending order, AC3 points to the lowest byte in the source string. When the process is performed in descending order, AC3 points to the highest byte in the source string.

The fields may overlap in any way. However, the instruction moves bytes one at a time, so certain types of overlap may produce unusual side effects.

Upon completion, AC0 contains the resolved address of the translation table and AC1 contains the number of bytes that were not moved. AC2 contains a byte pointer to the byte following the last byte written in the destination field. AC3 contains a byte pointer either to the delimiter or to the first byte following the source string. The value of carry is indeterminate and *overflow* is 0.

NOTE: *The original contents of AC0, AC2, and AC3 must be valid byte pointers to an area in the user's address space. If they are invalid a protection fault occurs, even if no bytes are to be stored. AC1 contains the code 4.*

Wide Character Move
WCMV

1	0	0	0	0	1	1	1	0	1	1	1	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Under control of the four 32-bit accumulators, moves a string of bytes from one area of memory to another and returns a value in carry reflecting the relative lengths of source and destination strings.

The instruction copies the source string to the destination field, one byte at a time. The four accumulators contain parameters passed to the instruction. Two accumulators specify the starting address, number of bytes to be copied, and the direction of processing (ascending or descending addresses) for each field.

AC0 specifies the length and direction of processing for the destination field. If the field is to be processed from its lowest memory location to the highest, AC0 contains the unsigned value of the number of bytes in the destination field. If the field is to be processed from its highest memory location to the lowest, AC0 contains the two's complement of the number of bytes in the destination field.

AC1 specifies the length and direction of processing for the source string. If the string is to be processed from its lowest memory location to the highest, AC1 contains the unsigned value of the number of bytes in the source string. If the field is to be processed from its highest memory location to the lowest, AC1 contains the two's complement of the number of bytes in the source string.

AC2 contains a byte pointer to the first byte to be written in the destination field. When the field is written in ascending order, AC2 points to the lowest byte. When the field is written in descending order, AC2 points to the highest byte.

AC3 contains a byte pointer to the first byte copied in the source string. When the field is copied in ascending order, AC3 points to the lowest byte. When the field is copied in descending order, AC3 points to the highest byte.

The fields may overlap in any way. However, the instruction moves bytes one at a time, so certain types of overlap may produce unusual side effects.

Upon completion, AC0 contains 0 and AC1 contains the number of bytes left to fetch from the source field. AC2 contains a byte pointer to the byte following the destination field; and AC3 contains a byte pointer to the byte following the last byte fetched from the source field. The value of carry is indeterminate and *overflow* is 0.

If the source field is shorter than the destination field, the instruction pads the destination field with space characters <040_8>. If the source field is longer than the destination field, the instruction terminates when the destination field is filled and returns the value 1 in carry; otherwise, the instruction returns the value 0 in carry.

NOTES: *If AC0 contains the number 0 at the beginning of this instruction, no bytes are fetched and none are stored. If AC1 is 0 at the beginning of this instruction, the destination field is filled with space characters; note that AC3 must still contain a valid byte pointer.*

The original values of AC2 and AC3 must be valid byte pointers to an area in the user's address space. If they are invalid a protection fault occurs, even if no bytes are to be moved. AC1 contains the code 4.

Wide Count Bits

WCOB *acs,acd*

1	ACS	ACD	1	0	0	1	0	0	0	1	0	0	1
0	1–2	3–4	5	6	7	8	9	10	11	12	13	14	15

Counts the number of bits in ACS whose value is 1. Adds the count of non-zero bits to the 32-bit, signed contents of ACD. The contents of ACS remain unchanged, unless ACS and ACD are the same accumulator. Carry is unchanged and *overflow* is 0.

Wide Complement

WCOM *acs,acd*

1	ACS	ACD	1	0	0	0	1	0	1	1	0	0	1
0	1–2	3–4	5	6	7	8	9	10	11	12	13	14	15

Forms the one's complement of the 32-bit fixed point integer contained in ACS and loads the result into ACD. The contents of ACS remain unchanged, unless ACS equals ACD. Carry is unchanged and *overflow* is 0.

Wide Character Translate

WCTR

1	0	0	0	0	1	1	1	0	1	1	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Under control of the four accumulators, translates a string of bytes from one data representation to another and either moves it to another area of memory or compares it to a second translated string.

The instruction operates in two modes: translate and move, and translate and compare.

When operating in translate and move mode, the instruction translates each byte in string 1 and places it in a corresponding position in string 2. Translation is performed by using each byte as an 8-bit index into a 256-byte translation table. The byte addressed by the index then becomes the translated value.

When operating in translate and compare mode, the instruction translates each byte in string 1 and string 2 as described above, and compares the translated values. Each translated byte is treated as an unsigned 8-bit binary quantity in the range $0–255_{10}$. If two translated bytes are not equal, the string whose byte has the smaller numerical value is, by definition the *lower valued* string. Both strings remain unchanged.

AC0 specifies the address, either direct or indirect, of a word which contains a byte pointer to the first byte in the 256-byte translation table.

AC1 specifies the length of the two strings and the mode of processing. If string 1 is to be processed in translate and move mode, AC1 contains the two's complement of the number of bytes in the strings. If the strings are to be processed in translate and compare mode, AC1 contains the unsigned value of the number of bytes in the strings. Both strings are processed from lowest memory address to highest.

AC2 contains a 32-bit byte pointer to the first byte in string 2.

AC3 contains a 32-bit byte pointer to the first byte in string 1.

Upon completion of a translate and move operation, AC0 contains the address of the word which contains the byte pointer to the translation table and AC1 contains 0. AC2 contains a byte pointer to the byte following string 2 and AC3 contains a byte pointer to the byte following string 1. The value of carry is unchanged and *overflow* is 0.

Upon completion of a translate and compare operation, AC0 contains the address of the word which contains the byte pointer to the translation table. AC1 contains a return code as calculated in the table below. AC2 contains a byte pointer to either the failing byte in string 2 (if an inequality was found) or the byte following string 2 if the strings were identical. AC3 contains a byte pointer to either the failing byte in string 1 (if an inequality was found) or the byte following string 1 if the strings were identical. The value of carry is unchanged and *overflow* is 0.

Code	Result
-1	Translated value of string 1 < Translated value of string 2
0	Translated value of string 1 = Translated value of string 2
+1	Translated value of string 1 > Translated value of string 2

If the length of both string 1 and string 2 is zero, the compare option returns a 0 in AC1.

The fields may overlap in any way. However, processing is done one character at a time, so unusual side effects may be produced by certain types of overlap.

> **NOTE:** *The original contents of AC0, AC2, and AC3 must be valid byte pointers to an area in the user's address space. If they are invalid a protection fault occurs, even if no bytes are to be moved or compared. AC1 contains the code 4.*

Wide Divide

WDIV *acs,acd*

1	ACS	ACD	0	0	1	0	1	1	1	1	0	0	1
0	1–2	3–4	5	6	7	8	9	10	11	12	13	14	15

Divides an integer contained in an accumulator by an integer contained in another accumulator.

The instruction sign extends the signed, 32-bit integer contained in ACD to 64 bits. Divides this integer by the signed, 32-bit integer contained in ACS. If the quotient is within the range -2,147,483,648 to +2,147,483,647 inclusive, loads the quotient in ACD. If the result is not within this range, or if ACS is zero, sets *overflow* to 1 and does not load the quotient into ACD; otherwise, *overflow* is 0. The contents of ACS and carry remain unchanged.

Wide Signed Divide

WDIVS

1	1	1	0	0	1	1	1	0	1	1	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Divides an integer contained in AC0 and AC1 by an integer contained in AC2.

AC0 and AC1 contain a 64-bit, signed integer. AC0 contains the high order bits.

The instruction divides the 64-bit, signed integer contained in AC0 and AC1 by the 32-bit, signed integer contained in AC2. If the quotient is within the range -2,147,483,648 to +2,147,483,647 inclusive, then places the 32-bit quotient in AC1 and the remainder in AC0. If the quotient is not within this range, or AC2 is zero, AC0 and AC1 remain unchanged and *overflow* is 1; otherwise, *overflow* is 0. AC2 and carry will always remain unchanged.

> **NOTE:** *Zero remainders are always positive. All other remainders have the same sign as the dividend.*

Pop Context Block

WDPOP

1	0	0	0	0	1	1	1	1	1	1	1	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Restores the state of the machine to what it was at the time of the last page fault.

The instruction uses the information pointed to by page zero locations $32–33_8$ of Segment 0 to restore the state of the CPU to that of the time of the last page fault. Execution of the interrupted program resumes before, during, or after the instruction that caused the fault, depending on the instruction type and how far it had proceeded before the fault. Carry is unchanged and *overflow* is 0.

NOTE: *This is a privileged instruction.*

Wide Edit
WEDIT

1	0	1	0	0	1	1	1	0	1	1	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Converts a decimal source number from either packed or unpacked form to a string of bytes under the control of an edit subprogram. This subprogram can perform many different operations on the number and its destination field including leading zero suppression, leading or trailing signs, floating fill characters, punctuation control, and insertion of text into the destination field. The instruction also performs operations on alphanumeric data if you specify data type 4.

Upon entry to the *Edit* instruction, the accumulators contain the following data:

- AC0 contains a 32-bit byte pointer to the first opcode of the *Edit* subprogram,
- AC1 contains a data-type indicator describing the number to be processed,
- AC2 contains a 32-bit byte pointer to the first byte of the destination field,
- AC3 contains a 32-bit byte pointer to the first byte of the source field.

The fields may overlap in any way. However, the instruction processes characters one at a time, so unusual side effects may be produced by certain types of overlap.

The instruction maintains two flags and three indicators or pointers. The flags are the Significance Trigger (*T*) and the Sign flag (*S*). The three indicators are the Source Indicator (SI), the Destination Indicator (DI), and the op-code Pointer (P).

At the start of execution, the *Edit* instruction sets *T* to 0. When the instruction manipulates the first non-zero digit, it sets *T* to 1 (unless an edit op-code specifies otherwise).

The instruction sets *S* to reflect the sign of the number currently being processed. If the number is positive, the instruction sets *S* to 0. If the number is negative, the instruction sets *S* to 1.

Each of the three indicators is 16 bits wide and contains a byte pointer to the *current* byte in each respective area. At the start of execution, the *Edit* instruction sets SI to the value contained in AC3 (the starting address of the source string). It also sets DI to the value contained in AC2 (the starting address of the destination string), and P to the value contained in AC0 (a pointer to the first *Edit* opcode).

During execution, the subprogram can test and modify *S* and *T*, as well as modify SI, DI and P.

When execution begins, the instruction checks the sign of the source number for validity. If the sign is invalid, the instruction ends. If the sign is valid, execution continues with the *Edit* sub-program.

The sub-program is made up of 8-bit op-codes followed by one or more 8-bit operands. The byte pointer contained in P acts as the *program counter* for the subprogram. The subprogram proceeds sequentially until a branching operation occurs — much the same way programs are processed. Unless instructed to do otherwise, the *Edit* instruction

updates P after each operation to point to the next sequential op-code. The instruction continues to process 8-bit opcodes until directed to stop by the **DEND** op-code. Note that all 8-bit opcodes must be contained in the current segment.

Upon successful termination, carry contains *T*; AC0 contains P, which points to the next opcode to be processed; AC1 is undefined; AC2 contains DI, which points to the next destination byte; and AC3 contains SI, which points to the next source byte. The value of carry is indeterminate and *overflow* is 0.

> **NOTES:** *If SI references bytes not contained in the source number, then the instruction supplies zeroes for future manipulations. The instruction will use these zeroes for all subsequent operations, even if SI later references bytes contained by the source number.*
>
> *Opcodes that move numeric data may perform special actions. Opcodes that move non-numeric data copy characters exactly into the destination string.*
>
> *The* Edit *instruction places information on the wide stack. Therefore, the stack must be set up and have at least 16 words available for use.*
>
> *If an interrupt occurs during the* Edit *instruction, the instruction places restart information on the stack and in the accumulators and sets bit 2 of the PSR to 1.*
>
> *If bit 2 of the PSR contains a 1, then the* Edit *instruction assumes it is restarting from an interrupt. Make sure you do not set this bit under any other circumstances.*

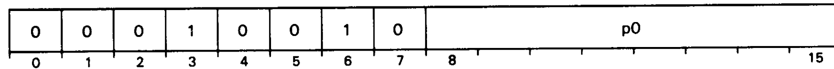
Many of the *Edit* opcodes use the symbol *j*. This symbol represents a number; when *j* is greater than or equal to zero, it specifies the number of characters the instruction should process. When *j* is less than zero, it represents a pointer into the wide stack. The pointer references a stack word that denotes the number of characters the instruction should process. The number on the stack is at address:

WSP + 2 + 2**j*.

An *Edit* operation that processes numeric data (e.g., **DMVN**) skips a leading or trailing sign code it encounters; similarly, such an operation converts a high-order or low-order sign to its correct numeric equivalent.

Add To DI

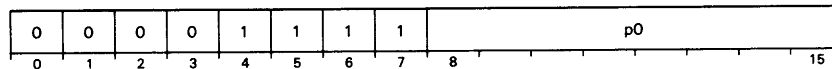
DADI *p0*



Adds the 8-bit two's complement integer specified by *p0* to the Destination Indicator (DI).

Add To P Depending On S

DAPS *p0*

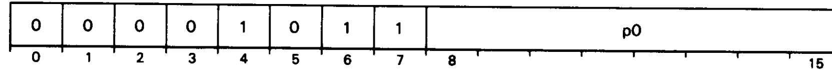


If *S* is 0, the instruction adds the 8-bit two's complement integer specified by *p0* to the op-code Pointer (P). Before the add is performed, P is pointing to the byte containing the

DAPS op-code.

Add To P Depending On T

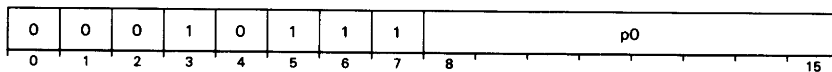
DAPT *p0*



If *T* is one, the instruction adds the 8-bit two's complement integer specified by *p0* to the op-code Pointer (P). Before the add is performed, P is pointing to the byte containing the **DAPT** op-code.

Add To P

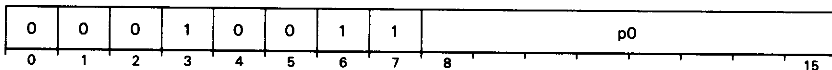
DAPU *p0*



Adds the 8-bit two's complement integer specified by *p0* to the op-code Pointer (P). Before the add is performed, P is pointing to the byte containing the **DAPU** op-code.

Add To SI

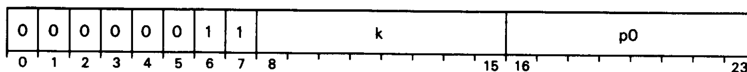
DASI *p0*



Adds the 8-bit two's complement integer specified by *p0* to the Source Indicator (SI).

Decrement And Jump If Non-Zero

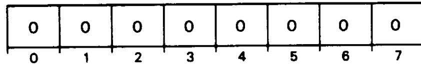
DDTK *k,p0*



Decrements a word in the stack by one. If the decremented value of the word is non-zero, the instruction adds the 8-bit two's complement integer specified by *p0* to the op-code Pointer (P). Before the add is performed, P is pointing to the byte containing the **DDTK** op-code. If the 8-bit two's complement integer specified by *k* is negative, the word decrement is at the address (WSP + 2 + (2*)*k)*. If *k* is positive, the word decremented is at the address (WFP + 2 + (2*)*k*).

End Edit

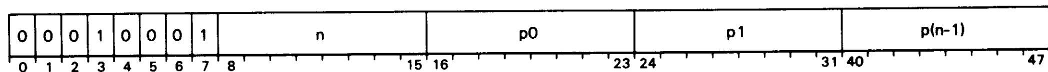
DEND



Terminates the EDIT sub-program.

Insert Characters Immediate

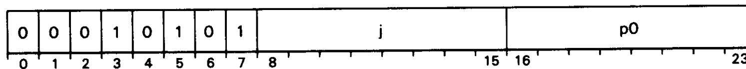
DICI *n,p0,p1,...,p(n-1)*



Inserts *n* characters from the op-code stream into the destination field beginning at the position specified by DI. Increases P by $(n+2)$, and increases DI by *n*.

Insert Character J Times

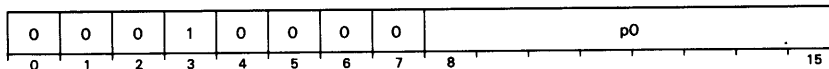
DIMC *j,p0*



Inserts the character specified by *p0* into the destination field a number of times equal to *j* beginning at the position specified by DI. Increases DI by *j*.

Insert Character Once

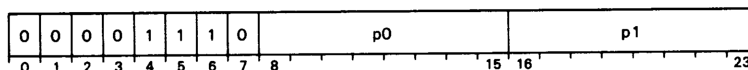
DINC *p0*



Inserts the character specified by *p0* in the destination field at the position specified by DI. Increments DI by 1.

Insert Sign

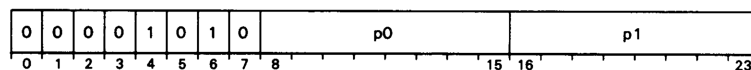
DINS *p0,p1*



If the Sign flag (S) is 0, the instruction inserts the character specified by *p0* in the destination field at the position specified by DI. If *S* is 1, the instruction inserts the character specified by *p1* in the destination field at the position specified by DI. Increments DI by 1.

Insert Character Suppress

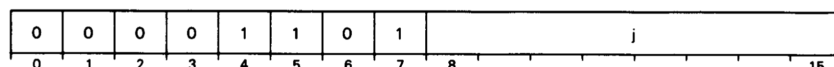
DINT *p0,p1*



If the significance Trigger *(T)* is 0, the instruction inserts the character specified by *p0* in the destination field at the position specified by DI. If *T* is 1, the instruction inserts the character specified by *p1* in the destination field at the position specified by DI. Increments DI by 1.

Move Alphabetics

DMVA *j*

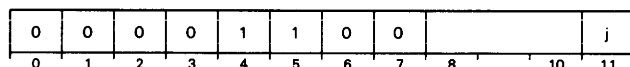


Moves *j* characters from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. Increases both SI and DI by *j*. Sets *T* to 1.

Initiates a commercial fault if the attribute specifier word indicates that the source field is data type 5 (packed). Initiates a commercial fault if any of the characters moved is not an alphabetic (A-Z, a-z, or space).

Move Characters

DMVC *j*

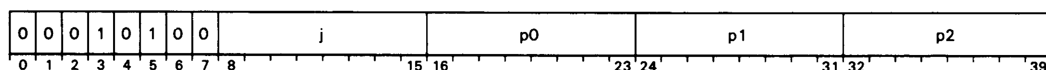


Increments SI if the source data type is 3 and $j>0$. The instruction then moves *j* characters from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. Increases both SI and DI by *j*. Sets *T* to 1.

Initiates a commercial fault if the attribute specifier word indicates that the source is data type 5 (packed). Performs no validation of the characters.

Move Float

DMVF *j,p0,p1,p2*

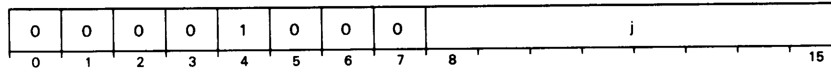


If the source data type is 3, $j>0$, and SI points to the sign of the source number, the instruction increments SI. Then for *j* characters, the instruction either places a digit substitute in the destination field beginning at the position specified by DI, or it moves a

digit from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. When *T* changes from 0 to 1, the instruction places both the digit substitute and the digit in the destination field, and compares *j* to the number of digits left to move. Increments SI by the smaller of the two values.

Move Numerics

DMVN *j*

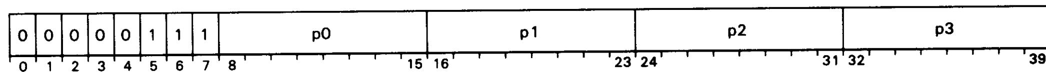


Increments SI if the source data type is 3 and $j>0$. Moves *j* characters from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. Increases DI by *j*. Compares *j* to the number of source characters left to move, and increments SI by the smaller of the two values. Sets *T* to 1.

Initiates a commercial fault if any of the characters moved is not valid for the specified data type.

Move Digit With Overpunch

DMVO *p0,p1,p2,p3*



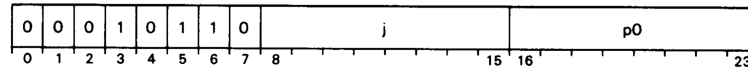
Increments SI if the source data type is 3 and SI points to the sign of the source number. The instruction then either places a digit substitute in the destination field at the position specified by DI, or it moves a digit plus overpunch the source field at the position specified by SI to the destination field at the position specified by DI. Increases DI by 1. Compares the number of digits left to move with 1 and increments SI by the smaller of the two values.

If the digit is a zero or space and *S* is 0, then the instruction places *p0* in the destination field. If the digit is a zero or space and *S* is 1, then the instruction places *p1* in the destination field. If the digit is a non-zero and *S* is 0, the instruction adds *p2* to the digit and places the result in the destination field. If the digit is a non-zero and *S* is 1, the instruction adds *p3* to the digit and places the result in the destination field. If the digit is a non-zero the instruction sets *T* to 1. The instruction assumes *p2* and *p3* are ASCII characters.

The instruction initiates a commercial fault if the character is not valid for the specified data type.

Move Numeric With Zero Suppression

DMVS *j,p0*

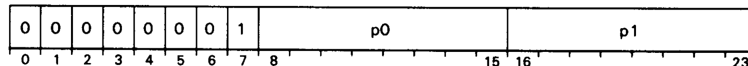


Increments SI if the source data type is 3 and *j*>0, and SI points to the sign of the source number. The instruction then moves *j* characters from the source field beginning at the position specified by SI to the destination field beginning at the position specified by DI. Moves the digit from the source to the destination if *T* is 1. Replaces all zeros and spaces with *p0* as long as *T* is 0. Sets *T* to 1 when the first non-zero digit is encountered. Increases DI by *j*. Compares *j* to the number of source characters left to move, and increments SI by the smaller of the two values.

Initiates a commercial fault if any of the characters moved is not a numeric (0-9 or space).

End Float

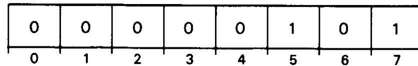
DNDF *p0,p1*



If *T* is 1, the instruction places nothing in the destination field and leaves DI unchanged. If *T* is 0 and *S* is 0, the instruction places *p0* in the destination field at the position specified by DI. If *T* is 0 and *S* is 1, the instruction places *p1* in the destination field at the position specified by DI. Increases DI by 1, and sets *T* to 1.

Set S To One

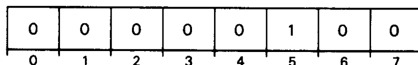
DSSO



Sets the Sign flag *(S)* to 1.

Set S To Zero

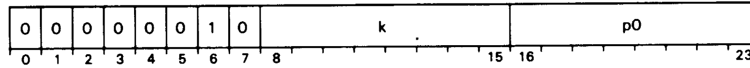
DSSZ



Sets the Sign flag *(S)* to 0.

Store In Stack

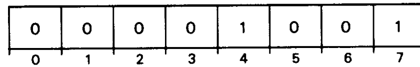
DSTK *k,p0*



Stores the byte specified by *p0* in bits 24-31 of a word in the wide stack. Sets bits 0-23 of the word that receives *p0* to 0. If the 8-bit two's complement integer specified by *k* is negative, the instruction addresses the word receiving *p0* by (WSP + 2 + (2*)*k*). If *k* is positive then the instruction stores *p0* at the address (WFP + 2 + (2*)*k*).

Set T To One

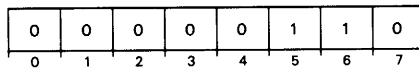
DSTO



Sets the significance Trigger *(T)* to 1.

Set T To Zero

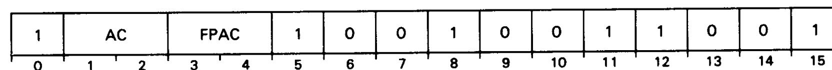
DSTZ



Sets the significance Trigger *(T)* to 0.

Wide Fix from Floating Point Accumulator

WFFAD *ac,fpac*



Converts the integer portion of the floating point number contained in the specified FPAC to a 32-bit, signed, two's complement integer. Places the result in an accumulator.

If the integer portion of the number contained in FPAC is less than -2,147,483,649 or greater than +2,147,483,648, the instruction sets *MOF* in the FPSR to 1. Takes the absolute value of the integer portion of the number contained in the FPAC. Takes the 31 least significant bits of the absolute value and appends a 0 onto the leftmost bit to give a 32-bit number. If the sign of the number is negative, formes the two's complement of the 32-bit result. Places the 32-bit integer in the specified accumulator.

If the integer portion is within the range of -2,147,483,648 to +2,147,483,647 inclusive, the instruction places the 32-bit, two's complement of the integer portion of the number contained in the FPAC in the specified accumulator.

The instruction leaves the FPAC and the *Z* and *N* flags of the FPSR unchanged.

Wide Float from Fixed Point Accumulator

WFLAD *ac,fpac*

1	AC		FPAC		1	0	0	1	0	1	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Converts the contents of a 32-bit accumulator to floating point format and places the result in a specified FPAC.

Converts the 32-bit, signed, two's complement number contained in the specified accumulator to a double precision floating point number. Places the result in the specified FPAC. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of the FPAC.

The range of numbers that can be converted is -2,147,483,648 to +2,147,483,647 inclusive.

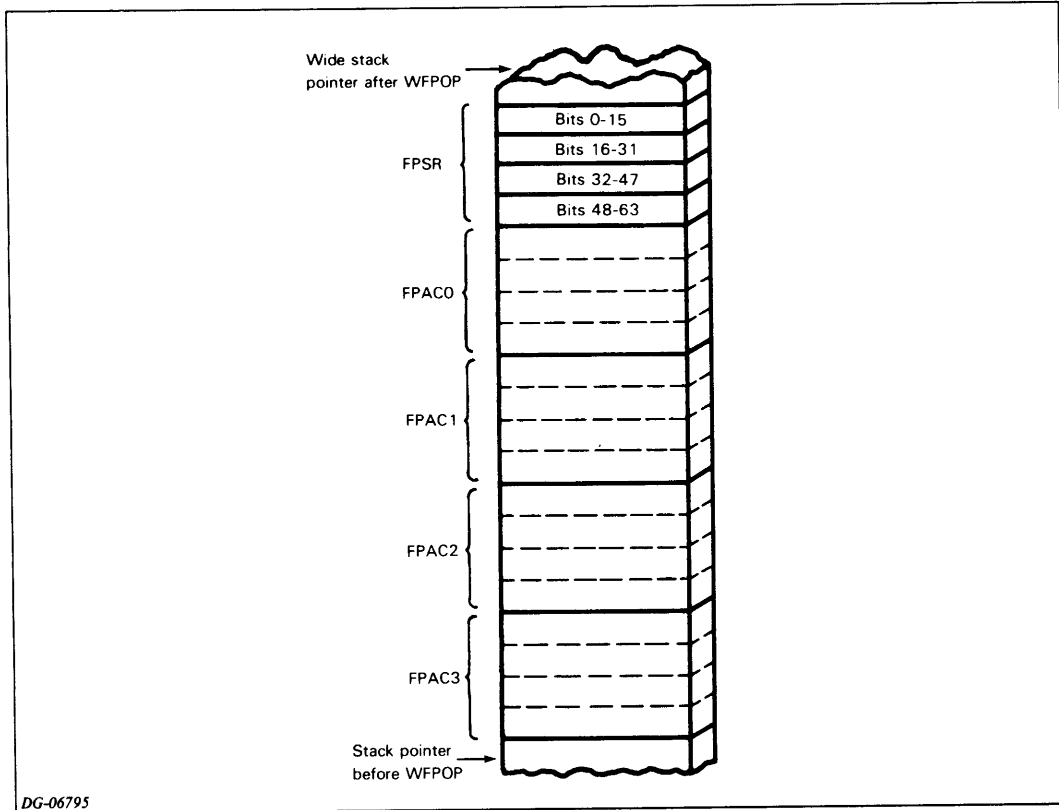
Wide Floating Point Pop

WFPOP

1	0	1	0	0	1	1	1	1	0	0	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pops the state of the floating point unit off the wide stack.

Pops a 20-word block off the wide stack and loads the contents into the FPSR and the four FPACs. The format of the 20-word block is shown below.



This instruction loads the FPSR as follows:

- Places bits 0-15 of the operand in bits 0-15 of the FPSR. Sets bits 16-32 of the FPSR to 0.
- If *ANY* is 0, bits 33-63 of the FPSR are undefined.
- If *ANY* is 1, the instruction places the value of the current segment in bits 33-35 of the FPSR, zeroes in bits 36-48, and bits 17-31 of the operand in bits 49-63 of the FPSR.

NOTES: *This instruction moves unnormalized data without change.*

This instruction does not set the ANY *flag from memory. If any of bits 1-4 are loaded as 1,* ANY *is set to 1; otherwise,* ANY *is 0.*

Bits 12-15 of the FPSR are not set from memory. These bits are the floating point identification code an are read protected. In the MV/8000 they are set to 0111.

This instruction does not initiate a floating point trap under any conditions of the FPSR.

See Chapter 8 and Appendix G for more information about floating point manipulation.

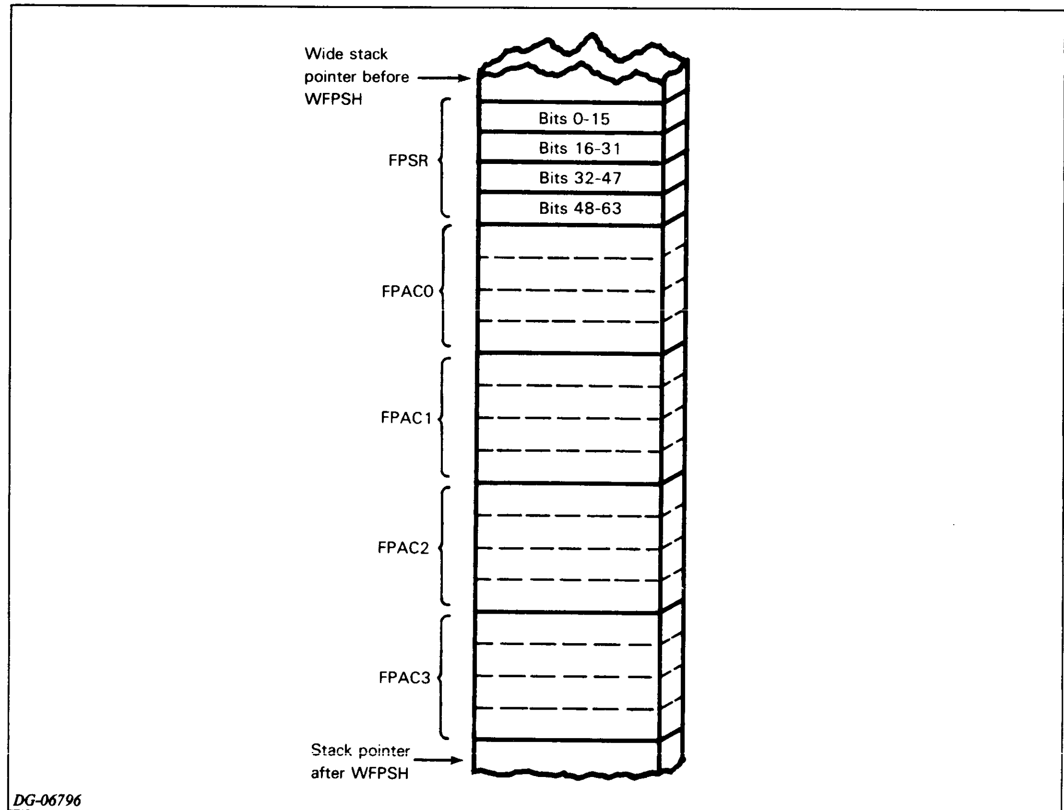
Wide Floating Point Push

WFPSH

1	0	0	0	0	1	1	1	1	0	1	1	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pushes the state of the floating point unit onto the wide stack.

Pushes a 20-word block onto the wide stack. The block contains the contents of the FPSR and the contents of the four FPACs, as shown in the figure below:



The instruction pushes the FPSR onto the stack as follows:

- Stores bits 0-15 of the FPSR in the first memory word.
- Sets bits 16-31 of the first memory double word and bit 0 of the second memory double word to 0.
- If *ANY* is 0, the contents of bits 1-31 of the second memory double word are undefined.
- If *ANY* is 1, the instruction stores bits 33-63 of the FPSR into bits 1-31 of the second memory double word.

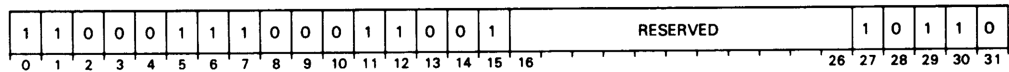
The rest of the block is pushed onto the stack after the FPSR has been pushed.

NOTES: *This instruction moves unnormalized data without change.*

This instruction does not initiate a floating point trap under any conditions of the FPSR.

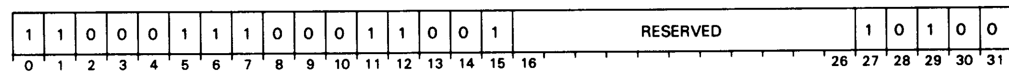
See Chapter 8 and Appendix G for more information about floating point manipulation.

Wide Search Queue Forward
WFSAC



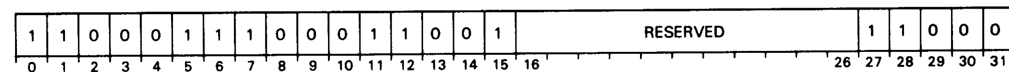
See instruction entry "Search Queue".

Wide Search Queue Forward
WFSAS



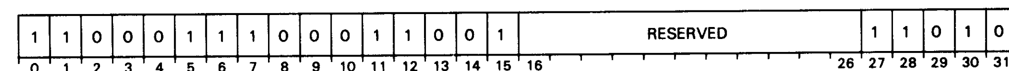
See instruction entry "Search Queue".

Wide Search Queue Forward
WFSE



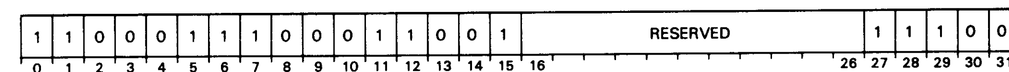
See instruction entry "Search Queue".

Wide Search Queue Forward
WFSGE



See instruction entry "Search Queue".

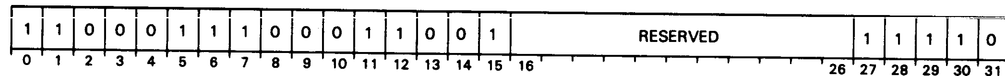
Wide Search Queue Forward
WFSLE



See instruction entry "Search Queue".

Wide Search Queue Forward

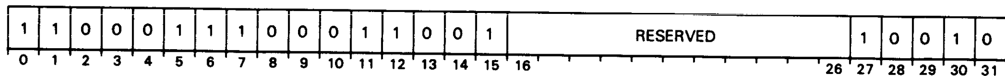
WFSNE



See instruction entry "Search Queue".

Wide Search Queue Forward

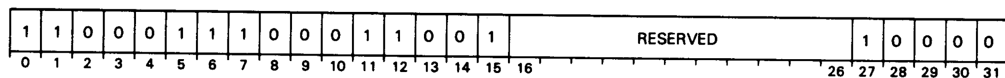
WFSSC



See instruction entry "Search Queue".

Wide Search Queue Forward

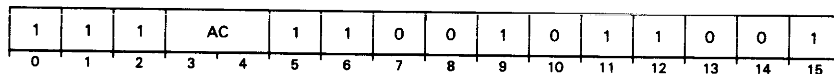
WFSSS



See instruction entry "Search Queue".

Wide Halve

WHLV *ac*



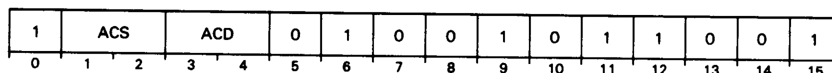
Divides the 32-bit contents of the specified accumulator by 2 and rounds the result toward 0.

The signed, 32-bit two's complement number contained in the specified accumulators divided by 2 and rounded toward 0. The result is placed in the specified accumulator.

This instruction leaves carry unchanged; *overflow* is 0.

Wide Increment

WINC *acs,acd*

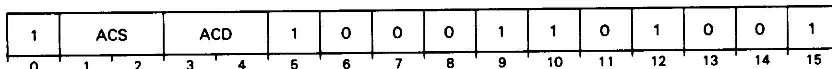


Increments an integer contained in an accumulator.

The instruction increments the 32-bit contents of ACS by 1 and loads the result into ACD. Sets carry to the value of ALU carry. Sets *overflow* to 1 if there is an ALU overflow. The contents of ACS remain unchanged, unless ACS equals ACD.

Wide Inclusive OR

WIOR *acs,acd*

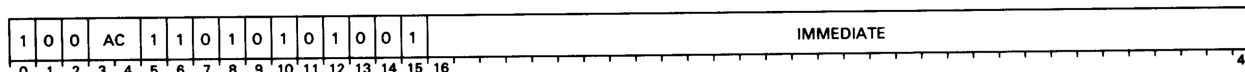


Performs an inclusive OR between two accumulators.

Forms the logical inclusive OR between corresponding bits of ACS and ACD. Loads the 32-bit result into ACD. The contents of ACS remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Inclusive OR Immediate

WIORI *ac, immediate*



Performs an inclusive OR.

The instruction forms the logical inclusive OR between corresponding bits of the specified accumulator and the value contained in the literal field. The instruction places the result of the inclusive OR in the specified accumulator. Carry is unchanged and *overflow* is 0.

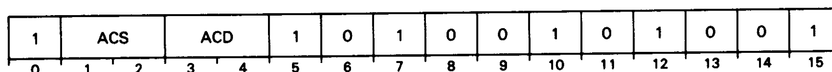
Wide Load with Wide Immediate

WLDAI *ac,immediate*

Loads the 32-bit value contained in the immediate field into the specified accumulator. Carry is unchanged and *overflow* is 0.

Wide Load Byte

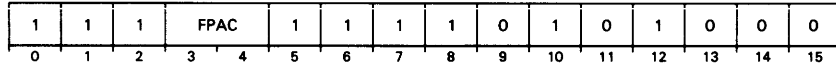
WLDB *acs,acd*



Uses the 32-bit byte address contained in ACS to load a byte into ACD. Sets bits 0–23 of ACD to zero. Bits 24–31 of ACD contain a copy of the contents of the addressed byte. The contents of ACS remain unchanged, unless ACS and ACD are the same accumulator. Carry is unchanged and *overflow* is 0.

Wide Load Integer

WLDI *fpac*



Translates a decimal integer from memory to floating point format and places the result in a floating point accumulator.

AC1 must contain the data-type indicator describing the integer.

AC3 must contain a 32-bit byte pointer pointing to the high-order byte of the integer in memory.

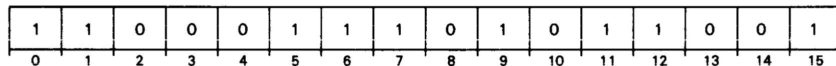
Uses AC1 and AC3 to convert a decimal integer to floating point form. Normalizes the result and places it in the specified FPAC. Updates the *Z* and *N* flags in the FPSR to describe the new contents of the specified FPAC. Leaves the decimal number unchanged in memory.

By convention, the first byte of a number stored according to data type 7 contains the sign and exponent of the floating point number. The instruction copies each byte (following the lead byte) directly to the mantissa of the specified FPAC. It then sets to zero each low-order byte in the FPAC that does not receive data from memory.

Upon successful completion, AC0 and AC1 remain unchanged. AC2 contains the original contents of AC3. AC3 points to the first byte following the integer field. Carry is unchanged and *overflow* is 0.

Wide Load Integer Extended

WLDIX



Distributes a decimal integer of data type 0, 1, 2, 3, 4, or 5 into the four FPACs.

AC1 must contain the data-type indicator describing the integer.

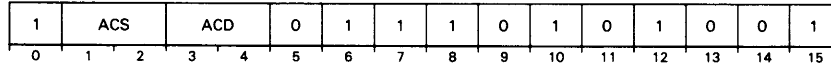
AC3 must contain a 32-bit byte pointer which is the address of the high-order byte of the integer.

The instruction uses the contents of AC3 to reference the integer. Extends the integer with high-order zeros until it is 32 digits long. Divides the integer into 4 units of 8 digits each and converts each unit to a floating point number. Places the number obtained from the 8 high-order digits into FAC0. Places the number obtained from the next 8 digits into FAC1. Places the number obtained from the next 8 digits into FAC2. Places the number obtained from the low-order 8 bits into FAC3. Sets the sign of each FPAC by checking the number just loaded into the FPAC. If the FPAC contains a nonzero number, then sets the sign of the FPAC to be the sign of the integer. If the FPAC contains an 8-digit zero, sets the FPAC to true zero. The *Z* and *N* flags in the floating point status register are unpredictable.

Upon successful termination, the contents of AC0 and AC1 remain unchanged. AC2 contains the original contents of AC3. AC3 points to the first byte following the integer field. Carry is unchanged and *overflow* is 0.

Wide Locate Lead Bit

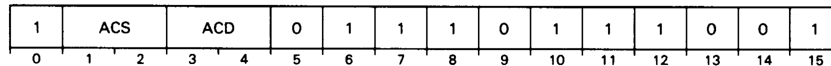
WLOB *acs,acd*



Counts the number of high-order zeroes in ACS. Adds the count of high-order zeroes to the 32-bit, signed contents of ACD. Stores the result of the add in ACD. The contents of ACS remain unchanged, unless ACS and ACD are the same accumulator. Carry is unchanged and *overflow* is 0.

Wide Locate and Reset Lead Bit

WLRB *acs,acd*



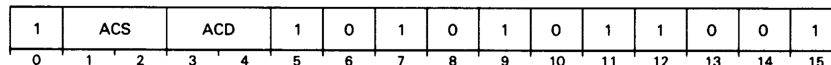
Counts the number of high-order zeroes in ACS.

The instruction counts the high order zeroes in ACS. Adds the count of high-order zeroes to the 32-bit, signed contents of ACD. Stores the result in ACD. Sets the leading bit of ACS to 0. Carry is unchanged and *overflow* is 0.

If ACS equals ACD, then sets the leading bit to 0 and adds nothing to the contents of the specified accumulator.

Wide Logical Shift

WLSH *acs,acd*



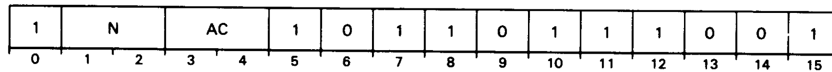
Shifts the 32-bit contents of ACD either left or right.

Bits 24–31 of ACS specify the number of bits to shift ACD. If this number is positive, then the instruction shifts the contents of ACD the appropriate number of bits to the left. If this number is negative, then the instruction shifts the contents of ACD the appropriate number of bits to the right. If ACS contains zero, then no shifting occurs. The instruction ignores bits 0–23 of ACS.

Bits shifted out during this instruction are lost. Zeroes fill the vacated bit positions. The contents of ACS remain unchanged, unless ACD equals ACS. Carry is unchanged and *overflow* is 0.

Wide Logical Shift Immediate

WLSI *n,ac*

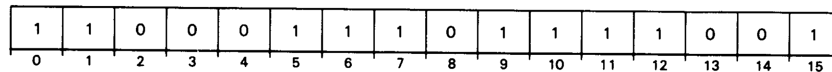


Shifts the contents of the specified accumulator to the left $n+1$ positions, where n is in the range 0 to 3. Carry is unchanged and *overflow* is 0.

> **NOTE:** *The assembler takes the coded value of* n *and subtracts one from it before placing it in the immediate field. Therefore, the programmer should code the exact value that he wishes to shift.*

Wide Load Sign

WLSN



Evaluates a decimal number as zero or nonzero, and the sign as positive or negative.

AC1 must contain the data type indicator describing the number.

AC3 must contain a byte pointer which is the address of the high-order byte of the number.

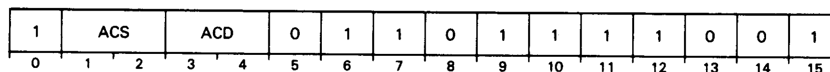
The instruction evaluates a decimal number in memory and returns in AC1 a code that classifies the number as zero or nonzero and identifies its sign. The meaning of the returned code is as follows:

Value of Number	Code
Positive non-zero	+1
Negative non-zero	-1
Positive zero	0
Negative zero	-2

Upon successful termination, the contents of AC0 remain unchanged; AC1 contains the value code; AC2 contains the original contents of AC3; and the contents of AC3 are unpredictable. The contents of the addressed memory locations remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Move

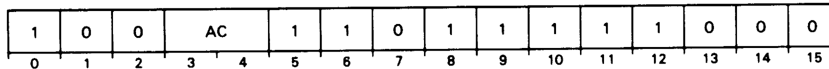
WMOV *acs,acd*



Moves a copy of the 32-bit contents of ACS into ACD. The contents of ACS remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Modify Stack Pointer

WMSP *ac*



Changes the value of the stack pointer and tests for potential overflow.

Shifts the contents of the specified accumulator left one bit. Adds the shifted value to the contents of the WSP and temporarily saves the result. Checks for fixed point overflow. If overflow occurs, the processor does not alter WSP and treats the overflow as a stack fault. AC1 contains the code 1.

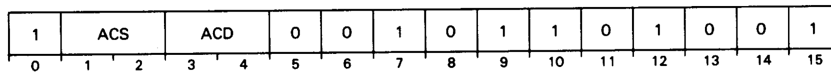
If no overflow occurs, the instruction checks the value of the result. If the result is positive, the processor checks it against the stack limit for stack overflow; if negative, against the stack limit for stack underflow. If underflow or overflow does not occur, the instruction loads WSP with the saved value.

If either overflow or underflow occurs, the instruction does not alter WSP and a stack fault occurs. AC1 contains the code 1. The PC in the return block points to this instruction.

This instruction does not change carry; *overflow* is 0.

Wide Multiply

WMUL *acs,acd*



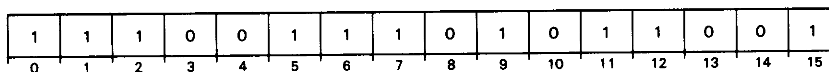
Multiplies two integers contained in accumulators.

The instruction multiplies the 32-bit, signed integer contained in ACD by the 32-bit, signed integer contained in ACS. Places the 32 least significant bits of the result in ACD. The contents of ACS and carry remain unchanged. *Overflow* is 0.

If the result is outside the range of -2,147,483,648 to +2,147,483,647 inclusive, sets *overflow* to 1; otherwise, *overflow* is 0. ACD will contain the 32 least significant bits of the result.

Wide Signed Multiply

WMULS

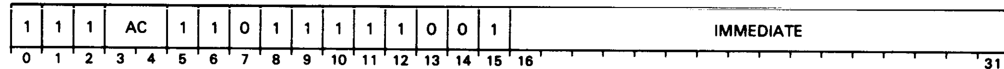


Multiplies two integers contained in accumulators.

The instruction multiplies the 32-bit, signed integer contained in AC1 by the 32-bit, signed integer contained in AC2. Adds the 32-bit signed integer contained in AC0 to the 64-bit result. Loads the 64-bit result into AC0 and AC1. AC0 contains the 32 high-order bits. AC2 and carry remain unchanged. *Overflow* is 0.

Wide Add with Narrow Immediate

WNADI *ac,immediate*

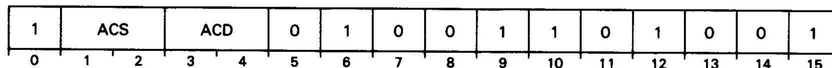


Adds an immediate value to an integer contained in an accumulator.

The instruction sign extends the two's complement literal value contained in the immediate field to 32 bits. Adds the sign extended value to the 32-bit integer contained in the specified accumulator. Loads the result into the specified accumulator. Sets carry to the value of ALU carry. Sets *overflow* to 1 if there is an ALU overflow.

Wide Negate

WNEG *acs,acd*

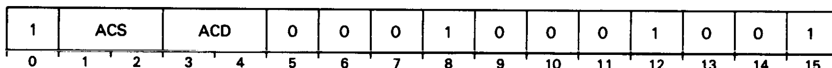


Negates the contents of an accumulator.

The instruction forms the two's complement of the 32-bit contents of ACS. Loads the result into ACD. Sets carry to the value of ALU carry. Sets *overflow* to 1 if there is an ALU overflow. The contents of ACS remain unchanged, unless ACS equals ACD.

Wide Pop Accumulators

WPOP *acs,acd*

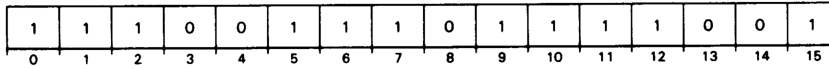


Pops up to 4 double words off the top of the wide stack and places them in the specified 32-bit accumulators.

Pops the top double word off the wide stack and places it in ACS. Pops the next double word off the wide stack and places it in ACS-1, and so on, until all specified accumulators have been loaded. If necessary, the accumulators wrap around, with AC3 following AC0, until all specified accumulators have been loaded. If ACS equals ACD, then the instruction pops only one double word off of the wide stack and places it in the specified accumulator.

The instruction decrements the contents of WSP by twice the number of double words popped. Carry is unchanged and *overflow* is 0.

Wide Pop Block
WPOPB



1	1	1	0	0	1	1	1	0	1	1	1	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Pops six double words off the wide stack and places them in the appropriate locations.

The popped words and their destinations are as follows:

Double Word Popped	Destination
1	Bit 0 to carry; bits 1-31 to PC
2	AC3
3	AC2
4	AC1
5	AC0
6	Bit 0 to OVK; bit 1 to OVR; bit 2 to IRES; bits 17-31 are multiplied by 2 and incremented by 12. This number is subtracted from WSP. WSP is loaded with the result.

If the instruction specifies an inward ring crossing, then a protection fault occurs and the current wide stack remains unchanged. Note that the return block pushed as a result of the protection fault will contain undefined information. After the fault return block is pushed, AC0 contains the contents of the PC (which point to the instruction that caused the fault) and AC1 contains the code 8.

If the instruction specifies an intra-ring address, it pops the six-double-word block, then checks for stack underflow. If underflow has occurred, a stack underflow fault occurs. Note that the return block pushed as a result of the stack underflow will contain undefined information. After the fault return block is pushed, AC0 contains the contents of the PC (which point to the instruction that caused the fault) and AC1 contains the code 3. If there is no underflow, execution continues with the location addressed by the program counter.

If the instruction specifies an outward ring crossing, it pops the six-double-word return block and checks for stack underflow. If underflow has occurred, a stack underflow fault occurs. Note that the return block pushed as a result of the stack underflow will contain undefined information. After the fault return block is pushed, AC0 contains the contents of the PC (which point to the instruction that caused the fault) and AC1 contains the code 3. If there is no underflow, the instruction stores WSP and WFP in the appropriate page zero locations of the current segment. It then performs the outward ring crossing and loads the wide stack registers with the contents of the appropriate page zero locations of the new ring. Loads WSP with the value:

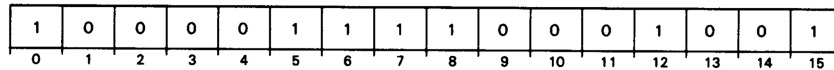
(*current contents of WSP*) - (2 x (*argument count*))

Checks for stack underflow. If underflow has occurred, a stack underflow fault occurs. Note that the return block pushed as a result of the stack underflow will contain undefined information. After the fault return block is pushed, AC0 contains the contents of the PC (which point to the instruction that caused the fault) and AC1 contains the code 3. If there is no underflow, execution continues with the location addressed by the

program counter.

Pop PC and Jump

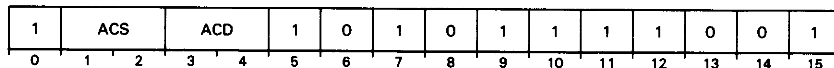
WPOPJ



Pops the top 31-bit value off the wide stack, loads it into the PC, then checks for stack overflow. Carry is unchanged and *overflow* is 0.

Push Accumulators

WPSH *acs,acd*



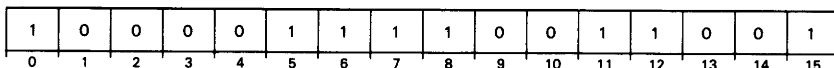
Pushes the contents of the specified 32-bit accumulators onto the top of the wide stack.

Pushes the contents of ACS onto the top of the wide stack, then pushes the contents of next sequential accumulators up to and including ACD. If necessary, the accumulators wrap around, with AC0 following AC3, until the contents of all specified accumulators have been pushed. If ACS equals ACD, then the instruction pushes the contents of only one accumulator onto the wide stack.

Note that the instruction increments the contents of WSP by two times the number of accumulators pushed (32-bit accumulators). Carry is unchanged and *overflow* is 0.

Wide Restore

WRSTR



Returns control from an interrupt.

When this instruction is used, the wide stack should contain the following information, in the given order:

Contents	Size of Word	Notes
WFP	(32 bits)	
WSP	(32 bits)	
WSL	(32 bits)	
WSB	(32 bits)	
SFA	(Lower 16 bits)	< Stack fault address
OVK, OVR	(Bits 0 and 1)	
AC0	(32 bits)	
AC1	(32 bits)	
AC2	(32 bits)	
AC3	(32 bits)	
Carry, PC	(32 bits)	This is the top of the wide stack.

The instruction checks to see if the ring crossing specified is inward. If the crossing is inward, a protection fault occurs (code=8 in AC1).

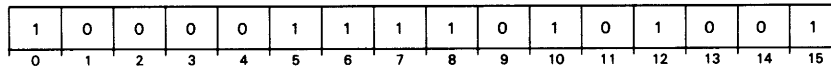
If the crossing is not inward, the instruction pops the return block on top of the wide stack and places the block contents in the appropriate registers. Next, the instruction pops the stack registers and the stack fault address, temporarily saves them, and checks for stack underflow. If no underflow occurs, further actions depend upon the type of ring call.

If the restore is to be to the same ring, the instruction places the temporarily saved stack management information in the four stack registers. Stores the stack fault address in location 14_8 of the current segment. Checks for stack underflow. If underflow has occurred, a stack underflow fault occurs (code=3 in AC1). If underflow has not occurred, execution continues with the location specified by the PC.

If the ring crossing is outward, the instruction stores the stack management information held internally into the appropriate page zero locations of the current segment. Performs the outward ring crossing. Loads the stack registers with the contents of the appropriate page zero locations of the new segment. Checks for stack underflow. If underflow has occurred, a stack underflow fault occurs (code=3 in AC1). If underflow has not occurred, execution continues with the location specified by the PC.

Wide Return

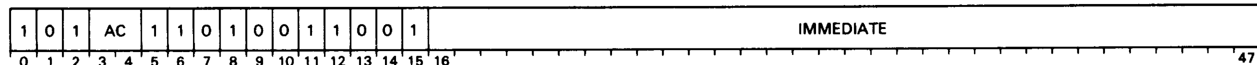
WRTN



Returns control from subroutines that issue a **WSAVS** or a **WSAVR** instruction at their entry point. Places the contents of WFP in WSP and executes a **WPOPB** instruction. Places the popped value of AC3 in WFP.

Wide Skip on All Bits Set in Accumulator

WSALA *ac,immediate*



Performs a logical AND between an immediate value and the contents of an accumulator. Skips depending on the result of the AND.

The instruction performs a logical AND on the contents of the immediate field and the complement of the contents of the specified accumulator. If the result of the AND is zero, then execution skips the next sequential word before continuing. If the result of the AND is nonzero, then execution continues with the next sequential word. The contents of the specified accumulator remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Skip on All Bits Set in Double-word Memory Location

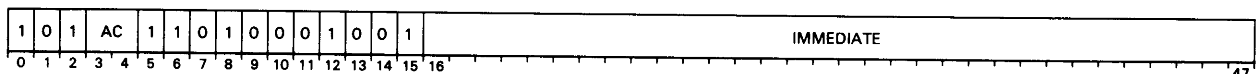
WSALM *ac,immediate*

Performs a logical AND between an immediate value and the complement of a memory word. Skips depending on the result of the AND.

The instruction performs a logical AND on the contents of the immediate field and the complement of the double word addressed by the specified accumulator. If the result of the AND is zero, then execution skips the next sequential word before continuing. If the result of the AND is nonzero, then execution continues with the next sequential word. The contents of the specified accumulator and memory location remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Skip on Any Bit Set in Accumulator

WSANA *ac,immediate*



Performs a logical AND between an immediate value and the contents of an accumulator. Skips depending on the result of the AND.

The instruction performs a logical AND on the contents of the immediate field and the contents of the specified accumulator. If the result of the AND is nonzero, then execution skips the next sequential word before continuing. If the result of the AND is zero, then execution continues with the next sequential word. The contents of the specified accumulator remain unchanged. Carry is unchanged and *overflow* is 0.

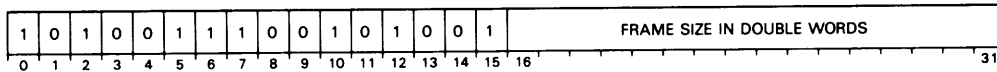
Wide Skip on Any Bit Set in Double-word Memory Location

WSANM *ac,immediate*

Performs a logical AND between an immediate value and the contents of a memory word. Skips depending on the result of the AND.

The instruction performs a logical AND on the contents of the immediate field and the contents of the double word addressed by the specified accumulator. If the result of the AND is nonzero, then execution skips the next sequential word before continuing. If the result of the AND is zero, then execution continues with the next sequential word. The contents of the specified accumulator and memory location remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Save/Reset Overflow Mask
WSAVR

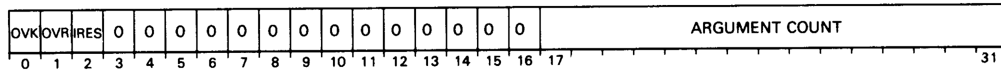


Pushes a return block onto the wide stack and resets **OVK**.

The instruction checks for stack overflow. If an overflow would occur, then control transfers to the wide stack fault routine. If no overflow would occur, then the instruction pushes five double words of a wide six-double word return block onto the wide stack. The words pushed have the following contents:

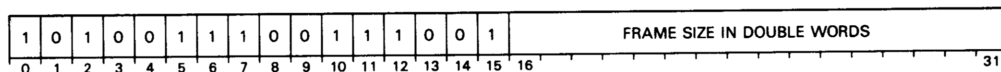
Double Word Pushed	Contents
1	AC0
2	AC1
3	AC2
4	AC3
5	carry and PC

Note that the five words described above do not make up the entire return block. Either the **LCALL** or the **XCALL** instruction pushes the first double word of the return block onto the wide stack. This word has the following format:



After pushing the return block, the instruction places the value of the stack pointer in WFP and AC3. Multiplies the 16-bit, unsigned integer contained in the second instruction word by 2. Adds the result to WSP. Sets **OVK** to 0, disabling integer overflow.

Wide Save/Set Overflow Mask
WSAVS

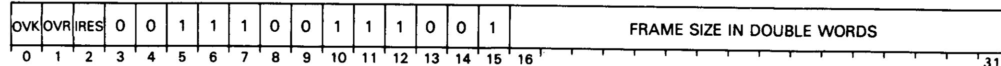


Pushes a return block onto the wide stack, resets WSP and WFP, and sets **OVK** to 1.

The instruction checks for stack overflow. If an overflow would occur, then control transfers to the wide stack fault routine. If no overflow would occur, then the instruction pushes five double words of a wide six-double word return block onto the stack. The words pushed have the following contents:

Double Word Pushed	Contents
1	AC0
2	AC1
3	AC2
4	AC3
5	carry and PC

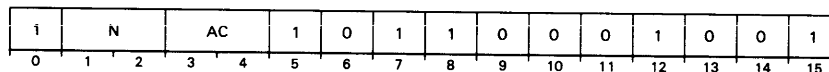
Note that the five double words described above do not make up the entire return block. Either the **LCALL** or the **XCALL** instruction pushes the first double word of the return block onto the wide stack. This word has the following format:



After pushing the return block, the instruction places the value of WSP in WFP and AC3. Multiplies the 16-bit, unsigned integer contained in the second instruction word by 2. Adds the result to WSP. Sets **OVK** to 1, enabling integer overflow.

Wide Subtract Immediate

WSBI *n,ac*



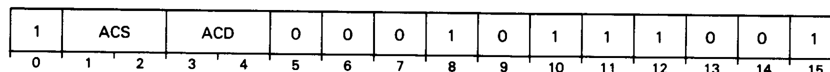
Subtracts an integer in the range 1 to 4 from an integer contained in an accumulator.

The instruction subtracts the value $n+1$ from the value contained in the specified accumulator. Stores the result in the specified accumulator. Sets carry to the value of ALU carry. Sets *overflow* to 1 if there is an ALU overflow.

> **NOTE:** *The assembler takes the coded value of* n *and subtracts 1 from it before placing it in the immediate field. Therefore, the programmer should code the exact value that he wishes to subtract.*

Wide Skip If Equal To

WSEQ *acs,acd*



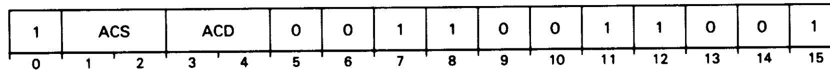
Compares one integer to another and skips if the two integers are equal. Carry is unchanged and *overflow* is 0.

The instruction compares the 32-bit integer contained in ACS to the 32-bit integer in ACD. If the integer contained in ACS is equal to the integer contained in ACD, the next 16-bit word is skipped; otherwise, the next word is executed.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to zero. The skip will occur if the integer equals zero.

Wide Signed Skip If Greater Than Or Equal To

WSGE *acs,acd*



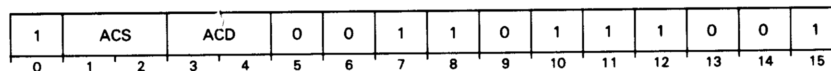
Compares one integer to another and skips if the first is greater than or equal to the second. Carry is unchanged and *overflow* is 0.

The instruction compares the signed, 32-bit integer contained in ACS to the signed, 32-bit integer in ACD. If the integer contained in ACS is greater than or equal to the integer contained in ACD, then the next word is skipped; otherwise, the next instruction is executed.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to zero. The skip will occur if the integer is greater than or equal to zero.

Wide Signed Skip If Greater Than

WSGT *acs,acd*



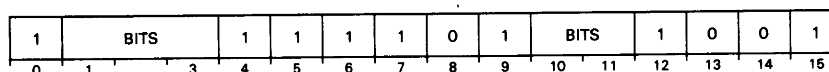
Compares one integer to another and skips if the first is greater than the second. Carry is unchanged and *overflow* is 0.

The instruction compares the signed, 32-bit integer contained in ACS to the signed 32-bit integer in ACD. If the integer contained in ACS is greater than the integer contained in ACD, the next word is skipped; otherwise, the next word is executed.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to zero. The skip will occur if the integer is greater than zero.

Wide Skip on Bit Set to One

WSKBO *bit number*



Tests a specified bit in AC0 and skips if the bit is one.

The instruction uses the bits specified in bits 1–3 and 10–11 to specify a bit position in the range 0–31. This number specifies one bit in AC0; the value 0 specifies the highest-order bit, and the value 31 specifies the lowest-order bit. If the specified bit has the value 1, then the next sequential word is skipped. If the bit has the value 0, then the

next sequential word is executed. The contents of AC0 remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Skip on Bit Set to Zero

WSKBZ *bit number*

1	BITS	1	1	1	1	1	0	BITS	1	0	0	1
0	1 3	4	5	6	7	8	9	10 11	12	13	14	15

Tests a specified bit in AC0 and skips if the bit is 0.

The instruction uses the bits specified in bits 1–3 and 10–11 to specify an a bit position in the range 0–31. This number specifies one bit in AC0; the value 0 specifies the highest-order bit, and the value 31 specifies the lowest-order bit. If the specified bit has the value 0, then the next sequential word is skipped. If the bit has the value 1, then the next sequential word is executed. The contents of AC0 remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Signed Skip If Less Than Or Equal To

WSLE *acs,acd*

1	ACS	ACD	0	0	1	1	0	1	0	1	0	0	1
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Compares one integer to another and skips if the first is less than or equal to the second. Carry is unchanged and *overflow* is 0.

The instruction compares the signed, 32-bit integer contained in ACS to the signed, 32-bit integer in ACD. If the integer contained in ACS is less than or equal to the integer contained in ACD, the next word is skipped; otherwise, the next sequential word is executed.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to zero. The skip will occur if the integer is less than or equal to zero.

Wide Signed Skip If Less Than

WSLT *acs,acd*

1	ACS	ACD	0	1	0	1	0	0	0	1	0	0	1
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

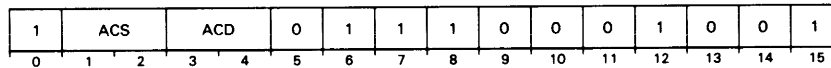
Compares one integer to another and skips if the first is less than the second. Carry is unchanged and *overflow* is 0.

The instruction compares the signed, 32-bit integer contained in ACS to the signed, 32-bit integer in ACD. If the integer contained in ACS is less than the integer contained in ACD, the next word is skipped; otherwise, the next sequential word is executed.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to zero. The skip will occur if the integer is less than zero.

Wide Skip on Nonzero Bit

WSNB *acs,acd*



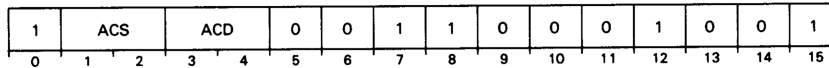
Tests the value of an addressed bit and skips if the bit is one. Carry is unchanged and *overflow* is 0.

The instruction forms a bit pointer from the contents of ACS and ACD. ACS contains the high-order bits of the bit pointer; ACD contains the low-order bits. ACS and ACD can be specified to be the same accumulator; in this case, the specified accumulator supplies the low-order bits of the bit pointer. The high-order bits are treated as if they were zero in the current segment.

The instruction checks the value of the bit referenced by the bit pointer. If the bit has the value 1, the next sequential word is skipped. If the bit has the value 0, the next sequential instruction is executed.

Wide Skip If Not Equal To

WSNE *acs,acd*



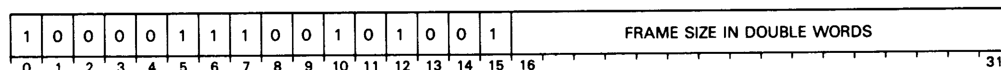
Compares one integer to another and skips if the two are not equal. Carry is unchanged and *overflow* is 0.

The instruction compares the 32-bit integer contained in ACS to the 32-bit integer in ACD. If the integer contained in ACS is not equal to the integer contained in ACD, then execution skips the next word; otherwise, execution proceeds with the next sequential word.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to zero. The skip will occur if the integer does not equal zero.

Wide Special Save/Set Overflow Mask

WSSVR



Pushes a wide return block onto the wide stack and sets **OVK** to 0.

The instruction checks for stack overflow. If executing the instruction would cause an overflow, the instruction transfers control to the wide stack fault handler. The PC in the fault return block will contain the address of the **WSSVR** instruction.

Pushes a wide return block onto the wide stack. After pushing the sixth double word, places the value of WSP in WFP and AC3. Increments WSP by twice the frame size. The frame size is a 16-bit, unsigned integer contained in the second word of this instruction. Sets **OVK** to 0, which disables integer overflow. Sets **OVR** to 0.

The structure of the wide return block pushed is as follows:

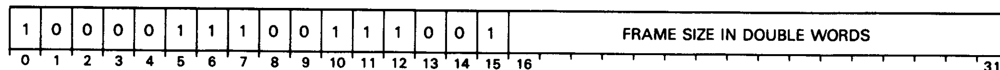
Word in Block	Contents
1-2	OVK, OVR, IRES, 29 zeroes
3-4	AC0
5-6	AC1
7-8	AC2
9-10	AC3
11-12	Previous WFP
13-14	Carry, return PC value
15-18	Stack frame

NOTE: *This instruction saves the information required by the* **WRTN** *instruction.*

This instruction is typically executed after an **XJSR** or **LJSR** instruction. Note that neither of these jump instructions can perform a cross ring call. However, they may be used with **WSSVS** to perform an intra-ring transfer to a subroutine that requires no parameters, and that uses **WRTN** to return control back to the calling sequence.

Wide Special Save/Set Overflow Mask

WSSVS



1	0	0	0	0	1	1	1	0	0	1	1	1	0	0	1	FRAME SIZE IN DOUBLE WORDS
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16 – 31

Pushes a wide return block onto the wide stack and sets **OVK** to 1.

The instruction checks for stack overflow. If executing the instruction would cause an overflow, the instruction transfers control to the wide stack fault handler. The PC in the fault return block will contain the address of the **WSSVS** instruction.

If no overflow would occur, the instruction pushes a wide return block onto the wide stack. After pushing the sixth double word, places the value of WSP in WFP and AC3. Increments WSP by twice the frame size (a 16-bit, unsigned integer contained in the second word of this instruction). Sets **OVK** to 1, which enables integer overflow. Sets **OVR** to 0.

The structure of the wide return block pushed is as follows:

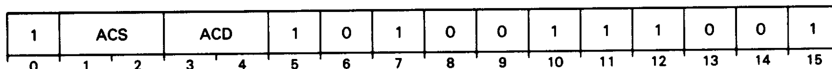
Word in Block	Contents
1-2	OVK, OVR, IRES , 29 zeroes
3-4	AC0
5-6	AC1
7-8	AC2
9-10	AC3
11-12	Previous WFP
13-14	Carry, return PC value
15-18	Stack frame

NOTE: *This instruction saves the information required by the* WRTN *instruction.*

This instruction is typically executed after an **XJSR** or **LJSR** instruction. Note that neither of these jump instructions can perform a cross ring call. However, they may be used with **WSSVR** to perform an intra-ring transfer to a subroutine that requires no parameters, and that uses **WRTN** to return control back to the calling sequence.

Wide Store Byte

WSTB *acs,acd*



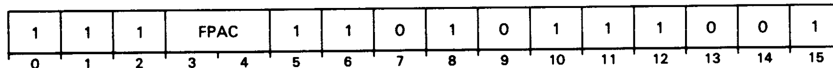
Stores a copy of the rightmost byte of ACD into memory at the address specified by ACS.

ACS contains a 32-bit byte address of some location of memory.

The instruction stores a copy of ACD's bits 24–31 at the locations specified by ACS. The contents of ACS and ACD remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Store Integer

WSTI *fpac*



Converts a floating point number to an integer and stores it into memory.

AC1 contains the data-type indicator that describes the integer.

AC3 contains a 32-bit byte pointer to a byte in memory. The instruction will store the high order byte of the number in this location, with the low order bytes following in subsequent locations.

Under the control of accumulators AC1 and AC3, the instruction translates the contents of the specified FPAC to an integer of the specified type and stores it, right-justified, in memory beginning at the specified location. The instruction leaves the floating point number unchanged in the FPAC, and destroys the previous contents of memory at the

specified location(s).

Upon successful completion, the instruction leaves accumulators AC0 and AC1 unchanged. AC2 contains the original contents of AC3. AC3 contains a byte pointer to the first byte following the destination field. The value of carry is indeterminate and *overflow* is 0.

> **NOTES:** *If the number in the specified FPAC has any fractional part, the result of the instruction is undefined. Use the* Integerize *instruction to clear any fractional part.*
>
> *If the number to be stored is too large to fit in the destination field, this instruction discards high-order digits until the number fits. This instruction stores the remaining low-order digits and sets carry to 1.*
>
> *If the number to be stored does not completely fill the destination field, the data type of the number determines the instruction's actions. If the number is data type 0, 1, 2, 3, 4, or 5, the instruction sets the high-order bytes to 0. If the number is data type 6, the instruction sign extends it to fill the gap. If the number is data type 7, the instruction sets the low-order bytes to 0.*

Wide Store Integer Extended
WSTIX

1	1	0	0	0	1	1	1	0	1	1	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Converts a floating point number to an integer and stores it in memory.

AC1 must contain the data-type indicator describing the integer.

AC3 must contain a 32-bit byte pointer pointing to the high-order byte of the destination field in memory.

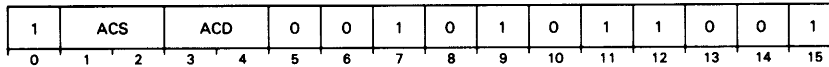
Using the information in AC1, the instruction converts the contents of each of the FPACs to integer form. Forms a 32-bit integer from the low-order 8 digits of each FPAC. Right justifies the integer and stores it in memory beginning at the location specified by AC3. The sign of the integer is the logical OR of the signs of all four FPAC's. The previous contents of the addressed memory locations are lost. Sets carry to 0. The contents of the FPACs remain unchanged. The condition codes in the FPSR are unpredictable.

Upon successful termination, the contents of AC0 and AC1 remain unchanged; AC2 contains the original contents of AC3; and AC3 contains a byte pointer pointing to the first byte following the destination field. The contents of carry are indeterminate and *overflow* is 0.

> **NOTES:** *If the integer is too large to fit in the destination field, the instruction discards high-order digits until the integer fits. The instruction stores remaining low-order digits and sets carry to 1.*
>
> *If the integer does not completely fill the destination field, the data type of the integer determines the instruction's actions. If the data type is 0, 1, 2, 3, 4, or 5, the instruction sets the high-order bytes to 0. Data types 6 and 7 are illegal and will cause a commercial fault.*

Wide Subtract

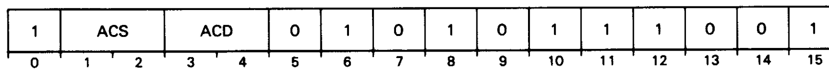
WSUB *acs,acd*



Subtracts the 32-bit integer contained in ACS from the 32-bit integer contained in ACD. Stores the result in ACD. Sets carry to the value of ALU carry. Sets *overflow* to 1 if there is an ALU overflow. The contents of ACS remain unchanged. Carry is unchanged and *overflow* is 0.

Wide Skip on Zero Bit

WSZB *acs,acd*



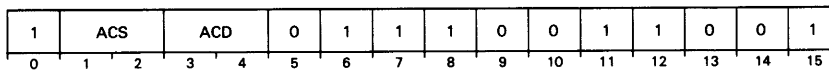
Tests a bit and skips if the bit is zero. Carry is unchanged and *overflow* is 0.

The instruction forms a bit pointer from the contents of ACS and ACD. ACS contains the high-order bits of the bit pointer; ACD contains the low-order bits. ACS and ACD can be specified to be the same accumulator; in this case, the specified accumulator supplies the low-order bits of the bit pointer. The high-order bits are treated as if they were zero in the current ring.

The instruction checks the value of the bit referenced by the bit pointer. If the bit has the value 0, the next sequential word is skipped. If the bit has the value 1, the next sequential word is executed.

Wide Skip on Zero Bit and Set Bit To One

WSZBO *acs,acd*



Tests a bit. Sets the tested bit to 1 and skips if the tested value was zero. Carry is unchanged and *overflow* is 0.

The instruction forms a bit pointer from the contents of ACS and ACD. ACS contains the high-order bits of the bit pointer; ACD contains the low-order bits. ACS and ACD can be specified to be the same accumulator; in this case, the specified accumulator supplies the low-order bits of the bit pointer. The high-order bits are treated as if they were zero.

The instruction checks the value of the bit referenced by the bit pointer. If the bit has the value 0, then the instruction sets the bit to one and skips the next sequential word. If the bit has the value 1, then no skip occurs.

Wide Unsigned Skip If Greater Than Or Equal To

WUSGE *acs,acd*

1	ACS	ACD	0	0	0	1	0	0	1	1	0	0	1
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Compares one integer to another and skips if the first is greater than or equal to the second. Carry is unchanged and *overflow* is 0.

The instruction compares the unsigned, 32-bit integer contained in ACS to the unsigned 32-bit integer in ACD. If the integer contained in ACS is greater than or equal to the integer contained in ACD, the next sequential word is skipped; otherwise, the next sequential word is executed.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to zero. The skip will occur if the integer is greater than or equal to zero.

Wide Unsigned Skip If Greater Than

WUSGT *acs,acd*

1	ACS	ACD	0	0	0	1	0	1	0	1	0	0	1
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Compares one integer to another and skips if the first is greater than the second. Carry is unchanged and *overflow* is 0.

The instruction compares the unsigned, 32-bit integer contained in ACS to the unsigned 32-bit integer in ACD. If the integer contained in ACS is greater than the integer contained in ACD, the next sequential word is skipped; otherwise, the next sequential word is executed.

If ACS and ACD are the same accumulator, then the instruction compares the integer contained in the accumulator to zero. The skip will occur if the integer is greater than zero.

Wide Exchange

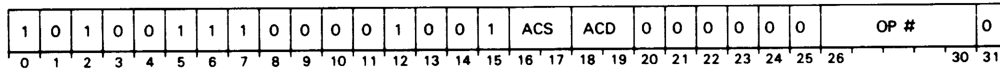
WXCH *acs,acd*

1	ACS	ACD	0	1	1	0	1	1	0	1	0	0	1
0	1 2	3 4	5	6	7	8	9	10	11	12	13	14	15

Exchanges the 32-bit contents of ACS and ACD. Carry is unchanged and *overflow* is 0.

Wide Extended Operation

WXOP *acs,acd,operation #*

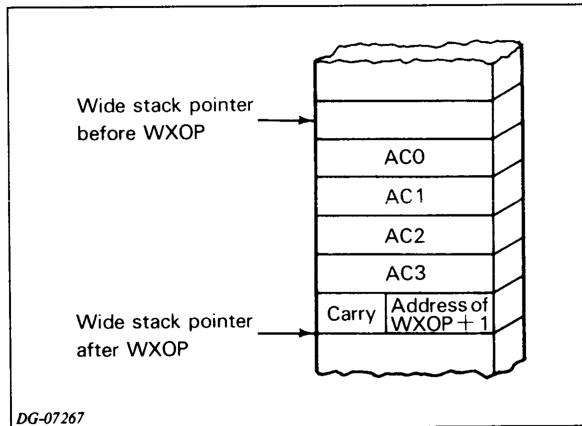


Pushes a return block onto the wide stack and transfers control to an extended operation procedure. Carry is unchanged and *overflow* is 0.

The instruction pushes a return block onto the wide stack. Places the address in the wide stack of ACS into AC2; places the address in the wide stack of ACD into AC3. Memory locations $12\text{–}13_8$ must contain the WXOP origin address, the starting address of a 40_8 word table of addresses. These addresses are the starting location of the various WXOP operations.

The instruction adds the operation number in the WXOP instruction to the WXOP origin address to produce the address of a double word in the WXOP table. Fetches that word and treats it as the intermediate address in the effective address calculation. After the indirection chain, if any, has been followed, the instruction places the effective address in the program counter. The contents of AC0, AC1, and the WXOP origin address remain unchanged. All addresses must be in the current segment.

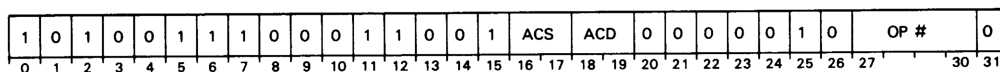
The format of the return block pushed by the instruction is as follows:



This return block is designed so that the WXOP procedure can return control to the calling program via the **WPOP** instruction.

Wide Alternate Extended Operation

WXOP1 *acs,acd,operation #*

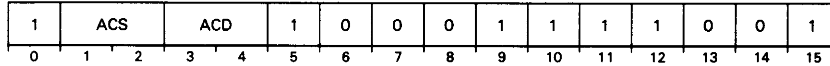


Pushes a return block and transfers control to an extended operation procedure. Carry is unchanged and *overflow* is 0.

The instruction operates in exactly the same way as **WXOP** except that it adds 40_8 to the entry number before it adds the entry number to the **WXOP** origin address. In addition, it can specify only 16 entry locations.

Wide Exclusive OR

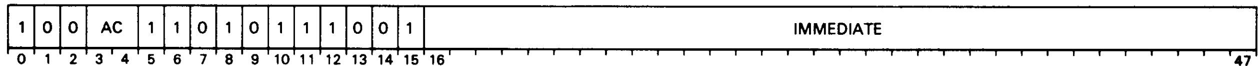
WXOR *acs,acd*



Forms the logical exclusive OR between corresponding bits of ACS and ACD. Loads the 32-bit result into ACD. The contents of ACS remain unchanged, unless ACS equals ACD. Carry is unchanged and *overflow* is 0.

Wide Exclusive OR Immediate

WXORI *ac,immediate*



Forms a logical OR between two values.

The instruction forms the logical exclusive OR between corresponding bits of the specified accumulator and the value contained in the literal field. The instruction places the result of the exclusive OR in the specified accumulator. Carry is unchanged and *overflow* is 0.

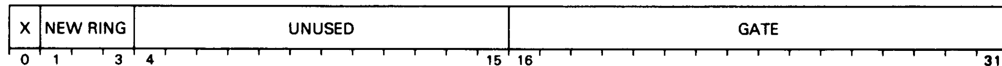
Call Subroutine (Extended Displacement)

XCALL *opcode,argument count,displacement*

Evaluates the address of a subroutine call.

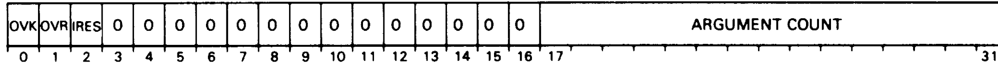
If the target address specifies an outward ring crossing, a protection fault (code=7 in AC1) occurs. Note that the contents of the PC in the return block are undefined.

If the target address specifies an inward ring call, then the instruction assumes the target address has the following format:

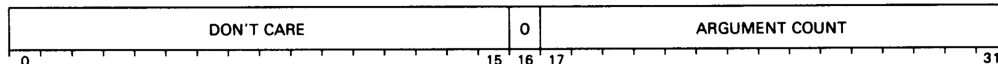


The instruction checks the gate field of the above format for a legal gate. If the specified gate is illegal, a protection fault (code = 6 in AC1) occurs and call is made. Note that the contents of the PC in the return block are undefined.

If the specified gate is legal, or if the target address specifies an intra-ring crossing, then the instruction loads the contents of the PC, + 3, into AC3. The contents of AC3 will always reference the current segment. If bit 0 of the argument count is 0, then the instruction creates a word with the following format:



The instruction pushes this word onto the wide stack. If a stack overflow occurs after this push, a stack fault occurs and no call is made. Note that the value of the PC in the return block is undefined. If bit 0 of the argument count is 1, then the instruction assumes the top word of the wide stack has the following format:

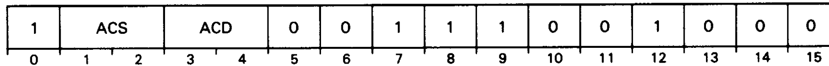


The instruction modifies this word to include the correct settings of **OVK** and **OVR** in bits 0 and 1.

Regardless of the setting of the argument count's bit 0, the instruction next unconditionally sets **OVR** to 0 and loads the PC with the target address. Execution continues with the word referenced by the PC.

Exchange Accumulators

XCH *acs,acd*



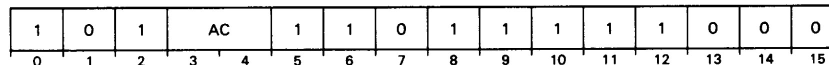
Exchanges the contents of two accumulators.

Places the original contents of bits 16-31 of ACS into bits 16-31 of ACD and the original contents of bits 16-31 of ACD in bits 16-31 of ACS. Carry remains unchanged and *overflow* is 0.

Bit 0-15 of the modified accumulator are undefined after completion of this instruction.

Execute

XCT *ac*



Executes the instruction contained in bits 16-31 of the specified accumulator as if it were in main memory in the location occupied by the *Execute* instruction. If the instruction in bits 16-31 of the specified accumulator is an *Execute* instruction that specifies the same accumulator, the processor is placed in a one-instruction loop.

This instruction leaves carry unchanged; *overflow* is 0.

Because of the possibility of bits 16-31 of the specified accumulator containing an *Execute* instruction, this instruction is interruptible. An I/O interrupt can occur immediately prior to each time the instruction in accumulator is executed. If an I/O interrupt does occur, the program counter in the return block pushed on the system stack points to the *Execute* instruction in main memory. This capability to execute an *Execute* instruction gives you a *wait for I/O interrupt* instruction.

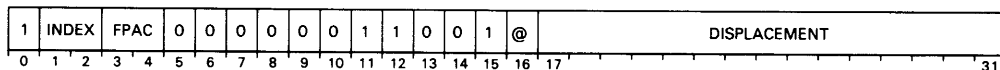
NOTES: *If bits 16-31 of the specified accumulator contains the first word of a two-word instruction, the word following the* XCT *instruction is used as the second word. Normal sequential operation then continues from the second word after the* XCT *instruction.*

Do not use the XCT *instruction to execute an instruction that requires all four accumulators, such as* CMV, CMT, CMP, CTR, *or* BAM.

The results of XCT are undefined if bits 16-31 of the specified accumulator contains an instruction that modifies that same accumulator.

Add Double (Memory to FPAC) (Extended Displacement

XFAMD *fpac,[@]displacement[,index]*



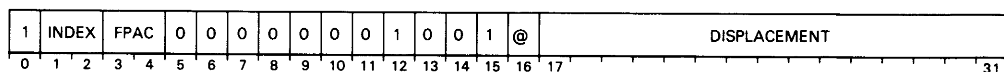
Adds the 64-bit floating point number in the source location to the 64-bit floating point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Adds this 64-bit floating point number to the floating point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Add Single (Memory to FPAC) (Extended Displacement)

XFAMS *fpac,[@]displacement[,index]*



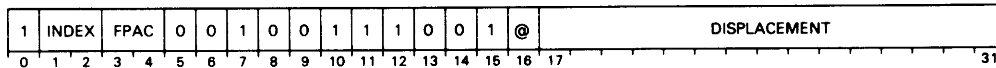
Adds the 32-bit floating point number in the source location to the 32-bit floating point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Adds this 32-bit floating point number to the floating point number in bits 0-31 of the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Divide Double (FPAC by Memory) (Extended Displacement)

XFDMD *fpac,[@]displacement[,index]*



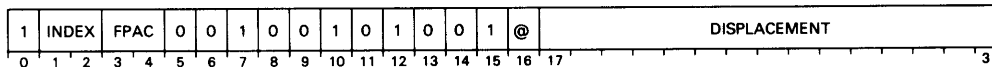
Divides the 64-bit floating point number in FPAC by the 64-bit floating point number in the source location and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Divides the floating point number in the specified FPAC by this 64-bit floating point number. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Divide Single (FPAC by Memory) (Extended Displacement)

XFDMS *fpac,[@]displacement[,index]*



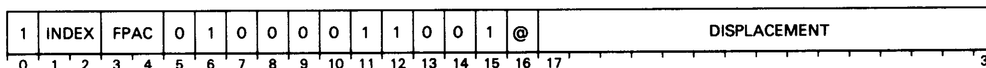
Divides the 32-bit floating point number in bits 0-31 of FPAC by the 32-bit floating point number in the source location and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Divides the floating point number in bits 0-31 of the specified FPAC by this 32-bit floating point number. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Extended Load Floating Point Double

XFLDD *fpac,[@]displacement[,index]*



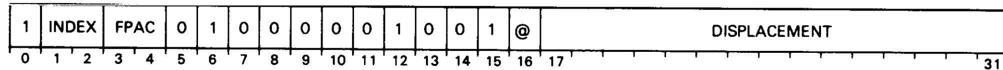
Moves four words out of memory and into a specified FPAC.

Computes the effective address, *E*. Fetches the double precision floating point number at the address specified by *E* and places it in FPAC. Updates the *Z* and *N* flags in the FPSR to reflect the new contents of FPAC.

> **NOTE:** *This instruction will move unnormalized data without change, but the* Z *and* N *flags will be undefined.*

Extended Load Floating Point Single

XFLDS *fpac,[@]displacement[,index]*



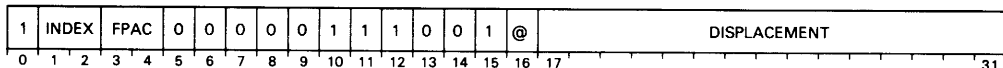
Moves two words out of memory into a specified FPAC.

Computes the effective address, *E*. Fetches the single precision floating point number at the address specified by *E*. Places the number in the high-order bits of FPAC. Sets the low-order 32 bits of FPAC to 0. Updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

> **NOTE:** *This instruction will move unnormalized or illegal data without change, but the Z and N flags will be undefined.*

Multiply Double (FPAC by Memory) (Extended Displacement)

XFMMD *fpac,[@]displacement[,index]*



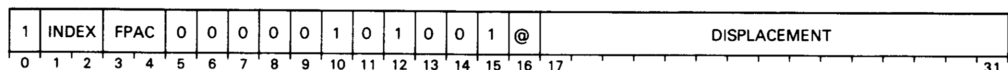
Multiplies the 64-bit floating point number in the source location by the 64-bit floating point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Multiplies this 64-bit floating point number by the floating point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Multiply Single (FPAC by Memory) (Extended Displacement)

XFMMS *fpac,[@]displacement[,index]*



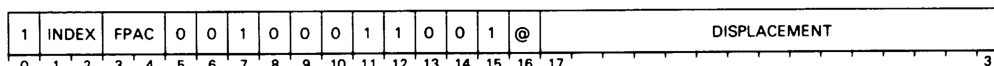
Multiplies the 32-bit floating point number in the source location by the 32-bit floating point number in bits 0-31 of FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Multiplies this 32-bit floating point number by the floating point number in bits 0-31 of the specified FPAC. Places the normalized result in bits 0-31 of the specified FPAC. Sets bits 32-63 of FPAC to 0. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Subtract Double (Memory from FPAC) (Extended Displacement)

XFSMD *fpac,[@]displacement[,index]*



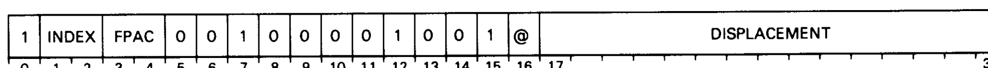
Subtracts the 64-bit floating point number in the source location from the 64-bit floating point number in FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a double precision (four word) operand. Subtracts this 64-bit floating point number from the floating point number in the specified FPAC. Places the normalized result in the specified FPAC. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Subtract Single (Memory from FPAC) (Extended Displacement)

XFSMS *fpac,[@]displacement[,index]*



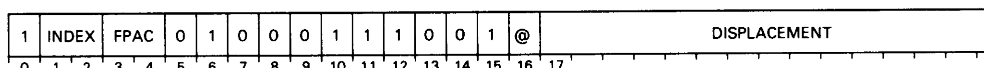
Subtracts the 32-bit floating point number in the source location from the 32-bit floating point number in bits 0-31 of FPAC and places the normalized result in FPAC.

Computes the effective address, *E*. Uses *E* to address a single precision (double word) operand. Subtracts this 32-bit floating point number from the floating point number in bits 0-31 of the specified FPAC. Places the normalized result in the specified FPAC. Sets bits 32-63 of FPAC to 0. Leaves the contents of the source location unchanged and updates the *Z* and *N* flags in the floating point status register to reflect the new contents of FPAC.

See Chapter 8 and Appendix G for more information about floating point manipulation.

Store Floating Point Double (Extended Displacement)

XFSTD *fpac,[@]displacement[,index]*



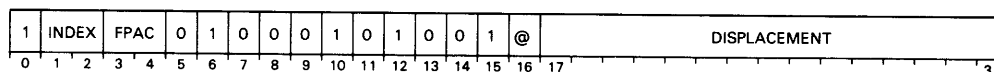
Stores the contents of a specified FPAC into a memory location.

Computes the effective address, *E*. Places the floating point number contained in FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location. The contents of FPAC and the condition codes in the FPSR remain unchanged.

NOTE: *This instruction moves unnormalized or illegal data without change.*

Store Floating Point Single (Extended Displacement)

XFSTS *fpac,[@]displacement[,index]*



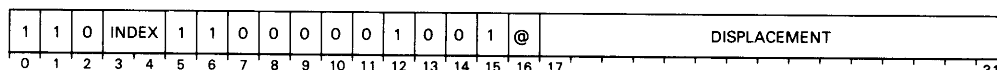
Stores the contents of a specified FPAC into a memory location.

Computes the effective address, *E*. Places the 32 high-order bits of FPAC in memory beginning at the location addressed by *E*. Destroys the previous contents of the addressed memory location. The contents of FPAC and the condition codes in the FPSR remain unchanged.

NOTE: *This instruction moves unnormalized or illegal data without change.*

Jump (Extended Displacement)

XJMP *index,displacement*

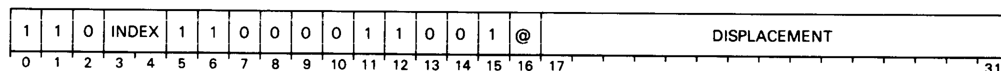


Calculates the effective address, *E*. Loads *E* into the PC. Carry is unchanged and *overflow* is 0.

NOTE: *The calculation of* E *is forced to remain within the current segment of execution.*

Jump to Subroutine (Extended Displacement)

XJSR *index,displacement*

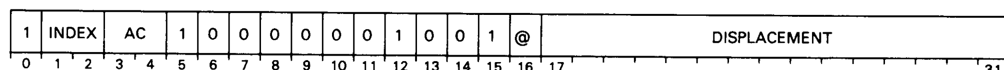


Calculates the effective address, *E*. Loads the current value of the PC, plus two, into AC3. Loads *E* into the PC. Carry is unchanged and *overflow* is 0.

NOTE: *The calculation of* E *is forced to remain within the current segment of execution.*

Load Effective Address (Extended Displacement)

XLEF *ac,index,displacement*

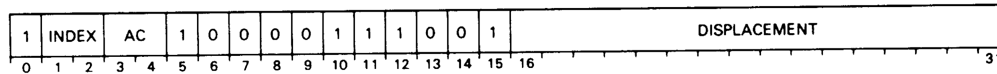


Loads an effective address into an accumulator.

The instruction calculates the effective address, *E*. Checks *E* for ring crossing errors. If no errors occur, loads *E* into the specified accumulator. If errors occur, issues a protection fault. Carry is unchanged and *overflow* is 0.

Load Effective Byte Address (Extended Displacement)

XLEFB *ac,index,displacement*



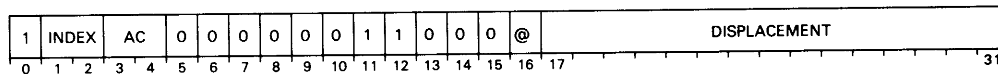
Loads an effective byte address into an accumulator. Carry is unchanged and *overflow* is 0.

The instruction calculates the effective byte address. Checks the byte address for ring crossing errors. If no errors occur, loads the byte address into the specified accumulator. If errors occur, issues a protection fault.

NOTE: *Index bits of 00 force the first address in the effective address calculation to be in the current segment of execution.*

Narrow Add Accumulator to Memory Word (Extended Displacement)

XNADD *ac,index,displacement*

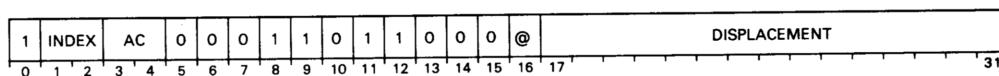


Adds an integer in a memory location to an integer in an accumulator.

The instruction calculates the effective address, *E*. Adds the 16-bit integer contained in the location specified by *E* to the integer contained in bits 16–31 of the specified accumulator. Sign extends the 16-bit result to 32 bits and loads it into the specified accumulator. Sets carry to the value of ALU carry, and *overflow* to 1 if there is an ALU overflow. The contents of the referenced memory location remain unchanged.

Narrow Divide Memory Word (Extended Displacement)

XNDIV *ac,index,displacement*

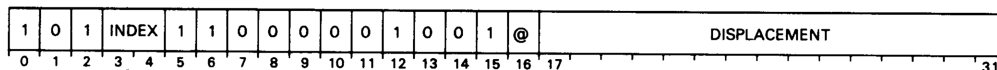


Divides an integer contained in an accumulator by an integer in memory.

The instruction calculates the effective address, *E*. Sign extends the integer contained in bits 16–31 of the specified accumulator to 32 bits and divides it by the 16-bit integer contained in the location specified by *E*. If the quotient is within the range -32,768 to +32,767 inclusive, sign extends the result to 32 bits and loads it into the specified accumulator. If the quotient is outside of this range, or if the divisor is zero, the instruction sets *overflow* to 1 and leaves the specified accumulator unchanged. Otherwise, *overflow* is 0. The contents of the referenced memory location and carry remain unchanged.

Narrow Decrement and Skip if Zero (Extended Displacement)

XNDSZ *index,displacement*

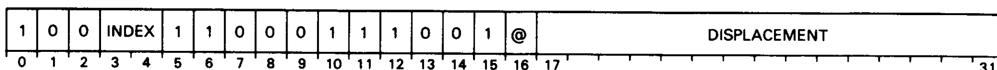


Calculates the effective address, *E*. Decrements the 16-bit contents of the location addressed by *E*. If the decremented result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction is indivisible.*

Narrow Increment and Skip if Zero (Extended Displacement)

XNISZ *index,displacement*

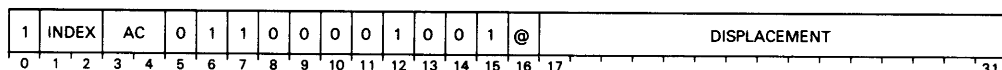


Calculates the effective address, *E*. Increments the 16-bit contents of the location specified by *E*. If the incremented result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

NOTE: *This instruction is indivisible.*

Narrow Load Accumulator (Extended Displacement)

XNLDA *ac,index,displacement*

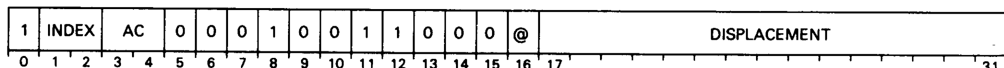


Loads a value into an accumulator.

The instruction calculates the effective address, *E*. Uses *E* as the address of a 16-bit value. Loads this 16-bit value into the specified accumulator, then sign extends the value to 32 bits. Carry is unchanged and *overflow* is 0.

Narrow Multiply Memory Word (Extended Displacement)

XNMUL *ac,index,displacement*



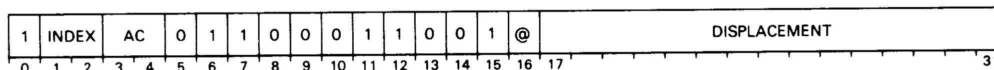
Multiplies an integer in an accumulator by an integer in memory.

The instruction calculates the effective address, *E*. Multiplies the 16-bit, signed integer contained in the location referenced by *E* by the signed integer contained in bits 16–31 of the specified accumulator. If the result is outside the range of -32,768 to +32,767 inclusive, sets *overflow* to 1; otherwise, *overflow* is 0. Sign extends the result to 32 bits and places the result in the specified accumulator. The contents of the referenced

memory location and carry remain unchanged.

Narrow Store Accumulator (Extended Displacement)

XNSTA *ac,index,displacement*

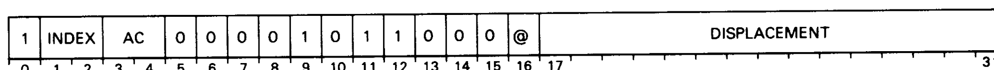


Stores the contents of an accumulator into memory.

The instruction calculates the effective address, *E*. Stores a copy of the 16-bit contents of the specified accumulator in the location specified by *E*. Carry is unchanged and *overflow* is 0.

Narrow Subtract Memory Word (Extended Displacement)

XNSUB *ac,index,displacement*

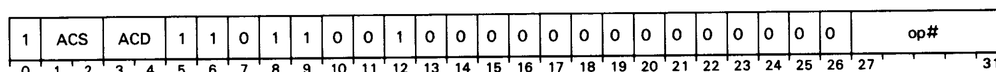


Subtracts an integer in memory from an integer in an accumulator.

The instruction calculates the effective address, *E*. Subtracts the 16-bit integer contained in the location referenced by *E* from the integer contained in bits 16–31 of the specified accumulator. Sign extends the result to 32 bits and stores it in the specified accumulator. Sets carry to the value of ALU carry, and *overflow* to 1 if there is an ALU overflow. The contents of the specified memory location remain unchanged.

Extended Operation

XOP0 *acs,acd,operation #*

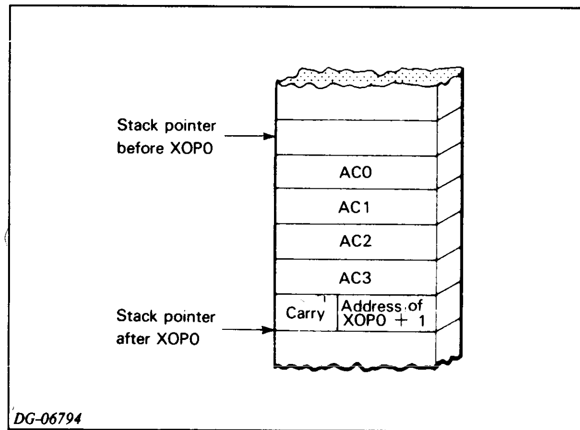


Pushes a return block onto the narrow stack and transfers control to an extended operation procedure.

The instruction pushes a return block onto the narrow stack. Places the address in the narrow stack of ACS into AC2; places the address in the narrow stack of ACD into AC3. Memory location 44_8 must contain the **XOP0** origin address, the starting address of a 40_8 word table of addresses. These addresses are the starting location of the various **XOP0** operations.

The instruction adds the operation number in the **XOP0** instruction to the **XOP0** origin address to produce the address of a double word in the **XOP0** table. Fetches that word and treats it as the intermediate address in the effective address calculation. After the indirection chain, if any, has been followed, the instruction places the effective address in the program counter. The contents of carry, AC0, AC1, and the **XOP0** origin address remain unchanged. *Overflow* is 0.

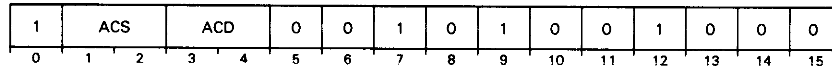
The format of the return block pushed by the instruction is as follows:



This return block is designed so that the **XOP0** procedure can return control to the calling program via the *Pop Block* instruction.

Exclusive OR

XOR *acs,acd*

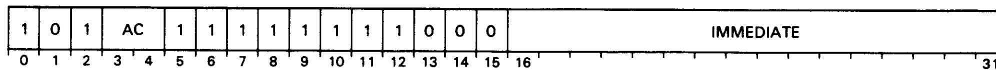


Forms the logical exclusive OR of the contents of bits 16-31 of ACS and the contents of bits 16-31 of ACD and places the result in bits 16-31 of ACD. Sets a bit position in the result to 1 if the corresponding bit positions in the two operands are unlike; otherwise, the instruction sets result bit to 0. The contents of ACS and carry remain unchanged. *Overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Exclusive OR Immediate

XORI *i,ac*

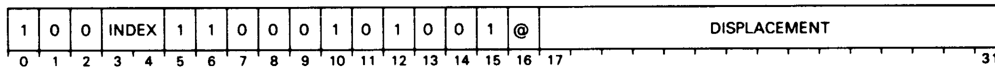


Forms the logical exclusive OR of the contents of the immediate field and the contents of bits 16-31 of the specified accumulator and places the result in bits 16-31 of the specified accumulator. Carry remains unchanged and *overflow* is 0.

Bits 0-15 of the modified accumulator are undefined after completion of this instruction.

Push Address (Extended Displacement)

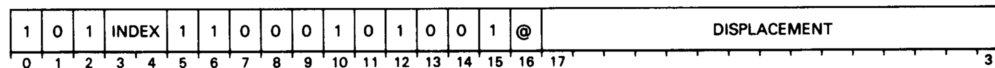
XPEF *index, displacement*



Calculates the effective address, *E*. Pushes *E* onto the wide stack, then checks for stack overflow. Carry is unchanged and *overflow* is 0.

Push Byte Address (Extended Displacement)

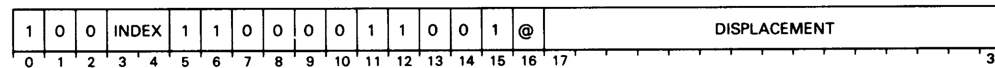
XPEFB *index, displacement*



Calculates a 32-bit byte address. Pushes this byte address onto the wide stack, then checks for stack overflow. Carry is unchanged and *overflow* is 0.

Push Jump (Extended Displacement)

XPSHJ *index,displacement*

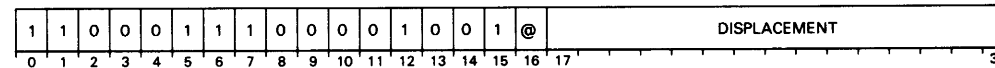


Calculates the effective address, *E*. Pushes the current 31-bit current value of the PC plus two onto the wide stack. Loads the PC with *E*. Checks for stack overflow. Carry is unchanged and *overflow* is 0.

NOTE: *The address pushed onto the wide stack will always reference the current segment.*

Vector on Interrupting Device (Extended Displacement)

XVCT



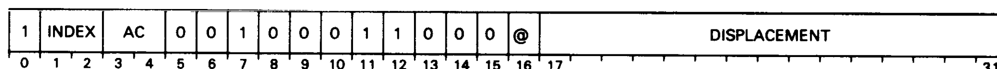
When a device requests an interrupt, transfers control to the appropriate interrupt sequence. Carry is unchanged and *overflow* is 0.

The instruction interprets the displacement field as an absolute address in the current segment. See the chapter on interrupt processing for a complete description of this instruction.

NOTE: *This is a privileged instruction.*

Wide Add Accumulator to Memory Word (Extended Displacement)

XWADD *ac,index,displacement*

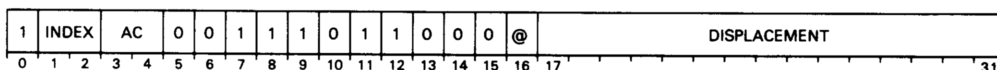


Adds an integer contained in memory to an integer contained in an accumulator.

The instruction calculates the effective address, *E*. Adds the 32-bit integer contained in the location specified by *E* to the 32-bit integer contained in the specified accumulator. Loads the result into the specified accumulator. Sets carry to the value of ALU carry, and *overflow* to 1 if there is an ALU overflow. The contents of the referenced memory location remain unchanged.

Wide Divide Memory Word (Extended Displacement)

XWDIV *ac,index,displacement*



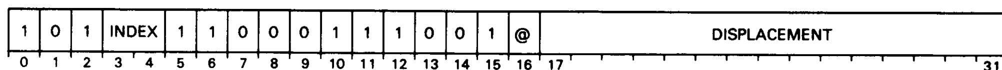
Divides an integer in an accumulator by an integer in memory.

The instruction calculates the effective address, *E*. Sign extends the 32-bit integer contained in the specified accumulator and divides it by the 32-bit integer contained in the location specified by *E*. If the quotient is within the range of -2,147,483,648 to +2,147,483,647 inclusive, the instruction loads it into the specified accumulator. If the quotient is outside this range, the instruction does not load it into the specified accumulator. The contents of the referenced memory loaction and carry remain unchanged.

If the divisor in memory is zero, or if the dividend is the largest negative number and the divisor is -1, the instruction sets *overflow* to 1 and leaves the specified accumulator unchanged. Otherwise, *overflow* is 0.

Wide Decrement and Skip if Zero (Extended Displacement)

XWDSZ *index,displacement*

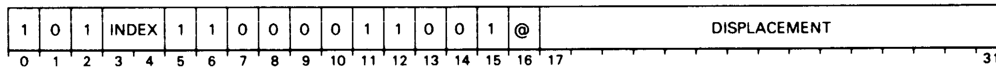


Calculates the effective address, *E*. Decrements the 32-bit contents of the location addressed by *E* by one. If the decremented result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

> **NOTE:** *This instruction executes in one indivisible memory cycle if the word to be decremented is located on a double word boundary.*

Wide Increment and Skip if Zero (Extended Displacement)

XWISZ *index,displacement*

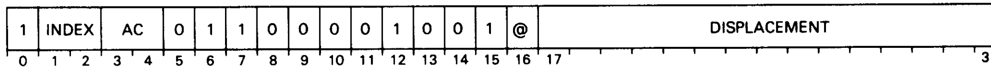


Calculates the effective address, *E*. Increments the 32-bit contents of the location addressed by *E* by one. If the incremented result is equal to zero, then the instruction skips the next sequential word. Carry is unchanged and *overflow* is 0.

> **NOTE:** *This instruction executes in one indivisible memory cycle if the word to be incremented is located on a double word boundary.*

Wide Load Accumulator (Extended Displacement)

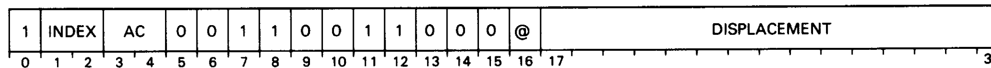
XWLDA *ac,index,displacement*



Calculates the effective address, *E*. Loads a copy of the 32-bit word addressed by *E* into the specified accumulator. Carry is unchanged and *overflow* is 0.

Wide Multiply Memory Word (Extended Displacement)

XWMUL *ac,index,displacement*



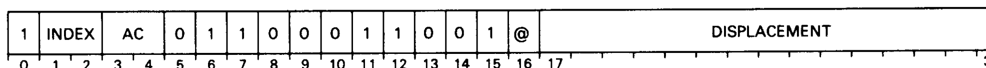
Multiplies an integer in an accumulator by an integer in memory.

The instruction calculates the effective address, *E*. Multiplies the 32-bit, signed integer contained in the location referenced by *E* by the 32-bit, signed integer contained in the specified accumulator. Loads the 32 least significant bits of the result into the specified accumulator.

If the result is within the range of -2,147-483,648 to +2,147,483,647 inclusive, the instruction sets *overflow* to 0; otherwise, *overflow* is 1. The contents of the referenced memory location and carry remain unchanged.

Wide Store Accumulator (Extended Displacement)

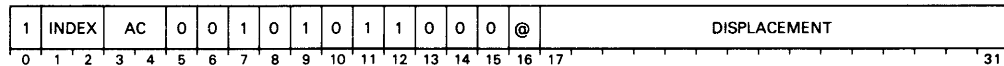
XWSTA *ac,index,displacement*



Calculates the effective address, *E*. Stores a copy of the 32-bit contents of the specified accumulator in the memory location specified by *E*. Carry is unchanged and *overflow* is 0.

Wide Subtract Memory Word (Extended Displacement)

XWSUB *ac,index,displacement*

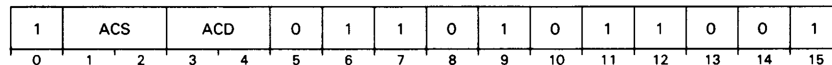


Subtracts an integer contained in memory from an integer contained in an accumulator.

The instruction calculates the effective address, *E*. Subtracts the 32-bit integer contained in the location referenced by *E* from the 32-bit integer contained in the specified accumulator. Loads the result into the specified accumulator. Sets carry to the value of ALU carry, and *overflow* to 1 if thee is an ALU overflow. The contents of the specified memory location remain unchanged.

Zero Extend

ZEX *acs,acd*



Zero extends the 16-bit integer contained in ACS to 32 bits and loads the result into ACD. The contents of ACS remain unchanged, unless ACS equals ACD. Carry is unchanged and *overflow* is 0.

Chapter 17
I/O Instruction Dictionary

This chapter lists the I/O instructions supported by the MV/8000, including those intended for a specific device such as the MAP, the BMC, and the CPU. These instructions appear in alphabetical order according to mnemonic.

In general, these I/O instructions can be executed only with *Lef* mode and I/O protection disabled. See Chapter 4 for a discussion of *Lef* mode and I/O protection.

General I/O Instructions

You can use the following general I/O instructions with any I/O device, using the appropriate device code.

Device Flag Commands

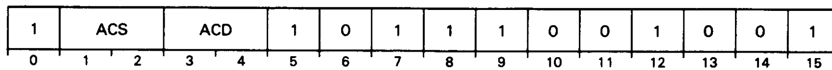
f=S Issues a Start pulse to the specified device.

f=C Issues a Clear pulse to the specified device.

f=P Issues an I/O pulse to the specified device.

IORST No effect.

Command I/O
CIO



Issues a read or write data command using the I/O system bus. Carry is unchanged and *overflow* is 0.

The command must have the form:

Bits 16-31 of ACS contain the command. Bit 16 of ACS indicates whether a read or a

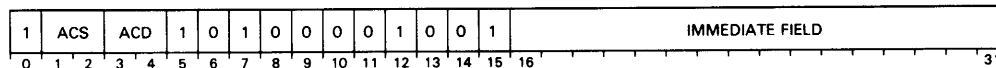
write operation is to take place.

The instruction issues the command contained in ACS directly via the I/O system bus. Bit 16 of ACS determines the operation to perform. If bit 16 of ACS is 0, the instruction performs a read data operation. The instruction receives the data via the I/O system bus and loads it into bits 16-31 of ACD. Bits 0-15 of ACD are undefined.

If bit 16 of ACS is 1, the instruction performs a write data operation and sends the data in bits 16-31 of ACD via the I/O system bus.

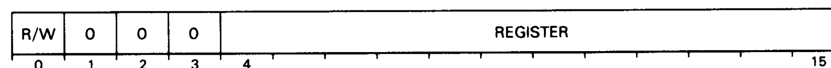
Command I/O Immediate

CIOI



Issues a command via the I/O system bus. Carry is unchanged and *overflow* is 0.

The command must have the form:



If ACS and ACD are the same, then the immediate field contains the command to be issued on the I/O system bus.

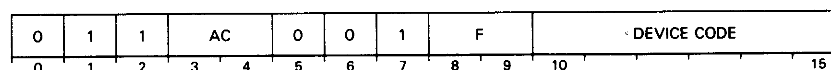
If ACS and ACD are different, then the logical OR of the immediate field and bits 16-31 of ACS is the command to be issued on the I/O system bus.

If bit 0 of the command is a 0, then a read data operation issued via the I/O system bus loads the received data into bits 16-31 of ACD. Bits 0-15 of ACD remain undefined.

If bit 0 of this state is 1, then a write data operation issued via the I/O system bus sends the contents of ACD bits 16-31 to the device.

Data In A

DIA*[f]* *ac,device*



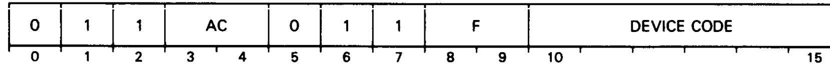
Transfers data from the A buffer of an I/O device to bits 16-31 of an accumulator.

The contents of the A input buffer in the specified device are placed in bits 16-31 of the specified accumulator. After the data transfer, the Busy and Done flags are set according to the function specified by *F*.

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device. Bits in the accumulator that do not receive data are set to 0.

Data in B

DIB*[f]* *ac,device*



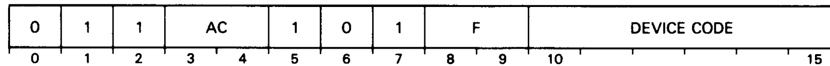
Transfers data from the B buffer of an I/O device to bits 16-31 of an accumulator.

Places the contents of the B input buffer in the specified device in bits 16-31 of the specified accumulator. After the data transfer, sets the Busy and Done flags according to the function specified by *F*.

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device. Bits in the accumulator that do not receive data are set to 0.

Data In C

DIC*[f]* *ac,device*



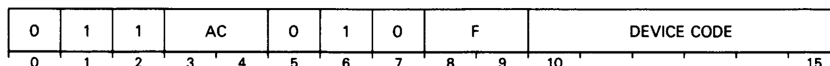
Transfers data from the C buffer of an I/O device to bits 16-31 of an accumulator.

Places the contents of the C input buffer in the specified device in bits 16-31 of the specified accumulator. After the data transfer, sets the Busy and Done flags according to the specified *F*.

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device. Bits in the accumulator that do not receive data are set to 0.

Data Out A

DOA*[f]* *ac,device*



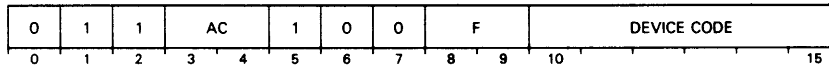
Transfers data from bits 16-31 of an accumulator to the A buffer of an I/O device.

Places the contents of bits 16-31 of the specified accumulator in the A output buffer of the specified device. After the data transfer, sets the Busy and Done flags according to the function specified by *F*. The contents of the specified accumulator remain unchanged.

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device.

Data Out B

DOB[*f*] *ac,device*



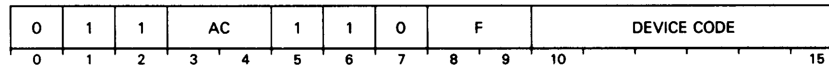
Transfers data from bits 16-31 of an accumulator to the B buffer of an I/O device.

Places the contents of bits 16-31 of the specified accumulator in the B output buffer of the specified device. After the data transfer, sets the Busy and Done flags according to the function specified by *F*. The contents of the specified accumulator remain unchanged.

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device.

Data Out C

DOC[*f*] *ac,device*



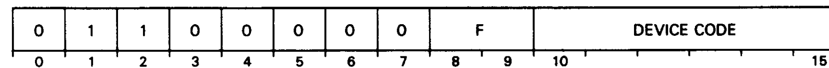
Transfers data from bits 16-31 of an accumulator to the C buffer of an I/O device.

Places the contents of bits 16-31 of the specified accumulator in the C output buffer of the specified device. After the data transfer, sets the Busy and Done flags according to the function specified by *F*. The contents of the specified accumulator remain unchanged.

The number of data bits moved depends upon the size of the buffer and the mode of operation of the device.

No I/O Transfer

NIO [*f*] *ac,device*

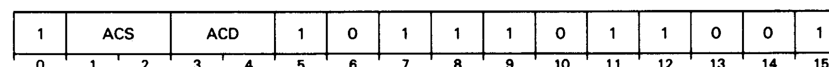


Used when a Busy or Done flag must be changed with no other operation taking place.

Sets the Busy and Done flags in the specified device according to the function specified by *F*.

Program I\O

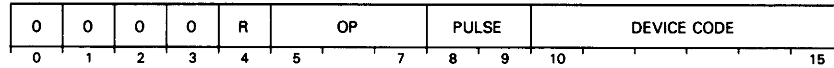
PIO *acs,acd*



Issues a programmed I/O command to an I/O device via the IOSB.

Bits 17-31 of ACS contain the command.

The command to the I/O device must have the form:

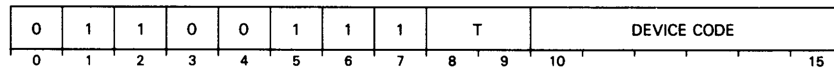


Note that the *Op, Pulse* and *Device Code* fields of this instruction correspond to the *Op, Pulse* and *Device Code* fields of C/350 instructions.

The instruction issues the command contained in ACS to the specified device. It performs the specified operation, using bits 16-31 of ACD as the source or destination of the specified transfer. If ACD is to be the destination of data from the specified device, the transfer stores the data in bits 16-31 of ACD. Bits 0-15 of ACS are undefined. Carry is unchanged and *overflow* is 0.

I/O Skip

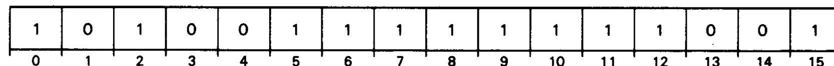
SKP*[t]* *device*



If the test condition specified by T is true, the instruction skips the next sequential word.

Wide Load Map

WLMP



Loads a series of double words into successive map slots.

AC0 contains a double word with the following format:

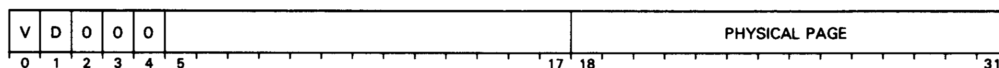
where

map slot references the first map slot to be loaded in the specified I/O channel.

Note that map slots 0-1777_8 refer to BMC slots. Map slots 2000-2177_8 refer to DCH slots.

Bits 16-31 of AC1 specify the number of map slots in the specified I/O channel to be loaded. This value is an unsigned number. The instruction ignores bits 0-15 of AC1.

AC2 points to the first double word that will be loaded into the referenced I/O slots. This double word has the format:



where

V is the valid field. 0 implies valid. 1 implies access denied.

D is the data field. 0 implies transfer data. 1 implies transfer zeroes.

PHYSICAL PAGE specifies the physical page of the data.

This command loads the contents of the double word specified by AC2 into the map slot specified by AC0. It decrements the count in AC1 by 1, increments the map slot number in AC0 by one and the address in AC2 by 2. and continues until AC1 contains zero in bits 16-31. Upon completion, AC0 references the first map slot to be loaded; AC1 contains a 0 in bits 16-31; AC2 contains the address of the word following the last double word loaded; AC3 and carry remain unchanged; *overflow* is 0.

If bits 16-31 of AC1 all initially contain zero, the instruction performs no operation.

NOTE: *This is a privileged instruction.*

Burst Multiplexor Channel

Device Code
5_8 (Primary)

Priority Mask Bit None

Device Flag Commands

f=**S** Sets the Busy flag to 1 and initiates a BMC map load or dump sequence.

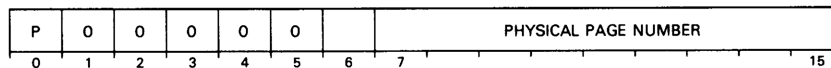
f=**C** Sets the status register (except bit 1) to 0.

f=**P** No effect.

IORST Sets the status register (except bit 1) to 0; turns off mapping.

Map Load Formats

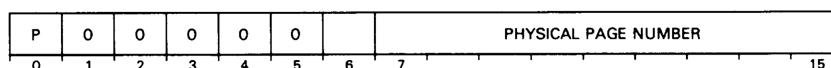
To load the map, the burst multiplexor transfers the contents of a memory buffer to the map register(s). The format of each word in the memory buffer is:



BITS	NAME	CONTENTS or FUNCTION
0	PROT	When 1, the channel cannot transfer data to/from the memory locations in the specified physical page. A transfer attempt results in a validity protect error.
1-5	---	Must be 0.
6-15	PPN	Specify the physical page number for address translation.

Map Dump Formats

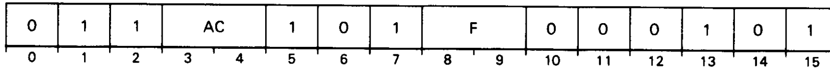
To dump the map, the burst multiplexor transfers the contents of the map register(s) to a memory buffer. The format of each word in the memory buffer is:



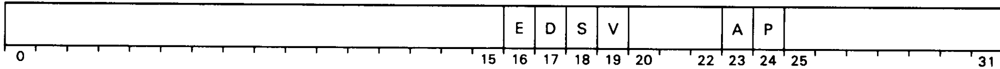
BITS	NAME	CONTENTS or FUNCTION
0	PROT	When 1, the channel cannot transfer data to/from memory in the specified physical page.
1-5	---	Reserved for future use.
6-15	PPN	Physical page number.

Read Status

DIC*[f]* *ac*,BMC



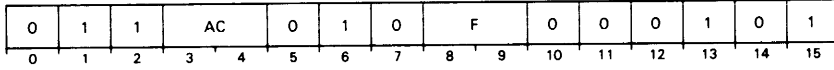
Loads the burst multiplexor status flags into bits 16-31 of the specified accumulator. The previous contents of the accumulator are lost. The format of the accumulator is shown below.



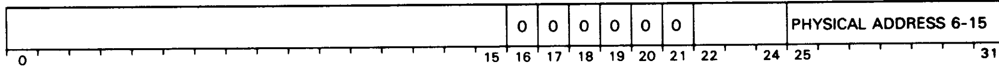
BITS	NAME	CONTENTS or FUNCTION
0-15	---	Reserved for future use.
16	E	When 1, the channel has detected a validity protect error, an address parity error, or a data parity error.
17	D	When 1, the direction for a map data transfer is from the register(s) to memory (dump).
18	S	When 1, the channel is in two step diagnostic mode.
19	V	When 1, the channel has detected a validity protect error.
20-22	---	Reserved for future use.
23	A	When 1, the channel has detected an address parity error.
24	P	When 1, the channel has detected a data parity error.
25-31	---	Reserved for future use.

Specify Low-Order Address

DOA*[f]* *ac*,BMC



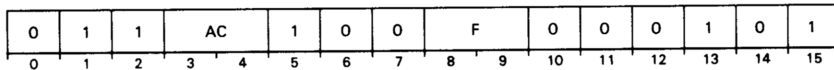
The contents of bits 16-31 of the specified accumulator specify the low-order 10 bits of the 20-bit physical memory address of the first word to be transferred to or from the map. The contents of the accumulator are unchanged. The format of the accumulator is shown below.



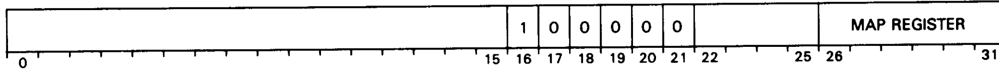
BITS	NAME	CONTENTS or FUNCTION
0-15	---	Reserved for future use.
16-21	---	Must be 0.
22-31	LO ADDR	Specify the least significant bits of the physical address for the start of a map data transfer.

Specify Initial Map Register

DOB*[f]* *ac*,BMC

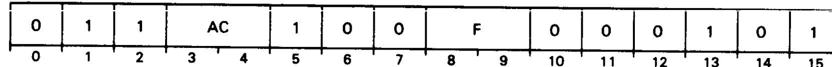


The contents of bits 16-31 of the specified accumulator select the first map register to be loaded or dumped in the next map data transfer. The contents of the accumulator are unchanged. The format of the accumulator is shown below.



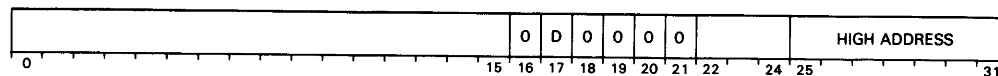
BITS	NAME	CONTENTS or FUNCTION
0-15 16	--- ---	Reserved for future use. Must be 1.
17-21	---	Must be 0.
22-31	MAP REGISTER	Specify a map register as the first location for a map load/dump.

Specify High-Order Address
DOB*[f]* *ac*,**BMC**



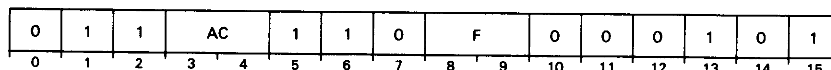
The contents of bits 17-31 of the specified accumulator determine the direction of the next map data transfer, as well as the high-order part of the physical memory address to be used. Bit 17 specifies whether map registers are to be loaded or dumped. Bits 22-31 are the the high-order 10 bits of the 20-bit physical address of the first word in memory to be transferred to or from the map.

The contents of the specified accumulator are unchanged. The format of the accumulator is shown below.

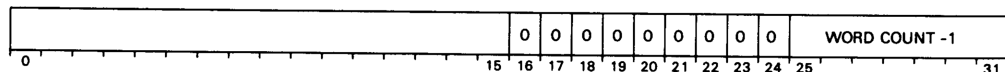


BITS	NAME	CONTENTS or FUNCTION
0-15 16	--- ---	Reserved for future use. Must be 0.
17	DUMP	When 1, the direction for the map data transfer is from the register(s) to memory.
18-21	—	Must be 0.
22-31	HIGH ADDRESS	Specify the most significant bits of the physical address for the start of the map data transfer.

Specify Word Count
DOC*[f]* *ac*,**BMC**

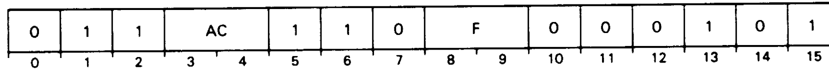


The contents of bits 16-31 of the specified accumulator determine the number of map registers to be loaded or dumped in the next map data transfer. The specified number must be one less than the number of words to be transferred. The contents of the specified accumulator are unchanged. The format of the accumulator is shown below.

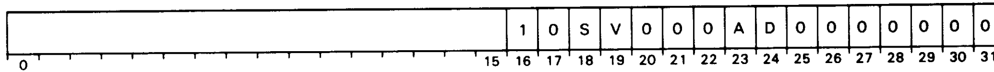


BITS	NAME	CONTENTS or FUNCTION
0-15		
16-24	—	Must be 0.
25-31	COUNT	Specify a number that is one less than the number of map registers to be loaded/dumped.

Set Status
DOC*[f]* ***ac*,BMC**



The contents of bits 16-31 of the specified accumulator control the diagnostic functions of the burst multiplexor. The contents of the accumulator are unchanged. The format of the accumulator is shown below.



BITS	NAME	CONTENTS or FUNCTION
0-15	---	Reserved for future use.
16	---	Must be 1.
17	---	Must be 0.
18	S	When 1, the channel enters two-step diagnostic mode.
19	V	When 1, the channel forces a validity protect error.
21-23	--	Must be 0.
24	A	When 1, the channel forces an address parity error.
25	D	When 1, the channel forces a data parity error.
26-31	---	Must be 0.

Central Processor

Device Code
77_8 (Primary)

Priority Mask Bit None

Device Flag Commands

Device flag commands to the CPU determine whether the current program can be interrupted by a program interrupt request. When the interrupt enable flag is set to 1, the program can be interrupted (once the instruction following the enable has begun). When the interrupt enable flag is set to 0, the program cannot be interrupted. The CPU interrupt enable flag is controlled by the device flag commands as follows:

f=S   Sets the interrupt enable flag to 1.

f=C   Sets the interrupt enable flag to 0.

f=P

IORST

Read Switches

DIA*[f]* ***ac*,CPU**

0	1	1	AC	0	0	1	F	1	1	1	1	1	1
0	1	2	3 4	5	6	7	8 9	10	11	12	13	14	15

Places the contents of the console switches into bits 16-31 of an accumulator.

Places the setting of the console data switches in the specified accumulator. After the transfer, sets the Interrupt On flag according to the function specified by *F*.

> **NOTE:** *The assembler recognizes the special mnemonic* **READS** *ac to be equivalent to* **DIA** ***ac*,CPU**.

Interrupt Acknowledge

DIB*[f]* ***ac*,CPU**

0	1	1	AC	0	1	1	F	1	1	1	1	1	1
0	1	2	3 4	5	6	7	8 9	10	11	12	13	14	15

Returns device code of an interrupting device.

Places the six-bit device code of that device requesting an interrupt which is physically closest to the CPU on the I/O bus in bits 26-31 of the specified accumulator; sets bits 0-25 to 0. After the transfer, sets the Interrupt On flag according to the function specified by *F*.

> **NOTE:** *The assembler recognizes the special mnemonic* **INTA** *ac to be equivalent to* **DIB** ***ac*,CPU**.

Reset

DIC*[f]* ***ac*,CPU**

0	1	1	AC	1	0	1	F	1	1	1	1	1	1
0	1	2	3 4	5	6	7	8 9	10	11	12	13	14	15

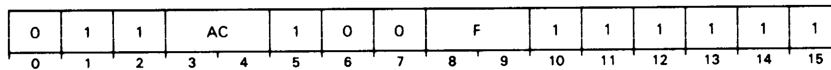
Sends a reset signal to all devices to clear their state.

Sets the 16-bit priority mask to 0. Sets the Interrupt On flag according to the function specified by *F*.

Note that you must code an accumulator to avoid assembly errors. During execution, the accumulator field is ignored and the contents of the accumulator remain unchanged.

> **NOTE:** *The assembler recognizes the special mnemonic* **IORST** *to be equivalent to* **DICC 0,CPU**. *This instruction sets the Busy and Done flags as described above, and sets the Interrupt On flag to 0.*

Mask Out
DOB*[f]* *ac*,**CPU**



Sets the priority mask.

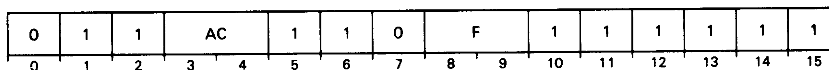
Places the contents of bits 16-31 of the specified accumulator in the priority mask. After the transfer, sets the Interrupt On flag according to the function specified by *F*. The contents of the specified AC remain unchanged.

NOTE: *A 1 in any bit disables interrupt requests at devices which use that bit as a mask.*

Do not use this instruction when interrupts are enabled.

The assembler recognizes the special mnemonic **MSKO** *ac to be equivalent to* **DOB** *ac*,**CPU**.

Halt
DOC*[f]* *ac*,**CPU**

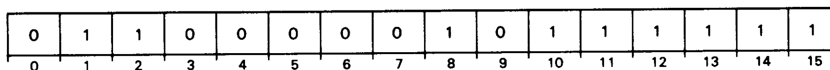


Stops the processor.

Sets the Interrupt On flag according to the function specified by *F*, then stops the processor.

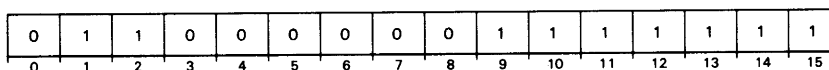
NOTE: *The assembler recognizes the special mnemonic* **HALT** *as equivalent to the instruction* **DOC 0,CPU.**

Interrupt Disable
INTDS
NIOC CPU



Sets Interrupt On flag to 0.

Interrupt Enable
INTEN
NIOS CPU

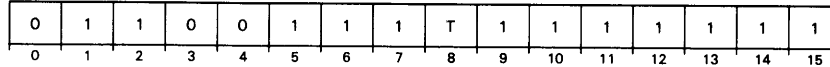


Sets Interrupt On flag to 1.

If the instruction changes the state of the Interrupt On flag, the CPU allows one more instruction to execute before the first I/O interrupt can occur. However, if the instruction is interruptible, then interrupts can occur as soon as the instruction begins to execute.

CPU Skip

SKP*[t]* CPU



If the test condition specified by T is true, the next sequential word is skipped.

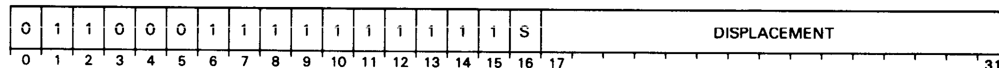
The following table lists the possible test conditions.

SYMBOL	VALUE	TEST
[T]=**BN**	00	Tests Interrupt On flag for 1
[T]=**BZ**	01	Tests Interrupt On flag for 0
[T]=**DN**	10	Tests Power Fail flag for 1
[T]=**DZ**	11	Tests Power Fail flag for 0

See *Programmer's Reference:Peripherals* (DGC No. 014-000632) for a complete set of examples on using the interrupt system.

Vector On Interrupting Device Code

VCT *[@]displacement[,index]*



Returns the device code of the interrupting device and uses that code as an index into a table. The value found in the table is used in one of two ways: it can be a pointer to the appropriate interrupt handler (Mode A), or as a pointer to another table (Modes B through E). This second table points to the interrupt handler and contains a new priority mask. Depending on the mode used, the instruction can also save the state of the machine by pushing various words onto the stack, create a new vector stack, and set up a priority structure.

The accompanying flow chart is a complete diagram of the operation of the Vector instruction. Note that all modes use the *vector table* to find the next address used. Mode A uses the vector table entry as the address of the interrupt handler and passes control to it immediately. Modes B through E all use the vector table address as a pointer into a *device control table* (DCT), where the address of the interrupt handler is found, along with a new priority mask.

Three control bits determine the mode of the *Vector* instruction which will be used. Their names and locations are:

Stack Change Bit
Bit 0 of the second word of the *Vector* instruction;

Direct Bit
Bit 0 of the selected vector table entry;

Push Bit
Bit 0 of the first word of the selected device control table.

The value of these bits collectively determine the mode of the *Vector* instruction. The bits determine the mode as follows:

Direct	Stack	Push	Mode
0	Don't care	Don't care	A
1	0	0	B
1	0	1	C
1	1	0	D
1	1	1	E

The diagram shows the arrangement of the various control bits and tables. The modes perform various functions as summarized below:

Mode	Function
A	Uses device code returned by INTA as table entry to find address of interrupt handler.
B	Mode A plus: resets priority mask (saving old one) and reenables interrupts.
C	Mode B plus: pushes a normal 5-word return block (4 ACs, the program counter, and carry) onto the stack.
D	Mode B plus: sets up a new vector stack for use by the interrupt handler and saves the old stack parameters.
E	Mode C plus Mode D.

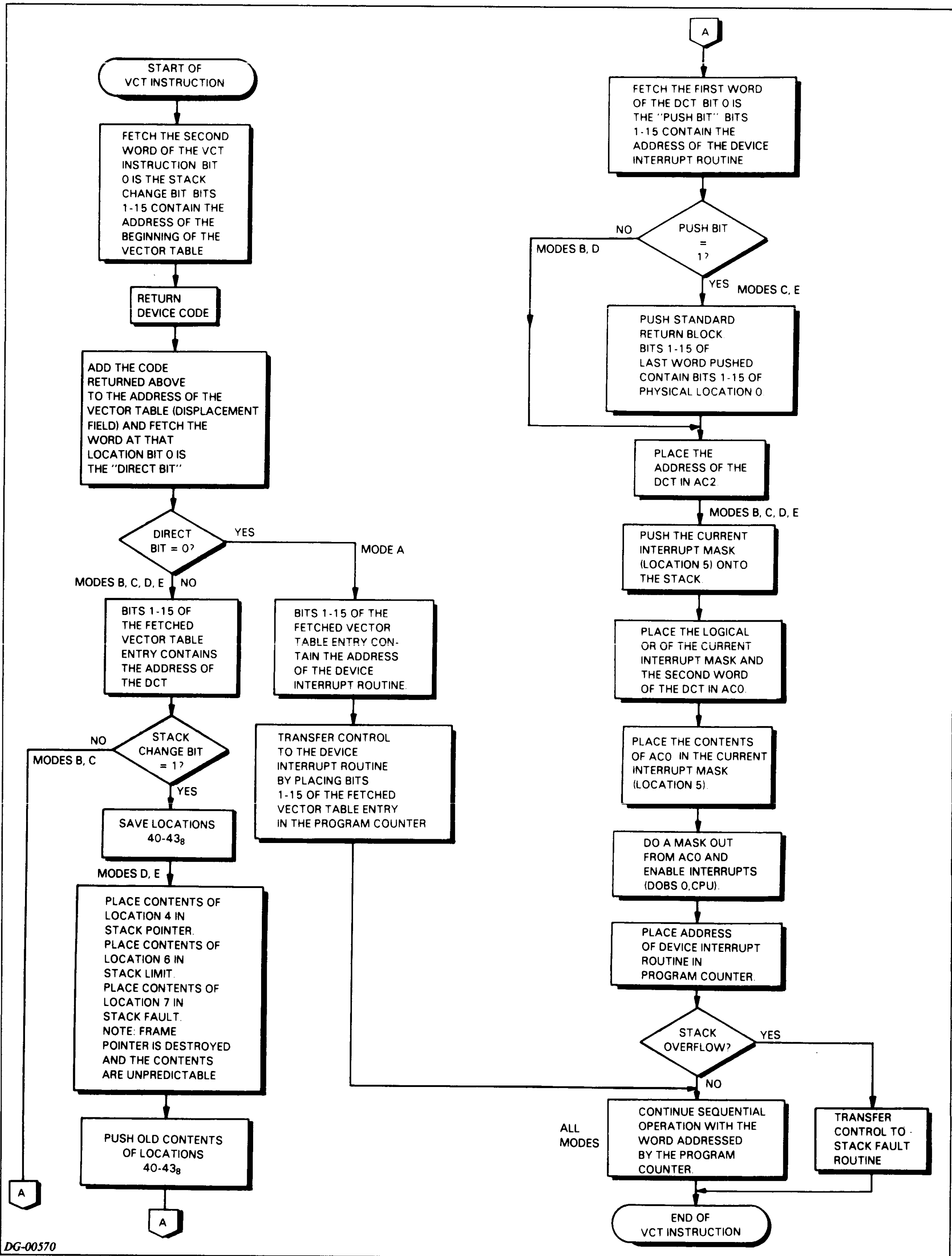


Figure 17.1

In the following paragraphs, we will consider each mode and follow through the process step-by-step.

Common Process

All modes perform the initial steps of the *Vector* instruction. These steps begin when the instruction returns the interrupting device code. The instruction adds the device code to the address of the start of the vector table (bits 1-15 of the second instruction word). The result is the address of an entry within the vector table. The instruction fetches the contents of this vector table entry and examines bit 0 of the entry (the direct bit).

Mode A

The instruction performs the functions of Mode A if the direct bit is 0. The values of the other control bits do not matter. In Mode A, the instruction uses bits 1-15 of the fetched vector table entry as the address of the interrupt handler of the interupting device. Control transfers immediately to the interrupt handler with all interrupts disabled.

Modes B Through E

The direct bit has the value 1 for all of these modes. The values of the push bit and the stack change bit determine which of the four modes will take place. The action of these modes can be divided into two parts: a first part, whose action varies from mode to mode; and a second part, whose action is identical for every mode. We discuss each part I separately, then the common second part.

Mode B Part I
When the stack change bit and the push bit both have the value of 0, then Mode B takes place. The instruction uses the vector table entry as the address of the device control table *(DCT)* for the interrupting device. Bits 1-15 of the first word of the DCT contain the address of The desired interrupt handler (bit 0 is the push bit). The second word of the DCT contains information used to construct the new interrupt priority mask. Succeeding words (if any) contain information to be used by the device interrupt handler.

Mode C Part I
When the stack change bit has the value 0 and the push bit has the value 1, then Mode C takes place. This mode performs the functions of Mode B; in addition, Mode C pushes a standard five-word return block onto the standard stack. The return block contains the contents of the four accumulators, the value of carry, and the contents of physical location 0 (the program counter return value).

Mode D Part I
When the stack change bit has the value 1 and the push bit has the value 0, then Mode D takes place. This mode performs the functions of Mode B; in addition, Mode D sets up a new stack for the interrupt handler (using the contents of locations 4, 6, and 7) and pushes the old contents of physical locations $40\text{-}43_8$ (the user stack control words) onto the new stack.

Mode E Part I
When the stack change bit and the push bit both have the value 1, then Mode E takes place. This mode combines the functions of modes C and D. That is, Mode E performs the functions of mode B, sets up a new stack, and pushes a 5-word return block and the old stack control words onto the new stack.

Modes B through E Part II

Modes B through E use the same procedure for the remainder of the *Vector* instruction. During this procedure, the instruction pushes the current priority mask (location 5) onto the stack. Next, the instruction updates location 5 and performs a *Mask Out* instruction, using the logical OR of the current mask and the second word of the DCT. The instruction then sets the Interrupt On flag to 1 and passes control to the selected device interrupt handler. Note that the CPU permits one more instruction to execute (in this case, the first instruction of the interrupt handler) before the next I/O interrupt can occur.

Host/IOP Communication

Device Code, Host to IOP

60_8 (Primary)

Priority Mask Bit, Host to IOP

5 (Primary)

Device Flag Commands

Device flag commands allow the host to interrupt the IOP. These commands do this by setting interrupt request flags, busy flags, and done flags to initiate the interrupt. The IOP and the host each have one busy flag, one done flag, and one interrupt request flag. The host's flags are located in and are tested by the IOP. The IOP's flags are located in and are tested by the host. The device flag commands for the four Host instructions are:

f=S Sets the Host Busy and the IOP interrupt request flags to 1.

f=C Sets the IOP Done and Host interrupt request flags to 0.

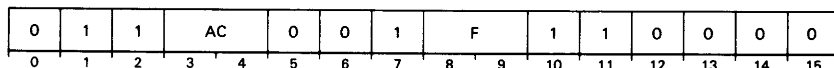
f=P Sets the IOP parity error and Host interrupt request flags to 0.

IORST Sets the IOP Done flag, Host interrupt request flag, and Host interrupt mask bit to 0, also resets the IOP processor and its I/O devices.

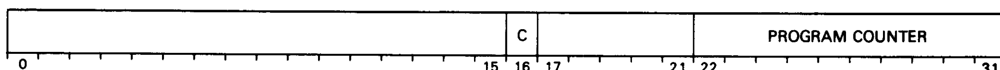
NOTE: *An IOP Run flag is read as Busy by the Host. It is set to 1 when the IOP is not in a halted state.*

Read PC Save Register

DIA*[f]* ***ac*,IOP**



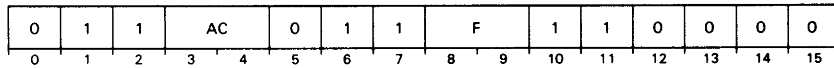
Loads the contents of the IOP's PC Save register into bits 16-31 of the specified accumulator. The IOP updates the PC Save register each time the IOP halts. The format of the accumulator is as follows.



BITS	NAME	CONTENTS or FUNCTION
0-15	---	Reserved for future use.
16	C	SP Carry
17-31	PC	SP Program counter

Read Console Buffer

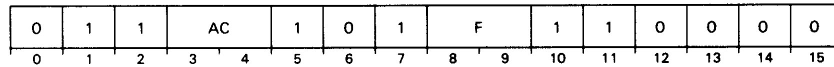
DIB*[f]* *ac*,**IOP**



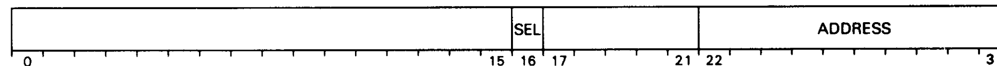
Loads the contents of the IOP console buffer into bits 16-31 of the specified accumulator. The console buffer contains the result of the last console operation performed by the IOP (such as *Examine*).

Read Address Buffer

DIC*[f]* *ac*,**IOP**



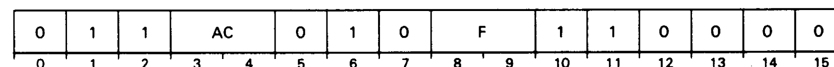
Loads the contents of the IOP address buffer into bits 16-31 of the specified accumulator. The IOP address buffer contains the address of the last Host memory reference. If the address save bit in the IOP is 1, it will contain the address of the last memory reference to either local or host memory. The format of the register is shown below.



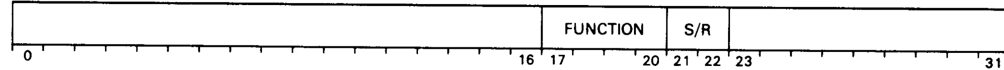
BITS	NAME	CONTENTS or FUNCTION
0-15 16	--- SEL	Reserved for future use. Current value of the least significant host data channel map select bit
17-31	ADDRESS	SP Logical Address

Control Console Function Register

DOA*[f]* *ac*,**IOP**



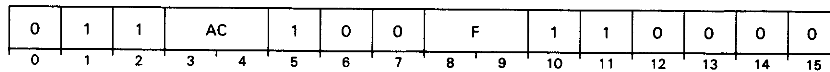
Stores the contents of bits 16-31 of the specified accumulator into the IOP console function register. The format of the specified accumulator is:



BITS	NAME	CONTENTS or FUNCTION
0-16	---	Reserved for future use
17-20		Selects action when bits 5 & 6 = 0: 0000 Examine AC0 0001 Examine AC1 0010 Examine AC2 0011 Examine AC3 0100 Deposit AC0 0101 Deposit AC1 0110 Deposit AC2 0111 Deposit AC3 1000 Deposit 1001 Deposit Next 1010 Examine 1011 Examine Next 1100 Start 1101 Execute 1110 Program Load 1111 Continue
21-22	S/R	11 No action 10 STOP 01 RESET 00 bits 1-4 specify action
23-31	---	Reserved for future use.

NOTE: *Select the* Inst *function by specifying* **STOP** *and* **CONTINUE** *together (i.e., bits 1-6 =* 111110_2*). The IOP ignores functions other than* **RESET, STOP, EXAMINE,** *and* **INST** *when it is running. The* STOP *function halts the IOP after it completes the instruction currently executing.*

Control Switch Register
DOB*[f]* *ac*,**IOP**



Stores the contents of bits 16-31 of the specified accumulator into the IOP switch register. The IOP switch register contains the address or data for the IOP console operations.

Memory Allocation and Protection

Device Code
3_8 (Primary)

Priority Mask Bit
None

Device Flag Commands

f=S	No effect.
f=C	No effect.
f=P	Enables Map Single Cycle.
IORST	Disables Map.

Load Map
LMP

1	0	0	1	0	1	1	1	0	0	0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Under control of AC1 and AC2, loads successive words from memory into the MAP where they are used to define a user or data channel map.

Bits 16-31 of AC1 must contain an unsigned integer, which is the number of words to be loaded into the MAP. Bits 17-31 of AC2 must contain the address of the first word to be loaded. If bit 0 of AC2 is 1, the instruction follows the indirection chain and places the resultant effective address in AC2. AC0 and AC3 are ignored and their contents remain unchanged.

For each word loaded, the instruction decrements the count in AC1 by one and increments the source address in AC2 by 1. Upon completion of the instruction, AC1 contains 0, and AC2 contains the address of the word following the last word loaded.

This instruction is interruptible in the same manner as the *Block add and move* instruction. If you issue this instruction while in mapped mode, with I/O protection enabled, the map will not be altered. AC1 and AC2 will be used and their contents modified as described above. No I/O trap will occur.

The words loaded into the MAP define the address translation functions for the various user and data channel maps. The contents of the **MAP** field (bits 6-8) of the MAP status register determine which map is affected by the *Load map* instruction. You can alter this field using either the *Load map status* or the *Initiate page check* instruction.

The format of the words loaded into the MAP is as follows:

WP	LOGICAL	PHYSICAL
0	1–5	6–15

BITS	NAME	CONTENTS or FUNCTION
0	WP	Unused for data channel maps; write protect for user maps.
1-5	LOGICAL	Logical page number.
6-15	PHYSICAL	Physical page number.

NOTE: *Declare a logical page invalid by setting the Write Protect bit to 1 and all of bits 6-15 to 1.*

Read Map Status
DIA*[f]* *ac*,MAP

0	1	1	AC		0	0	1	0	0	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Reads the status of the current map.

Places the contents of the MAP status register in bits 16-31 of the specified accumulator. The previous contents of the specified accumulator are lost. The format of the information placed in the specified accumulator is as follows:

	I/O	WP	ind	SC	MAP			lef	I/O	WP	ind	a/b	dch	u/m
0 … 17	18	19	20	21	22		24	25	26	27	28	29	30	31

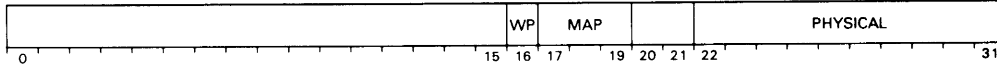
BITS	NAME	CONTENTS or FUNCTION
0-17	---	Reserved for future use.
18	I/O	If 1, the last protection fault was an I/O protection fault.
19	WP	If 1, the last protection fault was a write protection fault.
20	IND	If 1, the last protection fault was an indirect protection fault.
21	Single Cycle	If 1, the last map reference was a *Map Single Cycle* instruction.
22-24	Map	Indicates which map will be loaded by next *Load map* instruction as follows:
		000 User A
		001 Reserved for future use
		010 User B
		011 Reserved for future use
		100 Data channel A
		101 Data channel C
		110 Data channel B
		111 Data channel D
25	LEF	If 1, the *Load Effective Address* instruction was enabled by the last *Load Map Status* instruction.
26	I/O	If 1, I/O protection was enabled by the last *Load Map Status* instruction.
27	WP	If 1, write protection was enabled by the last *Load Map Status* instruction
28	IND	If 1, indirect protection was enabled by the last *Load Map Status* instruction.
29	A/B	If 0, the last *Load Map Status* instruction enabled map A.
		If 1, the last *Load Map Status* instruction enabled user map B.
30	DCH Enable	If 1, the mapping of the data channel addresses is enabled.
31	User Mode	If 1, the last I/O interrupt occurred while in user mode.

Page Check
DIC *ac*,MAP

0	1	1	AC	1	0	1	0	0	0	0	0	0	1	1	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

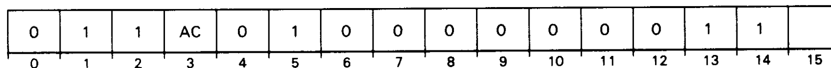
Provides the identity and some characteristics of the physical page corresponding to the logical page identified by the immediately preceding *Initiate Page Check* instruction.

Places the number of the physical page which corresponds to the logical page specified by the preceding *Initiate Page Check* or *Load Map Status* instruction in bits 22-31 of the specified accumulator. Places additional information about this page in bits 16-19 and destroys the previous contents of the accumulator. The format of the information placed in the specified accumulator is as follows:



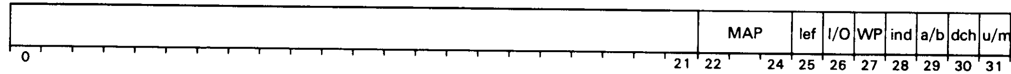
BITS	NAME	CONTENTS or FUNCTION
0-15 16	--- WP	Reserved for future use. The write protect bit for the logical page which corresponds to the physical page specified by bits 6-15.
17-19	Map	The map which was used to perform the translation between logical page number and physical page number is as follows: 000 User A 001 Reserved for future use. 010 User B 011 Reserved for future use. 100 Data channel A 101 Data channel C 110 Data channel B 111 Data channel D
20-21	---	Reserved for future use.
22-24	Validity	If these bits are 1, and bit 0 is 1, the logical page which corresponds to the physical page specified by bits 9-15 is validity protected.
25-31	Physical	The number of the physical page which

Load Map Status
DOA *ac*,**MAP**



Defines the parameters of a new map.

Places the contents of the specified accumulator in the MAP status register. The contents of the specified accumulator remain unchanged. The format of the specified accumulator is as follows:

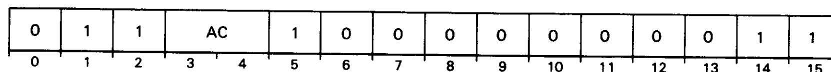


BITS	NAME	CONTENTS or FUNCTION
0-21	---	Reserved for future use.
22-24	MAP SEL	Specify which map will be loaded by the next Load Map instruction as follows: 000 User A 001 Reserved for future use 010 User B 011 Reserved for future use 100 Data channel A 101 Data channel C 110 Data channel B 111 Data channel D
25	LEF	If 1, the Load Effective Address instruction will be enabled for the next user
26	I/O	If 1, I/O protection will be enabled for the next user
27	WP	If 1, write protection will be enabled for the next user
28	IND	If 1, indirect protection will be enabled for the next user
29	A/B	If 0, the next user map enabled will be that for user A If 1, the next user map enabled will be that for user B
30	DCH Enable	If 1, the mapping of data channel addresses will be enabled immediately after this instruction
31	User Mode	If 1, mapping of CPU addresses will commence with the first memory reference *after* the next *indirect* reference or return type instruction (**POPB, POPJ, RTN, RSTR**)

If the *Load Map Status* instruction sets the User Enable bit to 1, this inhibits the interrupt system and the MAP waits for either an indirect reference or return type instruction. Either event releases the interrupt system and allows the MAP to begin translating addresses (using the user map specified by bit 13 of the MAP status register). Address translation resumes (1) after the first level of the next indirect reference; or (2) after the first *Pop Block, Pop Jump, Return*, or,*Restore* instruction that does not cause a stack fault.

Map Page 31

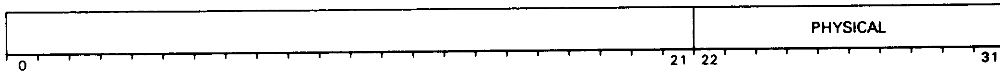
DOB *ac*,MAP



Specifies that mapping take place for a single page of an unmapped address space. Mapping is always done for locations 76000_8 through 77777_8 (logical page 31). This is the only page which can be mapped when in unmapped address space. You can use this instruction to access a page of a user's memory space when in unmapped mode.

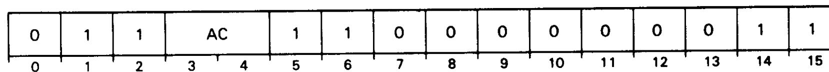
Bits 22-31 of the specified accumulator are transferred to the MAP. These bits specify a physical page number to which logical page 31 will be mapped when in the unmapped mode.

The contents of the specified accumulator remain unchanged. The format of the specified accumulator is as follows:



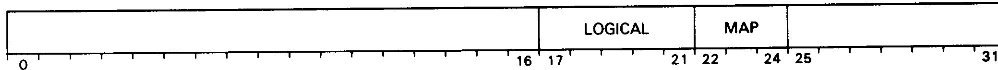
BITS	NAME	CONTENTS or FUNCTION
0-21	---	Reserved for future use.
22-31	Physical	The number of the physical page to which logical page 31 should be mapped when in unmapped mode.

Initiate Page Check
DOC *ac*,MAP



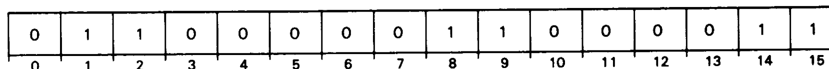
Identifies a logical page. The *Page Check* instruction will find the corresponding physical page.

Transfers the contents of bits 16-31 of the specified accumulator to the MAP for later use by the *Page Check* or *Load Map* instruction. Leaves the contents of the specified accumulator unchanged. The format of the specified accumulator is as follows:



BITS	NAME	CONTENTS or FUNCTION
0-16	---	Reserved for future use.
17-21	Logical Page	Number of the logical block for which the check is requested.
22-24	Map	Specify which map should be used for the check as follows: 000 User A 001 Reserved for future use 010 User B 011 Reserved for future use 100 Data channel A 101 Data channel C 110 Data channel B 111 Data channel D
25-31	---	Reserved for future use.

Map Single Cycle Disable User Mode
NIOP *ac*,MAP



Issued from unmapped mode, the instruction maps one memory reference using the last user map; issued from User mode with **LEF** mode and I/O protection disabled, the instruction simply turns off the map, returning it to unmapped mode. It is used by the supervisor to access a user's memory space when only one or two references are required. It is also used by a privileged user to turn off memory mapping.

From Unmapped Mode
Enables the user map for one memory reference. Maps the first memory reference of the next **LDA, ELDA, STA** or **ESTA** instruction. After the memory cycle is mapped, the instruction again disables the user map.

> **NOTE:** *The interrupt system is disabled from the beginning of the* Map single cycle *instruction until after the next* **LDA, ELDA, STA** *or* **ESTA** *instruction.*

From User Mode
If **LEF** Mode and I/O protection is disabled, this instruction turns off the MAP. All subsequent memory references are unmapped until the map is reactivated with a *Load map status* instruction.

Programmable Interval Timer

Device Code
43_8 (Primary) **Priority Mask Bit**
11

Device Flag Commands

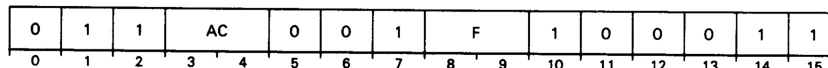
f=**S** Sets the Busy flag to 1 and the Done flag and interrupt request flag to 0; begins the counting cycle.

f=**C** Sets the Busy and Done flags and the interrupt request flag to 0; stops the counting cycle.

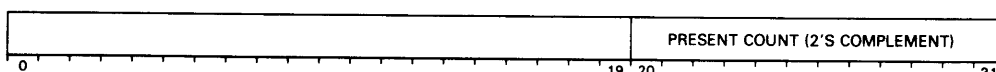
f=**P** No effect.

IORST Sets the Busy and Done flags, the interrupt request flag, the initial count register, the count output buffer, and the interrupt mask bit (bit 11) to 0; stops the counting cycle.

Read Count
DIA*[f]* *ac*,**PIT**

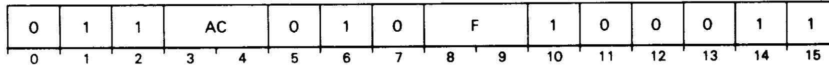


Places the value of the Programmable Interval Timer's Counter in bits 15-31 of the specified accumulator destroying the accumulator's previous contents. After the data transfer, performs the function specified by *F*. The format of the specified accumulator is as follows:



BITS	NAME	CONTENTS or FUNCTION
0-19 20-31	--- Count	Reserved for future use. Current value of the PIT counter within one count cycle.

Specify Initial Count
DOA*[f]* *ac*,PIT



Loads bits 16-31 of the specified accumulator into the Programmable Interval Timer's Initial Count Register. After the data transfer, performs the function specified by *F*. The contents of the specified accumulator remain unchanged; the format of the accumulator is as follows:

BITS	NAME	CONTENTS or FUNCTION
0-19 20-31	--- Initial Count	Reserved. Two's complement of the number of 100 microsecond intervals between interrupts.

Real Time Clock

Device Code
14_8 (Primary)

Priority Mask Bit
13

Device Flag Commands

f=**S** Sets the Busy flag to 1, and the Done flag and interrupt request flag to 0; enables RTC interrupts.

f=**C** Sets the Busy and Done flags and the interrupt request flag to 0; disables RTC interrupts.

f=**P** No effect.

IORST Sets the Busy and Done flags, the interrupt request flag, the interrupt mask bit (bit 13), and the clock frequency select bits to 0; disables RTC interrupts.

Select RTC Frequency
DOA*[f]* *ac*,RTC

0	1	1	AC		0	1	0	F		0	0	1	1	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The clock frequency is set according to bits 30-31 of the specified accumulator. The contents of the specified accumulator remain unchanged. Bits 0-29 of the specified accumulator are ignored. The format of the specified accumulator is as follows:

	RTC
0–29	30–31

BITS	NAME	CONTENTS or FUNCTION
0-29	---	Reserved for future use. (Set to 0)
30-31	RTC	Selects the clock frequency as follows: 00 ac line frequency 01 10Hz 10 100Hz 11 1000Hz

Primary Asynchronous Line Input

Device Code
10_8 (Primary)

Priority Mask Bit
14

Device Flag Commands

f=S Sets the Busy flag to 1 and the Done flag to 0.

f=C Sets the Busy and Done flags and the interrupt request flag to 0.

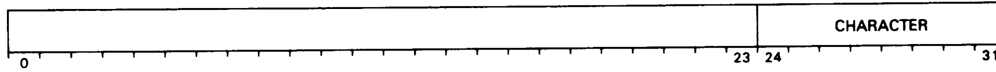
f=P No effect.

IORST Sets the Busy and Done flags, the interrupt request flag, and the interrupt mask bit (bit 14) to 0.

Read Character Buffer
DIA*[f]* *ac*,TTI

0	1	1	AC		0	0	1	F		0	0	1	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Places the contents of the controller's input buffer in bits 24-31 of the specified accumulator. After the data transfer, sets the controller's Busy and Done flags according to the function specified by *F*. The format of the specified accumulator is as follows:



BITS	NAME	CONTENTS or FUNCTION
0-23	----	Reserved for future use.
24-31	Character	The 8 bit character or 7 bit character with parity in bit position 8 read from the input buffer.

Primary Asynchronous Line Output

Device Code
11_8 (Primary)

Priority Mask Bit
15

Device Flag Commands

f=S Sets the Busy flag to 1 and the Done flag to 0; begins transmission of the character contained in the output buffer.

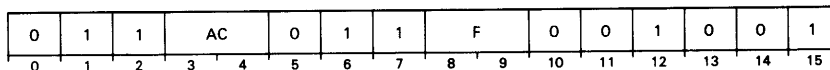
f=C Sets the Busy and Done flags and the interrupt request flag to 0.

f=P No effect.

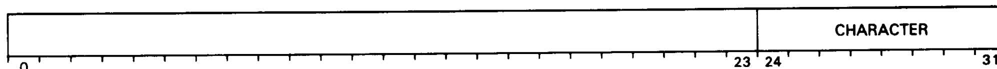
IORST Sets the Busy and Done flags, the interrupt request flag, and the interrupt mask bit (bit 15) to 0.

Load Character Buffer

DOA*[f]* *ac*,TTO



Loads bits 24-31 of the specified accumulator into the controller's output buffer. After the data transfer, sets the controller's Busy and Done flags according to the function specified by *F*. The contents of the specified accumulator remain unchanged. The format of the specified accumulator is as follows:



BITS	NAME	CONTENTS or FUNCTION
0-7	----	Reserved for future use.
8-15	DATA	The 8-bit character or 7-bit character with parity in bit position 8 to be placed in the output buffer.

Chapter 18
IOP Communication Instruction Dictionary

This chapter lists the IOP instructions used to communicate with the host.

IOP/Host Communication

Device Code, IOP to Host and local I/O
4_8 (Primary)

Priority Mask Bit, IOP to Host
5 (Primary)

Priority Mask Bit, IOP Map
None

Device Flag Commands

Device flag commands allow the IOP to interrupt the host. These commands do this by setting Interrupt Request flags, Busy flags, and Done flags to initiate the interrupt. The IOP and the host each have one Busy flag, one Done flag, and one Interrupt Request flag. The host's flags are located in and are tested by the IOP. The IOP's flags are located in and are tested by the host. The device flag commands for the four IOP instructions are:

f=S Sets the IOP Done and Host interrupt request flags to 1.

f=C Sets the Host Busy and IOP interrupt request flags to 0.

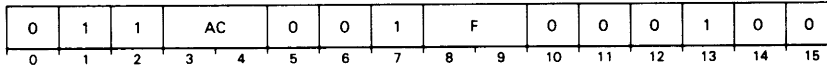
f=P Sets the Host Done and IOP interrupt request flags to 0.

IORST Sets bits 2-4, 14 and 15 of the map status and parity control register, Host Done flag, IOP interrupt request flag, and IOP interrupt mask bits to 0.

NOTE: *A IOP Run flag is read as Busy by the Host. It is set to 1 when the IOP is not in a halted state.*

Read Map Status and Parity Control

DIA*[f]* *ac*,**IOPI**

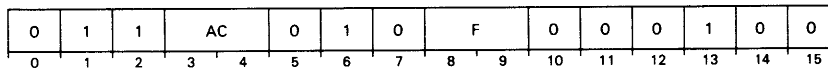


Loads bits 16-31 of the specified accumulator with the map status and parity control bits, as well as the MAP host/local flag selected by the previous **DOA** instruction. The format of the accumulator is shown below:

BITS	NAME	CONTENTS or FUNCTION
0-17	---	Reserved for future use.
18	A	Address Save on
19	P	Parity Test on
20	PE	Parity Checking on
21-26	---	Reserved for future use.
27	HP	Bit for MAP page selected by previous **DOA**: 0 = page is mapped into SPU local memory. 1 = page is mapped into host memory.
28-29	SEL	Currently selected host data channel:
30	DCH	Current state of DCH MODE.
31	UM	Current state of USER MODE.

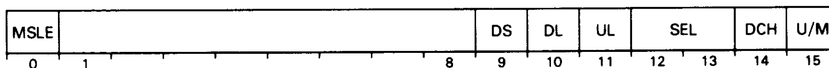
Control (and Select) Map and Page/Parity

DOA*[f]* *ac*,**IOPI**

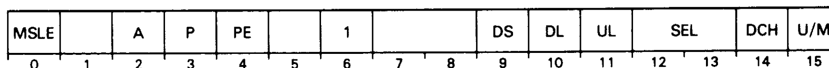


Stores the specified accumulator into the map status and parity control register as summarized below. Depending on the contents of bit 0 in the specified accumulator, the contents of bits 12-15 may be ignored; also, the contents of bit 6 controls the interpretation of bits 1-5. Consequently, the accumulator may take one of the following formats:

Format 1, Control Map Only



Format 2, Control Map and Parity



Format 3, Control Map and Select Page

MSLE	LOGICAL	0		DS	DL	UL	SEL	DCH	U/M
0	1 — 5	6	7 — 8	9	10	11	12 — 13	14	15

Bits	Name	Contents or Function
0	MSLE	Permit loading of map status (bits 12-15).
1	---	Used for map reading only.
2	A	Address Save on.
3	P	Parity Test on.
4	PE	Parity checking on.
1-5	LOGICAL	Selects Logical Page
6	PCL	Permit loading of parity control flags.
7-8	---	Reserved for future use.
9	DS	Suppress loading of DCH map select (bits 12,13).
10	DL	Suppress loading of DCH mode (bit 14).
11	UL	Suppress loading of USER mode (bit 15).
12,13	SEL	Selects host data channel map for references to host memory:
14	DCH	DCH mode on.
15	U/M	USER mode on.

Control Timer

DOC*[f]* *ac*,IOPI

0	1	1	AC	1	1	0	F	0	0	0	1	0	0
0	1	2	3 — 4	5	6	7	8 — 9	10	11	12	13	14	15

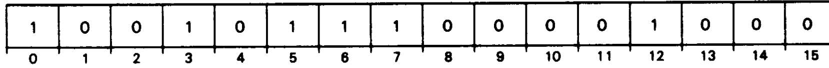
Stores the contents of bits 16-31 of the specified accumulator into the timer control register and performs the function determined by the bits as shown below.

	T		I	
0 — 15	16	17 — 21	22	23 — 31

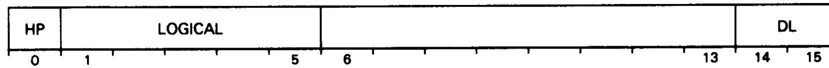
BITS	NAME	CONTENTS or FUNCTION
0-15	---	Reserved for future use.
16	T	0 = Stop timer 1 = start timer.
17-21	---	Reserved for future use.
22	I	If 1, clear interrupt.
23-31	---	Reserved for future use.

NOTE: *The timer interrupt may be disabled without stopping the timer by setting bit 2 in the IOP interrupt mask register.*

Load Map
LMP



Load a number of map entries from a table in memory. Bits 16-31 of AC2 must contain the starting address of the table. Bits 16-31 of AC1 must contain the number of entries to load. AC0 and AC3 are unused. The format of each entry to be loaded into the IOP map is shown below.



Bits	Name	Function
0	HP	0 = map page into local memory. 1 = map page into host memory.
1-5	LOGICAL	Logical page number.
6-14	---	Reserved for future use.
15	DL LD	0 = load entry into USER map. 1 = load entry into DCH map.

Appendix A

The ASCII Character Set

DECIMAL	OCTAL	HEX	KEY SYMBOL	MNEMONIC
0	000	00	↑@	NUL
1	001	01	↑A	SOH
2	002	02	↑B	STX
3	003	03	↑C	ETX
4	004	04	↑D	EOT
5	005	05	↑E	ENQ
6	006	06	↑F	ACK
7	007	07	↑G	BEL
8	010	08	↑H	BS (BACKSPACE)
9	011	09	↑I	TAB
10	012	0A	↑J	NEW LINE
11	013	0B	↑K	VT (VERT.TAB)
12	014	0C	↑L	FORM FEED
13	015	0D	↑M	CARRIAGE RETURN
14	016	0E	↑N	SO
15	017	0F	↑O	SI
16	020	10	↑P	DLE
17	021	11	↑Q	DC1
18	022	12	↑R	DC2
19	023	13	↑S	DC3
20	024	14	↑T	DC4
21	025	15	↑U	NAK
22	026	16	↑V	SYN
23	027	17	↑W	ETB
24	030	18	↑X	CAN
25	031	19	↑Y	EM
26	032	1A	↑Z	SUB
27	033	1B	ESC	ESCAPE
28	034	1C	↑\	FS
29	035	1D	↑]	GS
30	036	1E	↑↑	RS
31	037	1F	↑←	US

DECIMAL	OCTAL	HEX	KEY SYMBOL
32	040	20	SPACE
33	041	21	!
34	042	22	" (QUOTE)
35	043	23	#
36	044	24	$
37	045	25	%
38	046	26	&
39	047	27	' (APOS)
40	050	28	(
41	051	29	)
42	052	2A	*
43	053	2B	+
44	054	2C	, (COMMA)
45	055	2D	-
46	056	2E	. (PERIOD)
47	057	2F	/
48	060	30	0
49	061	31	1
50	062	32	2
51	063	33	3
52	064	34	4
53	065	35	5
54	066	36	6
55	067	37	7
56	070	38	8
57	071	39	9
58	072	3A	:
59	073	3B	;
60	074	3C	<
61	075	3D	=
62	076	3E	<
63	077	3F	?
64	100	40	@

DECIMAL	OCTAL	HEX	KEY SYMBOL
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D	]
94	136	5E	↑ OR ∧
95	137	5F	← OR —
96	140	60	` (GRAVE)

DECIMAL	OCTAL	HEX	KEY SYMBOL
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~ (TILDE)
127	177	7F	DEL (RUBOUT)

Appendix B
Context Block Format

The context block can be from 20 to 43 double-words long. The block has the format shown in Table B.1.

Words in Block	Contents
0-1	PSR, 16 zeroes, micro state block
2-3	GR2
4-5	AC0
6-7	AC1
8-9	AC2
10-11	AC3
12-13	carry, PC
14-15	ATU0 — See Table B.2.
16-17	LAR — Contains logical word address that caused the fault. If the original logical address was a word address, bit 0 of LAR is undefined. If the original address was a byte address, then bit 0 is the byte indicator.
18-19	ATUF
20-21	GR3
22-23	LINK
24-25	GR1
26-27	GR0
28-29	GR4
30-31	Micro stack count (number of double words that follow)
32-61 (max)	Micro stack save area (2 to 8 double words)
62-63	GR7
64-65	GR6
66-67	GR5
68-69	DSR
70-71	CPDR
72-73	MREG
74-75	Micro counter
76-77	Q
78-79	1 Scratch pad location
80-81	IPS

Table B.1 Context block format

The contents of ATU0 are shown in Table B.2.

Bits	Meaning
0	Page fault on reading an instruction.
1–3	Reserved.
4	Inclusive OR of bits ATU0 0 and 5.
5	Page fault due to instruction cache reference.
6	Page fault on reading a double word straddling a page boundary.
7	Reserved.
8	0 – – read request for logical location causing page fault.
	1 – – write request for logical location causing page fault.
9–10	00 – – word operand of word causing page fault.
	01 – – left byte of word causing page fault.
	10 – – double word causing page fault.
	11 – – right byte of word causing page fault.
11–15	Reserved.
16	Page fault occurred during effective address calculation.
17–19	Number of ring currently executing at time of page fault.
20–25	Reserved.
26–27	Modified and referenced bits of physical page last referenced.
28–31	Referenced bits of the 8 Kbyte physical page last referenced.

Table B.2 ATU0 contents

NOTE: *If either bit 5 or 16 of ATU0 is 1, the context block is only 20 words long.*

The format of ATUF is shown in Table B.3.

Bits	Contents
0–15	Reserved.
16–23	Scratch pad address register.
24–27	Reserved.
28–31	If bit 28 is 0, then bits 29-31 specify the fault code for a protection fault.
	If bit 28 is 1, then bits 28-31 are mutually exclusive:
	Bit 29 = 1: page fault occurred when referencing a page
	Bit 30 = 1: page fault occurred when referencing a page table (occurs during a two level reference only).
	Bit 31 = 1: page table depth fault.

Table B.3 ATUF format

All other words in the context block contain information used by the microcode and other internal systems. The floating point state is not saved in the context block. To save this information, use a *Push Floating Point State* instruction.

Note that the processor assumes that the context block save area, the pointer to the save area, and all indirect chains used are aligned on double-word boundaries. User programs should also follow this assumption.

Appendix C
MV/8000—C/350 Program Combinations

The MV/8000 computer allows you to execute C/350 programs when operating under the AOS/VS operating system. Programs that include C/350 instructions must meet certain requirements for them to be executed properly, however. This Appendix first describes what happens when C/350 instructions are used, then describes how to mix C/350 and MV/8000-specific instructions in the same program.

Note that the AOS/VS operating system handles interrupts caused by either MV/8000 or C/350 instructions. This means that programs do not have to use different interrupt sequences depending on the type of instruction that caused the interrupt.

Using C/350 Instructions

C/350 instructions that specify an accumulator as a source of information (such as CLM) do not change the contents of the accumulator. C/350 instructions that load data into an accumulator alter bits 16–31 of the referenced accumulator. Bits 0–15 are undefined.

When PC- or accumulator-relative modes of addressing are used with C/350 memory reference instructions, the processor forms a 31-bit result from the sum of the index register contents and the displacement.

All C/350 program flow instructions and all C/350 instructions that load an effective address will alter the wide PC or an accumulator. During program counter modification, the PC's 16 most significant bits are not altered. Bits 17–31 contain the C/350 effective address:

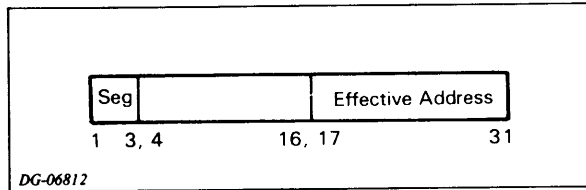
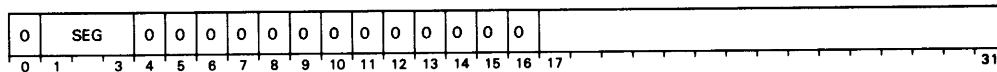


Figure C.1

During effective address calculation, the accumulator's 17 most significant bits appear as:



When a C/350 ALC instruction with the *no-load* option (bit 12 contains a 1) is specified, the accumulators remain unchanged. This option is particularly convenient when testing for some condition without destroying the contents of ACD.

If the C/350 MAP is enabled, MV/8000-specific instructions cannot be executed. Any attempt to do so causes a narrow I/O protection fault to occur and bit 2 of the MAP status register to be set to 1. The processor pushes a narrow return block onto the stack. The return address in the return block points to the instruction that caused the fault.

If the ATU is enabled, the C/350 **LMP, SYC,** or any C/350 MAP instructions cannot be executed. Any attempt to do so results in a protection fault. AC1 will contain a 9.

When the C/350 **XCT** instruction is specified, the processor interprets the 16 least significant bits of the specified accumulator. If these bits specify an MV/8000-specific instruction, the processor executes that instruction.

Expanding a C/350 Program to Run on the MV/8000 Computer

The appropriate MV/8000-specific instructions can be added to C/350 programs to produce a number of results:

- Expand the program beyond 64KB,
- Use large arrays and other expanded data areas,
- Use the MV/8000 32-bit fixed arithmetic.

There are several ways to expand C/350 programs beyond 64 Kbytes. One of the most reliable is to rewrite one of the subroutines so that it contains MV/8000-specific instructions, then place it in the high end of memory. This subroutine must be referenced with a **WJSR, LCALL,** or **XCALL** instruction in the main program, and the **WSAVE** and **WRTN** instructions must be used in the subroutine, so that control can transfer properly. Also, ensure that the subroutine uses MV/8000-specific instructions for all memory references.

To use expanded data areas, use MV/8000-specific instructions for all memory references, and 32-bit fixed-point arithmetic for all address calculations. This can be done by changing just the portions of the program that will reference the expanded data area. New subroutines can also be created that maintain these expanded data areas as well as reference them. If the latter approach is used, ensure that subroutines referenced by 32-bit addresses use an MV/8000 save/return setup. If a C/350 **SAVE/RTN** instruction combination is used in such a subroutine, the processor will not save the 16 high order bits of the four accumulators upon entering the subroutine.

MV/8000-specific instructions are required to perform all 32-bit fixed point arithmetic. As described above, altering a program to use 32-bit arithmetic means changing only the affected parts of the C/350 program, or writing new subroutines.

Calling a C/350 Subroutine From an MV/8000 Program

An MV/8000 program can call a C/350 subroutine, though such an arrangement requires many changes to the C/350 subroutine. These changes are shown in Table C.1.

Change to C/350 Subroutine	Reason for Change
Replace **SAVE** and **RTN** with **WSAVE** and **WRTN**	Must allow the main program to call this subroutine from high addresses (PC > 16 bits long). Must be able to handle 32-bit data in accumulators.
All references from outside routines may need new memory reference instructions	Must be able to handle 32-bit arguments passed from other routines. Must be able to call other subroutines located in high address space.
Short negative reference on the stack may require new displacements.	Using **WSAVE** in this subroutine changes the size of the stack block pushed. This means that the processor must recalulate a short negative reference.
Check routines that are referenced by a **JSR** through page zero. to save the 32-bit PC.	Long addresses require 32 bits and may cause you to run out of page zero locations. Use **WJSR**

Table C.1

Appendix D
Anomolies

This appendix explains differences that affect the conversion of C/350 programs to MV/8000 programs.

MV/8000 Instruction Opcodes

The processor recognizes the MV/8000-specific instructions supported by this machine by the instruction opcodes. All MV/8000-specific instructions are an outgrowth of the C/350 ALC no load-always skip opcode and the C/350 **XOP** and **XOP1** opcodes. This means that on this machine you cannot use any C/350 program that contains these instructions. The processor will interpret these instructions as MV/8000 instructions, not as C/350 instructions.

Program Counter Wraparound

The MV/8000 program counter is 31 bits wide. Bits 1–3 specify the current ring of execution. Bits 4–31 specify an address. When the PC is incremented, only bits 4–31 take part in the increment. This means that the PC will always contain an address in the current ring. PC wraparound will *not* occur at 77777_8 as it does in the C/350.

Float/Fixed Conversions

When the processor converts a floating point number to a fixed point integer, it converts the largest negative number correctly without **MOF** overflow. For single precision, the processor converts the integer portion of floating point numbers to an integer in the range -32,768 to +32,767 inclusive. For double precision, the processor converts the integer portion to an integer in the range -2,147,483,648 to 2,147,483,647 inclusive.

Address Wraparound

When using the C/350 **BAM, BLM, CMP, CMT, CMV, CTR,** and **EDIT** instructions, address wraparound may not occur at 77777_8. This means that a C/350 program could

possibly generate logical addresses greater than 64 kB. In this situation, results are undefined.

If any of the instructions listed in the paragraph above move data backwards (i.e., into descending addresses) and cross a ring boundary, a protection fault occurs. AC1 will contain the protection fault code 4.

C/350 Signed Divide Instructions

When the C/350 **DIVS** or **DIVX** instructions produce a result of –32,768, the MV/8000 processor sets carry to 0 (meaning no overflow). When this instruction is used on the C/350, the processor sets carry to 1 (meaning overflow). MV/8000 divide instructions set *overflow* to 0 when –32,768 results.

C/350 Vector and NIO Instructions

There are three additional instruction encodings for the C/350 **VCT** and **NIO** instructions. They are the presently defined ECLIPSE encoding with bits 3 and 4 being (0,1), (1,0), and (1,1). Consequently all C/350 **INTA** instructions should always use the encoding **00** in bits 8 and 9 to specify "unchanged" for the Interrupt On flag. A **11** encoding in bits 8 and 9 indicates the MV/8000 **XVCT** instruction.

Floating Point Trap

The MV/8000 processor responds to floating-point traps upon completion of the floating point instruction that caused the fault. In the C/350, the response to a floating point trap occurs when the *next* floating point instruction is encountered. In either case, the value of the floating point PC ic the same; that is, it contains the address of the floating point instruction that caused the fault.

Floating Point Numerical Algorithms

The C/350 floating-point loads (**FLDS, FLDD**) do not correct impure zero input. All loads simply move the memory operand to the specified FPAC. No normalization and correction to true zero is performed. The *Z* and *N* bits of the FPSR are set to reflect the loaded operand only if the operand is normalized. The *Z* and *N* flags are undefined if the operand is unnormalized.

For all instructions, *true zero* is guaranteed to be generated for valid inputs *only*. If an impure zero is generated with invalid inputs, the result is not necessarily converted to true zero.

The C/350 **FFAS** and **FFMD** instructions leave the *Z* and *N* bits of the FPSR unchanged.

Otherwise, when bit 8 of the FPSR is a 0, the results of the floating point computations performed on the MV/8000 processor are identical to those obtained on the C/350.

C/350 Commercial Faults

A C/350 comercial fault loads different information it AC0, AC2, and AC3 after the fault is taken. The size of the return block, the fault code in AC1, and the meaning of the PC in the return block are identical to the results obtained on the C/350.

C/350 MAP Instructions

As noted elsewhere in this manual, an attempt to execute the C/350 **SYC**, **LMP**, and MAP related instructions when the MV/8000 ATU is enabled results in a protection violation (code = 9 in AC1). Due to the nature of the MV/8000 ATU and virtual address space, these instructions have no rational meaning when the MV/8000 ATU is enabled.

Appendix E
C/350 Memory Allocation and Protection

The MV/8000 supports the C/350 MAP so that C/350 programs can be run on the MV/8000 without changing them. Like the MV/8000 ATU, the C/350 MAP performs address translation, and can run in mapped or unmapped mode. The MAP also has a *Lef* mode that determines whether the C/350 *Load Effective Address* instruction or the standard I/O instructions will be executed. MAP execution in any mode is closely governed by a protection system that checks for illegal references and protects important data.

> NOTE: *In the following section, "MAP" refers to the Memory Allocation and Protection unit, whereas "map" refers to a set of memory translation functions used by the MAP.*

MAP Functions

The MAP performs the following tasks:

- Translates virtual addresses to physical addresses
- Allows memory to be shared between users

Address Translation

The primary function of the MAP is address translation. The map divides each user's logical address space into 2 Kbyte pages and associates each logical page with a corresponding physical page. The address space the user sees is unchanged, but the map now translates each logical address into a physical address before memory is actually accessed.

Note that a user's physical pages can be in any particular order in physical memory. This means that the supervisor (the part of the operating system that controls system functions) can select unused pages for a new user without concern for maintaining any particular arrangement. It also means that physical memory can be used almost completely, since no contiguous blocks of memory larger than 2 Kbytes are required.

Sharing Physical Memory

The MAP can allow several users to use the same section of physical memory. This is useful if several users want to use a commonly used routine, such as trigonometric tables. Without this ability the processor would have to make a copy of the routine for each user.

MAP Modes

The MAP can operate in two different modes: mapped and unmapped. The processor generally operates in mapped mode. Unmapped mode is used to perform diagnostic and certain MAP functions.

Mapped Mode

In mapped mode, the MAP provides two types of maps:

- *User maps* are a set of address translation functions defined for a particular user. They translate logical addresses to physical addresses when the processor encounters memory reference instructions in a user's program.
- *Data channel maps* are a set of address translation functions defined by the user-specified map. These are defined for the memory references of a data channel used by a particular device. They translate logical addresses to physical addresses when data channel devices address the memory.

User Maps

Each user requires a separate user map. The MAP can hold two user maps, but only one can be enabled at any one time. This means that when two users exist, the processor specifies the user map for each and loads them into the MAP. The supervisor can then enable one or the other as needed.

If there are more than two users, the processor must load new user maps as they are needed. In some operating systems, the operating system itself uses one of the user maps, so the processor must load a new user map each time another user requires service. This is not as much of an overhead burden as it sounds; the *Load Map* instruction loads a complete map with one instruction and uses relatively little time.

Data Channel Maps

Data channel devices can access memory without direct control from the user's program. This offers no assurance that the proper user map will still be enabled at the time of the data channel request. Therefore, the MAP uses separate data channel devices when such a device accesses memory.

The MAP can hold four data channel maps. Enabling data channel mapping enables all four data channel maps at the same time. The I/O controller making the reference chooses which of the four maps to use. Those controllers not equipped to make this distinction use data channel map A by default. See the *Programmer's Reference Manual - Peripherals* (DGC No. 015-000021) for more information.

Unmapped Mode

This mode is used for diagnostic purposes and for certain MAP control functions. In unmapped mode, the processor does not translate addresses in the range $0\text{-}75777_8$ (which form logical pages 0-30). The processor uses a special map to translates addresses in the range $76000\text{-}77777_8$ for logical page 31. This allows you to access selected portions of user space while in unmapped mode.

MAP Protection Capabilities

The C/350 MAP is equipped with protection functions to protect the integrity of the system. It does this by preventing unauthorized access to certain parts of memory or to I/O devices. For example, a set of trigonometric functions stored in a section of memory accessible to all users can be *write protected* so that users can read the functions but cannot change them.

Validity Protection

Validity protection protects a user's memory space from inadvertant access by another user, thereby preserving the integrity and privacy of the user's memory space. When a user's map is specified, the blocks of logical addresses required by the user's program are linked to blocks of physical addresses. The remaining (unused) logical blocks are declared invalid to that user, and an attempt to access them will cause a validity protection fault.

Validity protection is always enabled, so the supervisor's responsibility is limited to declaring the appropriate blocks of logical addresses invalid.

Write Protection

Write protection permits users to read the protected memory addresses, but not to write into them. In this way, the integrity of common areas of memory can be protected. An attempt to write into a write protected area of memory will cause a protection fault.

A block of addresses is write protected when the map is specified. Write protection can be enabled or disabled at any time by the supervisor.

Indirect Protection

An indirection loop occurs when the effective address calculation follows a chain of indirect addresses and never finds a word with bit 0 set to 0. Without indirect protection, the CPU would be unable to proceed with any further instructions, thus effectively halting the system.

With indirect protection enabled, a chain of 15 indirect references will cause a protection fault. Indirect protection can be enabled or disabled at any time by the supervisor.

I/O Protection

I/O protection protects the I/O devices in the system from unauthorized access. In many systems, all I/O operations are performed through operating system calls. Clearly, it is undesirable to permit individual users to execute I/O instructions, since this will interfere with the operating system. If a user with I/O protection enabled attempts to execute an I/O instruction, a protection fault will occur. I/O protection can be enabled or disabled at any time.

MAP Protection Faults

When a user attempts to violate one of the enabled types of protection, a protection fault occurs, as follows:

- The current user map is disabled.
- A 5-word return block is pushed onto the system stack.
- Control is transferred to the protection fault handler, through an indirect jump via location 3.

The system programmer must supply a protection fault handler which determines the type of fault that occurred (using the *Read Map Status* instruction), and then takes the appropriate action.

A protection fault can occur at any point during the execution of an instruction. Therefore, the return address in the fifth word of the return block is not always correct. For I/O protection faults, however, the fifth word will always be the logical address of the instruction following the instruction that caused the fault.

Load Effective Address Mode

The *Load Effective Address* instruction has the same format as some of the I/O instructions. The MAP therefore has a *Lef* mode bit which determines whether an I/O format instruction will be interpreted as an I/O or a **LEF** instruction. When the *Lef* mode bit is 1 (*Lef* mode enabled), all I/O format instructions are interpreted as *Load Effective Address* instructions. When the *Lef* mode bit is 0, all I/O format instructions are interpreted as I/O instructions.

The *Load Effective Address* instruction is very useful for quickly loading a constant into an accumulator. In addition, a user operating in the *Lef* mode is effectively denied access to any I/O devices, because all I/O and *Lef* instructions are interpreted as *Lef* instructions in this mode. Thus, *Lef* mode can be used for I/O protection. Note, however, that no indication is given if an I/O instruction is interpreted as a *Lef* instruction.

When not operating in the *Lef* mode, all *Lef* and I/O instructions are interpreted as I/O instructions. With I/O protection enabled, these instructions will cause a protection fault in the normal manner. With I/O protection disabled, the *Lef* instruction will be executed as an I/O instruction if possible.

Initial Conditions

At power up, the user maps and the data channel maps are undefined, the MAP is in unmapped mode, and unmapped logical page 31 is mapped to physical page 31.

After an *I/O Reset*, the MAP is in unmapped mode, the data channel maps are disabled, and unmapped logical page 31 is mapped to physical page 31.

MAP Instructions

The MAP instructions control the actions of the MAP. They are used by the supervisor program to change the mapping functions or check the status of the various maps.

> **NOTE:** *MAP instructions* can *be executed in mapped mode if I/O protection and* Lef *mode are disabled for that user. When executed in mapped mode, the* Read Map Status, Initiate Page Check, *and* Page Check *instructions will return the desired information without changing the map. The* Map Single Cycle *instruction will disable the user map after the next memory reference. The remainder of the instructions will change the map while the map is enabled, with undesirable results for this user, another user, or the system as a whole.*
>
> *Enabling* Lef *mode only will convert all I/O instructions (including MAP instructions) to* Lef *instructions. The* Load Map *instruction, however, does not use the I/O format and therefore can still be executed. Enabling both* Lef *mode and I/O protection will prevent execution of the* Load Map *instruction.*

The MAP instructions are shown in Table E.1. All except *Load Map* are in I/O format using the device mnemonic MAP.

MNEM	Name	Action
DIA	Read Map Status	Reads the status of the current map.
DIC	Page Check	Provides the identity and some characteristics of the physical page corresponding to the logical page identified by the immediately preceding *Initiate Page Check* instruction.
DOA	Load Map Status	Defines the parameters of a new map.
DOB	Map Supervisor Page 31	Specifies the physical page corresponding to logical page 31 of the supervisor's address space.
DOC	Initiate Page Check	Identifies a logical page.
LMP	Load Map	Loads successive words from memory into the MAP where they are used to define a user or data channel map.
NIOP	Map Single Cycle	Maps one memory reference using the last user map.

Table E.1 C/350 MAP Instructions

Appendix F
Instruction Execution Times

The following table gives the average execution times of the instructions supported by the MV/8000. Times throughout are in microseconds.

ADC	0.33
ADD	0.33
ADDI	0.22
ADI	0.44
ANC	0.22
AND	0.33
ANDI	0.22
BAM	
BKPT	
BLM	
BTO	1.54 + ind
BTZ	1.54 + ind
CLM	
CMP	
CMT	
CMV	
COB	3.74
COM	0.33
CRYTC	0.22
CRYTO	0.22
CRYTZ	0.22
CTR	
CVWN	0.77
DAD	0.22
DEQUE	0.22
DHXL	1.10
DHXR	1.10
DIV	3.19
DIVS	3.74
DIVX	3.63
DLSH	3.52
DSB	0.22
DSPA	
DSZ	1.32
DSZTS	1.10
ECLID	
EDSZ	1.32
EISZ	1.32
EJMP	0.66
EJSR	0.66
ELDA	0.44
ELDB	0.44
ELEF	0.44
ENQH	3.52
ENQT	3.52
ESTA	0.44
ESTB	0.44
FAB	0.22
FAD	5.17

FAMD	5.83
FAMS	1.54
FAS	0.88
FCLE	0.66
FCMP	0.66
FDD	40.70 (FPSR8=0); 47.52 (FPSR8=1)
FDMD	41.36 (FPSR8=0); 48.18 (FPSR8=1)
FDMS	4.51 (FPSR8=0); 5.61 (FPSR8=1)
FDS	3.85 (FPSR8=0); 4.95 (FPSR8=1)
FEXP	0.44
FFAS	1.21
FFMD	2.13
FHLV	1.80 (FPSR8=0); 1.89 (FPSR8=1)
FINT	1.91
FLAS	0.77
FLDD	1.10
FLDS	0.66
FLMD	1.65
FLST	1.10
FMD	11.00
FMMD	11.66
FMMS	2.86
FMOV	0.66
FMS	2.20
FNEG	0.44
FNOM	1.21
FNS	0.22
FPOP	5.61
FPSH	4.40
FRH	0.44
FSA	0.44
FSCAL	2.75
FSD	5.17
FSEQ	0.88

FSGE	0.88
FSGT	0.88
FSLE	1.10
FSLT	0.88
FSMD	5.83
FSMS	1.54
FSND	0.88
FSNE	0.88
FSNER	0.66
FSNM	0.88
FSNO	0.88
FSNOD	0.88
FSNU	0.88
FSNUD	0.88
FSNUO	0.88
FSS	0.88
FSST	0.88
FSTD	0.88
FSTS	0.44
FTD	0.44
FTE	0.44
FXTD	0.22
FXTE	0.44
HLV	0.60
HXL	0.22
HXR	0.44
INC	0.33
IOR	0.22
IORI	0.22
ISZ	1.32
ISZTS	1.10
JMP	0.66
JSR	0.66
LCALL	2.20 intra ring, 6.82 cross ring
LCPID	
LDA	0.44
LDAFP	0.44
LDASB	0.44
LDASL	0.44
LDASP	0.44
LDATS	0.44
LDB	0.44
LDI	
LDIX	
LDSP	
LEF	0.44
LFAMD	5.83
LFAMS	1.54

LFDMD	41.36 (FPSR8=0); 48.18 (FPSR8=1)
LFDMS	4.51 (FPSR8=0); 5.61 (FPSR8=1)
LFLDD	1.10
LFLDS	0.66
LFLST	1.32
LFMMD	11.66
LFMMS	2.86
LFSMD	5.83
LFSMS	1.54
LFSST	0.88
LFSTD	0.88
LFSTS	0.44
LJMP	0.66
LJSR	0.66
LLDB	0.44
LLEF	0.44
LLEFB	0.66
LMRF	1.76
LNADD	0.66
LNDIV	3.74
LNDSZ	1.32
LNISZ	1.32
LNLDA	0.44
LNMUL	2.53
LNSTA	0.44
LNSUB	0.66
LOB	1.54
LPEF	0.88 + EFA
LPEFB	0.88 + EFA
LPHY	1.98
LPSHJ	1.76 + EFA
LPSR	0.44
LRB	1.76
LSBRA	8.58 + PURGE BUSY
LSBRS	8.36
LSH	2.42
LSN	0.44
LSTB	0.66
LWADD	5.50
LWDIV	5.50
LWDSZ	1.54
LWISZ	0.44

DG-07325

Instruction	Time
LWLDA	
LWMUL	3.19
LWSTA	0.66
LWSUB	0.33
MOV	2.20
MSP	3.30
MUL	2.20
MULS	2.20
NADD	0.22
NADDI	0.44
NADI	0.44
NBSAC	0.88 + 1.54 per search
NBSAS	0.88 + 1.54 per search
NBSE	0.88 + 1.54 per search
NBSGE	0.88 + 1.54 per search
NBSLE	0.88 + 1.54 per search
NBSNE	0.88 + 1.54 per search
NBSSC	0.88 + 1.54 per search
NBSSS	0.88 + 1.54 per search
NDIV	3.52
NEG	0.33
NFSAC	0.66 + 1.54 per search
NFSAS	0.66 + 1.54 per search
NFSE	0.66 + 1.54 per search
NFSGE	0.66 + 1.54 per search
NFSLE	0.66 + 1.54 per search
NFSNE	0.66 + 1.54 per search
NFSSC	0.66 + 1.54 per search
NFSSS	0.66 + 1.54 per search
NLDAI	
NMUL	2.31
NNEG	0.22
NSALA	0.88
NSALM	1.10
NSANA	0.66
NSANM	1.10
NSBI	0.44
NSUB	0.22
ORFB	
PATU	0.66 + Purge Busy
PBX	6.60 + executed instruction
POP	3.52
POPB	4.84
POPJ	3.96
PSH	3.08
PSHJ	3.52
PSHR	3.52
RRFB	1.54 + 0.66 per bit
RSTR	3.96
RTN	4.84
SAVE	4.18
SBI	0.44
SEX	0.22
SGE	
SGT	
SMRF	1.76
SNB	1.98
SNOVR	
SPSR	0.22

Instruction	Time
STA	0.44
STAFP	0.44
STASB	0.66
STASL	0.66
STASP	0.44
STATS	0.44
STB	0.44
STI	
STIX	
SUB	0.33
SYC	
SZB	1.98
SZBO	1.98
VBP	
VWP	
WADC	0.22
WADD	0.22
WADDI	0.44
WADI	0.44
WANC	0.22
WAND	0.22
WANDI	0.22
WASH	4.18
WBLM	
WBR	0.66
WBSAC	0.88 + 1.32 per search
WBSAS	0.88 + 1.32 per search
WBSE	0.88 + 1.32 per search
WBSGE	0.88 + 1.32 per search
WBSLE	0.88 + 1.32 per search
WBSNE	0.88 + 1.32 per search
WBSSC	0.88 + 1.32 per search
WBSSS	0.88 + 1.32 per search
WBTO	1.54
WBTZ	1.54
WCLM	
WCMP	
WCMT	
WCMV	
WCOB	7.26
WCOM	0.22
WCTR	
WDIV	5.28
WDIVS	5.94
WDPOP	0.44 for restart, 18.48 for restore
WFFAD	1.25
WFLAD	1.21
WFPOP	5.61
WFPSH	4.18
WFSAC	0.66 + 1.32 per search
WFSAS	0.66 + 1.32 per search
WFSE	0.66 + 1.32 per search
WFSGE	0.66 + 1.32 per search
WFSLE	0.66 + 1.32 per search
WFSNE	0.66 + 1.32 per search
WFSSC	0.66 + 1.32 per search
WFSSS	0.66 + 1.32 per search
WHLV	0.60
WINC	0.22
WIOR	0.22
WIORI	0.22
WLDAI	
WLDB	0.44

Instruction	Time
WLDI	
WLDIX	
WLOB	1.32
WLRB	1.76
WLSH	2.42
WLSI	0.77
WLSN	
WMOV	0.22
WMSP	1.32
WMUL	2.97
WMULS	2.86
WNADI	0.44
WNEG	0.22
WPOP	2.20
WPOPB	5.28
WPOPJ	1.10
WPSH	2.20
WRSTR	5.72
WRTN	6.16
WSALA	0.88
WSALM	1.10
WSANA	0.66
WSANM	1.10
WSAVR	2.64
WSAVS	2.64
WSBI	0.44
WSEQ	
WSGE	
WSGT	
WSKBO	1.10
WSKBZ	1.110
WSLE	
WSLT	
WSNB	1.98
WSNE	
WSSVR	3.08
WSSVS	3.08
WSTB	0.44
WSTI	
WSTIX	
WSUB	0.22
WSZB	1.98
WSZBO	1.98
WUSGE	
WUSGT	
WXCH	0.66
WXOP	
WXOP1	
WXOR	0.22
WXORI	0.22
XCALL	2.20 intra ring, 6.82 cross ring
XCH	0.66
XCT	0.66 + executed instruction
XFAMD	5.83
XFAMS	1.54
XFDMD	41.36 (FPSR8=0); 48.18 (FPSR8=1)
XFDMS	4.51 (FPSR8=0); 5.61 (FPSR8=1)
XFLDD	1.10
XFLDS	0.66
XFMMD	11.66
XFMMS	2.86
XFSMD	5.83
XFSMS	1.54

DG-07325

Appendix G
Floating Point Operations

As noted in Chapter 8, the processor performs floating point operations between floating point operands. This appendix describes these operations and how they are performed.

The four floating point operations are:

- Addition,
- Subtraction,
- Multiplication,
- Division.

Floating Point Addition

The processor compares the exponents of the two floating point operands and finds the absolute value of the difference between them (a value *d*). The mantissa of the operand with the smaller exponent is shifted *d* hex digits to the right. The processor then adds the two mantissas using the rules of algebra to get the result mantissa and sign. The result exponent has the same value as does the exponent of the larger of the two operands.

If the result mantissa is too large, the processor shifts it to the right one hex digit, places the value 0001 in the mantissa's most significant digit, and adds one to the result mantissa.

Floating Point Subtraction

The processor complements the sign bit of the operand to be subtracted (located in ACS or in memory, depending on the instruction), then performs a floating point add as described above.

Floating Point Multiplication

The processor uses the rules of algebra to multiply the mantissas of the two operands together and determine the sign. The exponents of the operands are added together.

Floating Point Division

The operand in ACS or memory is called the divisor; that in ACD, the dividend. If the divisor is 0, the processor sets the **DVZ** flag in the FPSR to 1 and ends the instruction.

If the divisor is not 0, the processor compares the mantissas of the two operands. If the divisor mantissa is greater than or equal to the dividend mantissa, the processor shifts the dividend mantissa to the right one hex digit, places 0000 in the dividend mantissa's most significant digit, and adds one the the dividend exponent. Using the rules of algebra, the processor divides the dividend mantissa by the divisor mantissa, and determines the sign. The divisor mantissa is subtracted from the dividend exponent.

Appendix H
Standard I/O Device Codes

OCTAL DEVICE CODES	MNEMONIC	PRIORITY MASK BIT	DEVICE NAME	OCTAL DEVICE CODES	MNEMONIC	PRIORITY MASK BIT	DEVICE NAME
00	----	--	Unused	41[3]	DPO	8	IPB full duplex output
01	----	--	Unused	40	SCR	8	Synch. communication receiver
02	ERCC	--	Error checking and correction	41	SCT	8	Synch. communication transmitter
03	MAP	--	Memory allocation and protection unit	42	DIO	7	Digital I/O
04				43	DIOT	6	Digital I/O timer
					PIT	6	Programmable Interval Timer
05				44	MXM	12	Modem control for MX1/MX2
06	MCAT	12	Multiprocessor adapter transmitter	45			
07	MCAR	12	Multiprocessor adapter receiver	46	MCAT1	12	Second multiprocessor transmitter
10	TTI	14	TTY input	47	MCAR1	12	Second multiprocessor receiver
11	TTO	15	TTY output	50	TTI1	14	Second TTY input
12	PTR	11	Paper tape reader	51	TTO1	15	Second TTY output
13	PTP	13	Paper tape punch	52	PTR1	11	Second paper tape reader
14	RTC	13	Real-time clock	53	PTP1	13	Second paper tape punch
15	PLT	12	Incremental plotter	54	RTC1	13	Second real-time clock
16	CDR	10	Card reader	55	PLT1	12	Second incremental plotter
17	LPT	12	Line printer	56	CDR1	10	Second card reader
20	DSK	9	Fixed head disc	57	LPT1	12	Second line printer
21	ADCV	8	A/D converter	60	DSK1	9	Second fixed head disc
22	MTA	10	Magnetic tape	61	ADCV1	8	Second A/D converter
23	DACV	--	D/A converter	62	MTA1	10	Second magnetic tape
24	DCM	0	Data communications multiplexor	63	DACV1	--	Second D/A converter
25				64			
26	DKB	9	Fixed head DG/Disc	65			
27	DPF	7	DG/Disc storage subsystem	66	DKB1	9	Second Fixed Head DG/Disc
30	QTY	14	Asynch. hardware multiplexor	67	DPF1	7	Second DG/Disc storage subsystem
30	SLA	14	Synchronous line adapter	70	QTY1	14	Second asynch. hardware mux
31[1]	IBM1	13	IBM 360/370 interface	70	SLA1	14	Second synchronous line adapter
32	IBM2	13	IBM 360/370 interface	71[1]		13	Second IBM 360/370 interface
33	DKP	7	Moving head disc	72		13	Second IBM 360/370 interface
34[1]	CAS[1]	10	Cassette tape	73	DKP1	7	Second moving head disc
	DCU[4]	4	Data Control Unit				
34	MX1	11	Multiline asynchronous controller	74	CAS1	10	Second cassette tape
35	MX2	11	Multiline asynchronous controller	74[1]		11	Second multiline asynch. controller
36	IPB	6	Interprocessor bus--half duplex	75		11	Second multiline asynch. controller
37	IVT	6	IPB watchdog timer	76	DPU	4	DCU To Host Interface
40[2]	DPI	8	IPB full duplex input	77	CPU	--	CPU and console functions

DG-07317

1. Code returned by INTA and used by VCT
2. Can be set up with any unused even device code equal to 40 or above
3. Can be set up with any unused odd device code equal to 41 or above
4. Can be set to any unused device code between 1 and 76
5. Microinterrupts are not maskable.

Index by Instruction Name

Add Complement 158
Add 158
Extended Add Immediate 159
Add Immediate 159
AND With Complemented Source 160
AND 160
AND Immediate 160
Block Add and Move 160
Breakpoint 161
Block Move 162
Set Bit To One 162
Set Bit To Zero 163
Compare To Limits 163
Character Compare 164
Character Move Until True 165
Character Move 166
Count Bits 167
Complement 167
Complement Carry 168
Set Carry to One 168
Set Carry to Zero 168
Character Translate 168
Convert to 16-Bit Integer 170
Decimal Add 170
Double Hex Shift Left 170
Double Hex Shift Right 171
Unsigned Divide 171
Signed Divide 172
Sign Extend and Divide 172
Double Logical Shift 172
Decimal Subtract 173
Dispatch 173
Decrement And Skip If Zero 174
Decrement the Word Addressed by WSP and Skip if Zero 175
Load CPU Identification 175
Edit 176
Add To DI 177
Add To P Depending On S 177
Add To P Depending On T 178
Add To P 178
Add To SI 178

Decrement and Jump If Non-Zero 178
End Edit 179
Insert Characters Immediate 179
Insert Character J Times 179
Insert Character Once 179
Insert Sign 179
Insert Character Suppress 180
Move Alphabetics 180
Move Characters 180
Move Float 180
Move Numerics 181
Move Digit With Overpunch 181
Move Numeric With Zero Suppression 182
End Float 182
Set S To One 182
Set S To Zero 183
Store In Stack 183
Set T To One 183
Set T To Zero 183
Extended Decrement and Skip if Zero 183
Extended Increment And Skip If Zero 184
Extended Jump 184
Extended Jump To Subroutine 184
Extended Load Accumulator 185
Extended Load Byte 185
Load Effective Address 186
Enqueue Towards the Head 186
Enqueue Towards the Tail 187
Extended Store Accumulator 188
Extended Store Byte 188
Absolute Value 189
Add Double (FPAC to FPAC) 189
Add Double (Memory to FPAC) 189
Add Single (Memory to FPAC) 190
Add Single (FPAC to FPAC) 190
Clear Errors 190
Compare Floating Point 191
Divide Double (FPAC by FPAC) 191
Divide Double (FPAC by Memory) 192
Divide Single (FPAC by Memory) 192
Divide Single (FPAC by FPAC) 193
Load Exponent 193
Fix To AC 194
Fix To Memory 194
Halve 195
Integerize 195
Float From AC 195
Load Floating Point Double 196
Load Floating Point Single 196
Float From Memory 196
Load Floating Point Status 197
Multiply Double (FPAC by FPAC) 197
Multiply Double (FPAC by Memory) 198
Multiply Single (FPAC by Memory) 198
Move Floating Point 199
Multiply Single (FPAC by FPAC) 199
Negate 199

Normalize 199
No Skip 200
Pop Floating Point State 200
Push Floating Point State 202
Read High Word 203
Skip Always 203
Scale 203
Subtract Double (FPAC from FPAC) 204
Skip On Zero 204
Skip On Greater Than Or Equal To Zero 204
Skip On Greater Than Zero 204
Skip On Less Than Or Equal To Zero 204
Skip On Less Than Zero 205
Subtract Double (Memory from FPAC) 205
Subtract Single (Memory from FPAC) 205
Skip On No Zero Divide 206
Skip On Non-Zero 206
Skip On No Error 206
Skip On No Mantissa Overflow 206
Skip On No Overflow 206
Skip On No Overflow and No Zero Divide 207
Skip On No Underflow 207
Skip On No Underflow And No Zero Divide 207
Skip On No Underflow And No Overflow 207
Subtract Single (FPAC from FPAC) 207
Store Floating Point Status 208
Store Floating Point Double 208
Store Floating Point Single 208
Trap Disable 209
Trap Enable 209
Fixed Point Trap Disable 209
Fixed Point Trap Enable 210
Halve 210
Hex Shift Left 210
Hex Shift Right 210
Increment 211
Inclusive OR 211
Inclusive OR Immediate 212
Increment And Skip If Zero 212
Increment the Word Addressed by WSP and Skip if Zero 212
Jump 212
Jump To Subroutine 213
Call Subroutine (Long Displacement) 213
Load CPU Identification 214
Load Accumulator 214
Load Accumulator with WFP 215
Load Accumulator with WSB 215
Load Accumulator with WSL 215
Load Accumulator with WSP 215
Load Accumulator with Double Word 216
Load Byte 216
Load Integer 216
Load Integer Extended 217
Dispatch (Long Displacement) 217
Load Effective Address 218
Add Double (Memory to FPAC) (Long Displacement) 219
Add Single (Memory to FPAC) (Long Displacement) 219

Divide Double (FPAC by Memory) (Long Displacement) **220**
Divide Single (FPAC by Memory) (Long Displacement) **220**
Load Floating Point Double (Long Displacement) **220**
Floating Point Load Single (Long Displacement) **221**
Load Floating Point Status (Long Displacement) **221**
Multiply Double (FPAC by Memory) (Long Displacement) **221**
Multiply Single (FPAC by Memory) (Long Displacement) **222**
Subtract Double (Memory from FPAC) (Long Displacement) **222**
Subtract Single (Memory from FPAC) (Long Displacement) **223**
Store Floating Point Status (Long Displacement) **223**
Store Floating Point Double (Long Displacement) **223**
Store Floating Point Single (Long Displacement) **224**
Jump (Long Displacement) **224**
Jump to Subroutine (Long Displacement) **224**
Load Byte (Long Displacement) **225**
Load Effective Address (Long Displacement) **225**
Load Effective Byte Address (Long Displacement) **225**
Load Modified and Referenced Bits **225**
Narrow Add Memory Word to Accumulator (Long Displacement) **226**
Narrow Divide Memory Word (Long Displacement) **226**
Narrow Decrement and Skip if Zero (Long Displacement) **226**
Narrow Increment and Skip if Zero (Long Displacement) **227**
Narrow Load Accumulator (Long Displacement) **227**
Narrow Multiply Memory Word (Long Displacement) **227**
Narrow Store Accumulator (Long Displacement) **227**
Narrow Subtract Memory Word (Long Displacement) **228**
Locate Lead Bit **228**
Push Address (Long Displacement) **228**
Push Byte Address (Long Displacement) **228**
Load Physical **229**
Push Jump (Long Displacement) **229**
Load Processor Status Register into AC0 **230**
Locate and Reset Lead Bit **230**
Load All Segment Base Registers **230**
Load Segment Base Registers 1-7 **231**
Logicai Shift **232**
Load Sign **232**
Store Byte (Long Displacement) **233**
Wide Add Memory Word to Accumulator (Long Displacement) **233**
Wide Divide From Memory (Long Displacement) **233**
Wide Decrement and Skip if Zero (Long Displacement) **234**
Wide Increment and Skip if Zero (Long Displacement) **234**
Wide Load Accumulator (Long Displacement) **234**
Wide Multiply From Memory (Long Displacement) **234**
Wide Store Accumulator (Long Displacement) **235**
Wide Subtract Memory Word (Long Displacement) **235**
Move **235**
Modify Stack Pointer **236**
Unsigned Multiply **236**
Signed Multiply **236**
Narrow Add **237**
Narrow Extended Add Immediate **237**
Narrow Add Immediate **237**
Search Queue **238**
Narrow Search Queue Backward **239**
Narrow Search Queue Backward **239**
Narrow Search Queue Backward **240**

Narrow Search Queue Backward **240**
Narrow Search Queue Backward **240**
Narrow Search Queue Backward **240**
Narrow Search Queue Backward **240**
Narrow Search Queue Backward **241**
Narrow Divide **241**
Negate **241**
Narrow Search Queue Forward **241**
Narrow Search Queue Forward **242**
Narrow Search Queue Forward **242**
Narrow Search Queue Forward **242**
Narrow Search Queue Forward **242**
Narrow Search Queue Forward **242**
Narrow Search Queue Forward **243**
Narrow Search Queue Forward **243**
Narrow Load Immediate **243**
Narrow Multiply **243**
Narrow Negate **243**
Narrow Skip on All Bits Set in Accumulator **244**
Narrow Skip on All Bits Set in Memory Location **244**
Narrow Skip on Any Bit Set in Accumulator **244**
Narrow Skip on Any Bit Set in Memory Location **245**
Narrow Subtract Immediate **245**
Narrow Subtract **245**
OR Referenced Bits **245**
Purge the ATU **246**
Pop Block and Execute **246**
Pop Multiple Accumulators **247**
Pop Block **247**
Pop PC And Jump **248**
Push Multiple Accumulators **248**
Push Jump **248**
Push Return Address **249**
Reset Referenced Bit **249**
Restore **249**
Return **250**
Save **251**
Subtract Immediate **252**
Sign Extend **252**
Skip If ACS Greater Than Or Equal to ACD **252**
Skip If ACS Greater Than ACD **253**
Store Modified and Referenced Bits **253**
Skip On Non-Zero Bit **253**
Skip on OVR Reset **254**
Store Processor Status Register From AC0 **254**
Store Accumulator **254**
Store Accumulator in WFP **255**
Store Accumulator in WSB **255**
Store Accumulator in WSL **255**
Store Accumulator in WSP **255**
Store Accumulator into Stack Pointer Contents **256**
Store Byte **256**
Store Integer **256**
Store Integer Extended **257**
Subtract **258**
System Call **258**
Skip On Zero Bit **259**

Skip On Zero Bit And Set To One **259**
Skip on Valid Byte Pointer **260**
Skip on Valid Word Pointer **260**
Wide Add Complement **260**
Wide Add **261**
Wide Add With Wide Immediate **261**
Wide Add Immediate **261**
Wide AND with Complemented Source **261**
Wide AND **262**
Wide AND Immediate **262**
Wide Arithmetic Shift **262**
Wide Block Move **263**
Load PC **264**
Wide Search Queue Backward **264**
Wide Search Queue Backward **264**
Wide Search Queue Backward **264**
Wide Search Queue Backward **264**
Wide Search Queue Backward **265**
Wide Search Queue Backward **265**
Wide Search Queue Backward **265**
Wide Search Queue Backward **265**
Wide Set Bit to One **265**
Wide Set Bit to Zero **266**
Wide Compare to Limits **266**
Wide Character Compare **267**
Wide Character Move Until True **268**
Wide Character Move **269**
Wide Count Bits **270**
Wide Complement **270**
Wide Character Translate **270**
Wide Divide **272**
Wide Signed Divide **272**
Pop Context Block **272**
Wide Edit **273**
Add To DI **274**
Add To P Depending On S **274**
Add To P Depending On T **275**
Add To P **275**
Add To SI **275**
Decrement And Jump If Non-Zero **275**
End Edit **276**
Insert Characters Immediate **276**
Insert Character J Times **276**
Insert Character Once **276**
Insert Sign **276**
Insert Character Suppress **277**
Move Alphabetics **277**
Move Characters **277**
Move Float **277**
Move Numerics **278**
Move Digit With Overpunch **278**
Move Numeric With Zero Suppression **279**
End Float **279**
Set S To One **279**
Set S To Zero **279**
Store In Stack **280**
Set T To One **280**

Set T To Zero 280
Wide Fix from Floating Point Accumulator 280
Wide Float from Fixed Point Accumulator 281
Wide Floating Point Pop 281
Wide Floating Point Push 282
Wide Search Queue Forward 284
Wide Search Queue Forward 284
Wide Search Queue Forward 284
Wide Search Queue Forward 284
Wide Search Queue Forward 284
Wide Search Queue Forward 285
Wide Search Queue Forward 285
Wide Search Queue Forward 285
Wide Halve 285
Wide Increment 285
Wide Inclusive OR 286
Wide Inclusive OR Immediate 286
Wide Load with Wide Immediate 286
Wide Load Byte 286
Wide Load Integer 287
Wide Load Integer Extended 287
Wide Locate Lead Bit 288
Wide Locate and Reset Lead Bit 288
Wide Logical Shift 288
Wide Logical Shift Immediate 289
Wide Load Sign 289
Wide Move 289
Wide Modify Stack Pointer 290
Wide Multiply 290
Wide Signed Multiply 290
Wide Add with Narrow Immediate 291
Wide Negate 291
Wide Pop Accumulators 291
Wide Pop Block 292
Pop PC and Jump 293
Push Accumulators 293
Wide Restore 293
Wide Return 294
Wide Skip on All Bits Set in Accumulator 294
Wide Skip on All Bits Set in Double-word Memory Location 295
Wide Skip on Any Bit Set in Accumulator 295
Wide Skip on Any Bit Set in Double-word Memory Location 295
Wide Save/Reset Overflow Mask 296
Wide Save/Set Overflow Mask 296
Wide Subtract Immediate 297
Wide Skip If Equal To 297
Wide Signed Skip If Greater Than Or Equal To 298
Wide Signed Skip If Greater Than 298
Wide Skip on Bit Set to One 298
Wide Skip on Bit Set to Zero 299
Wide Signed Skip If Less Than Or Equal To 299
Wide Signed Skip If Less Than 299
Wide Skip on Nonzero Bit 300
Wide Skip If Not Equal To 300
Wide Special Save/Set Overflow Mask 300
Wide Special Save/Set Overflow Mask 301
Wide Store Byte 302

Wide Store Integer 302
Wide Store Integer Extended 303
Wide Subtract 304
Wide Skip on Zero Bit 304
Wide Skip on Zero Bit and Set Bit To One 304
Wide Unsigned Skip If Greater Than Or Equal To 305
Wide Unsigned Skip If Greater Than 305
Wide Exchange 305
Wide Extended Operation 306
Wide Alternate Extended Operation 306
Wide Exclusive OR 307
Wide Exclusive OR Immediate 307
Call Subroutine (Extended Displacement) 307
Exchange Accumulators 308
Execute 308
Add Double (Memory to FPAC) (Extended Displacement 309
Add Single (Memory to FPAC) (Extended Displacement) 309
Divide Double (FPAC by Memory) (Extended Displacement) 310
Divide Single (FPAC by Memory) (Extended Displacement) 310
Extended Load Floating Point Double 310
Extended Load Floating Point Single 311
Multiply Double (FPAC by Memory) (Extended Displacement) 311
Multiply Single (FPAC by Memory) (Extended Displacement) 311
Subtract Double (Memory from FPAC) (Extended Displacement) 312
Subtract Single (Memory from FPAC) (Extended Displacement) 312
Store Floating Point Double (Extended Displacement) 312
Store Floating Point Single (Extended Displacement) 313
Jump (Extended Displacement) 313
Jump to Subroutine (Extended Displacement) 313
Load Effective Address (Extended Displacement) 313
Load Effective Byte Address (Extended Displacement) 314
Narrow Add Accumulator to Memory Word (Extended Displacement) 314
Narrow Divide Memory Word (Extended Displacement) 314
Narrow Decrement and Skip if Zero (Extended Displacement) 315
Narrow Increment and Skip if Zero (Extended Displacement) 315
Narrow Load Accumulator (Extended Displacement) 315
Narrow Multiply Memory Word (Extended Displacement) 315
Narrow Store Accumulator (Extended Displacement) 316
Narrow Subtract Memory Word (Extended Displacement) 316
Extended Operation 316
Exclusive OR 317
Exclusive OR Immediate 317
Push Address (Extended Displacement) 318
Push Byte Address (Extended Displacement) 318
Push Jump (Extended Displacement) 318
Vector on Interrupting Device (Extended Displacement) 318
Wide Add Accumulator to Memory Word (Extended Displacement) 319
Wide Divide Memory Word (Extended Displacement) 319
Wide Decrement and Skip if Zero (Extended Displacement) 319
Wide Increment and Skip if Zero (Extended Displacement) 320
Wide Load Accumulator (Extended Displacement) 320
Wide Multiply Memory Word (Extended Displacement) 320
Wide Store Accumulator (Extended Displacement) 320
Wide Subtract Memory Word (Extended Displacement) 321
Zero Extend 321

Index by Mnemonics

ADC*[c][sh][#]* **158**
ADD*[c][sh][#]* **158**
ADDI **159**
ADI **159**
ANC **160**
AND*[c][sh][#]* **160**
ANDI **160**
BAM **160**
BKPT **161**
BLM **162**
BTO **162**
BTZ **163**
CLM **163**
CMP **164**
CMT **165**
CMV **166**
COB **167**
COM*[c][sh][#]* **167**
CRYTC **168**
CRYTO **168**
CRYTZ **168**
CTR **168**
CVWN **170**
DAD **170**
DHXL **170**
DHXR **171**
DIV **171**
DIVS **172**
DIVX **172**
DLSH **172**
DSB **173**
DSPA **173**
DSZ **174**
DSZTS **175**
ECLID **175**
EDIT **176**
DADI **177**
DAPS **177**
DAPT **178**
DAPU **178**
DASI **178**
DDTK **178**
DEND **179**
DICI **179**
DIMC **179**
DINC **179**
DINS **179**
DINT **180**
DMVA **180**
DMVC **180**
DMVF **180**
DMVN **181**
DMVO **181**
DMVS **182**
DNDF **182**
DSSO **182**
DSSZ **183**
DSTK **183**
DSTO **183**
DSTZ **183**
EDSZ **183**
EISZ **184**
EJMP **184**
EJSR **184**
ELDA **185**
ELDB **185**
ELEF **186**
ENQH **186**
ENQT **187**
ESTA **188**
ESTB **188**
FAB **189**
FAD **189**
FAMD **189**
FAMS **190**
FAS **190**
FCLE **190**
FCMP **191**
FDD **191**
FDMD **192**
FDMS **192**
FDS **193**
FEXP **193**
FFAS **194**
FFMD **194**
FHLV **195**
FINT **195**
FLAS **195**
FLDD **196**
FLDS **196**
FLMD **196**
FLST **197**
FMD **197**
FMMD **198**
FMMS **198**
FMOV **199**
FMS **199**
FNEG **199**
FNOM **199**
FNS **200**
FPOP **200**
FPSH **202**
FRH **203**
FSA **203**
FSCAL **203**
FSD **204**
FSEQ **204**
FSGE **204**
FSGT **204**
FSLE **204**
FSLT **205**
FSMD **205**
FSMS **205**
FSND **206**
FSNE **206**
FSNER **206**
FSNM **206**
FSNO **206**
FSNOD **207**
FSNU **207**
FSNUD **207**
FSNUO **207**
FSS **207**
FSST **208**
FSTD **208**
FSTS **208**
FTD **209**
FTE **209**
FXTD **209**
FXTE **210**
HLV **210**
HXL **210**
HXR **210**
INC*[c][sh][#]* **211**
IOR **211**

IORI 212
ISZ 212
ISZTS 212
JMP 212
JSR 213
LCALL 213
LCPID 214
LDA 214
LDAFP 215
LDASB 215
LDASL 215
LDASP 215
LDATS 216
LDB 216
LDI 216
LDIX 217
LDSP 217
LEF 218
LFAMD 219
LFAMS 219
LFDMD 220
LFDMS 220
LFLDD 220
LFLDS 221
LFLST 221
LFMMD 221
LFMMS 222
LFSMD 222
LFSMS 223
LFSST 223
LFSTD 223
LFSTS 224
LJMP 224
LJSR 224
LLDB 225
LLEF 225
LLEFB 225
LMRF 225
LNADD 226
LNDIV 226
LNDSZ 226
LNISZ 227
LNLDA 227
LNMUL 227
LNSTA 227
LNSUB 228
LOB 228
LPEF 228
LPEFB 228
LPHY 229
LPSHJ 229
LPSR 230
LRB 230
LSBRA 230
LSBRS 231
LSH 232
LSN 232
LSTB 233
LWADD 233
LWDIV 233
LWDSZ 234
LWISZ 234
LWLDA 234
LWMUL 234
LWSTA 235
LWSUB 235
MOV*[c][sh][#]* 235
MSP 236
MUL 236
MULS 236
NADD 237
NADDI 237
NADI 237
NBSAC 239
NBSAS 239
NBSE 240
NBSGE 240
NBSLE 240
NBSNE 240
NBSSC 240
NBSSS 241
NDIV 241
NEG*[c][sh][#]* 241
NFSAC 241
NFSAS 242
NFSE 242
NFSGE 242
NFSLE 242
NFSNE 242
NFSSC 243
NFSSS 243
NLDAI 243
NMUL 243
NNEG 243
NSALA 244
NSALM 244
NSANA 244
NSANM 245
NSBI 245
NSUB 245
ORFB 245
PATU 246
PBX 246
POP 247
POPB 247
POPJ 248
PSH 248
PSHJ 248
PSHR 249
RRFB 249
RSTR 249
RTN 250
SAVE 251
SBI 252
SEX 252
SGE 252
SGT 253
SMRF 253
SNB 253
SNOVR 254
SPSR 254
STA 254
STAFP 255
STASB 255
STASL 255
STASP 255
STATS 256
STB 256
STI 256
STIX 257
SUB*[c][sh][#]* 258
SYC 258
SZB 259
SZBO 259
VBP 260
VWP 260
WADC 260
WADD 261
WADDI 261
WADI 261
WANC 261
WAND 262
WANDI 262
WASH 262
WBLM 263
WBR 264
WBSAC 264
WBSAS 264
WBSE 264
WBSGE 264
WBSLE 265
WBSNE 265
WBSSC 265
WBSSS 265
WBTO 265
WBTZ 266
WCLM 266
WCMP 267
WCMT 268
WCMV 269
WCOB 270
WCOM 270
WCTR 270
WDIV 272
WDIVS 272
WDPOP 272
WEDIT 273
DADI 274
DAPS 274
DAPT 275
DAPU 275
DASI 275
DDTK 275
DEND 276
DICI 276
DIMC 276
DINC 276

DINS 276
DINT 277
DMVA 277
DMVC 277
DMVF 277
DMVN 278
DMVO 278
DMVS 279
DNDF 279
DSSO 279
DSSZ 279
DSTK 280
DSTO 280
DSTZ 280
WFFAD 280
WFLAD 281
WFPOP 281
WFPSH 282
WFSAC 284
WFSAS 284
WFSE 284
WFSGE 284
WFSLE 284
WFSNE 285
WFSSC 285
WFSSS 285
WHLV 285
WINC 285
WIOR 286
WIORI 286
WLDAI 286
WLDB 286
WLDI 287
WLDIX 287
WLOB 288
WLRB 288
WLSH 288
WLSI 289
WLSN 289
WMOV 289
WMSP 290
WMUL 290
WMULS 290
WNADI 291
WNEG 291
WPOP 291
WPOPB 292
WPOPJ 293
WPSH 293
WRSTR 293
WRTN 294
WSALA 294
WSALM 295
WSANA 295
WSANM 295
WSAVR 296
WSAVS 296
WSBI 297
WSEQ 297
WSGE 298
WSGT 298
WSKBO 298
WSKBZ 299
WSLE 299
WSLT 299
WSNB 300
WSNE 300
WSSVR 300
WSSVS 301
WSTB 302
WSTI 302
WSTIX 303
WSUB 304
WSZB 304
WSZBO 304
WUSGE 305
WUSGT 305
WXCH 305
WXOP 306
WXOP1 306
WXOR 307
WXORI 307
XCALL 307
XCH 308
XCT 308
XFAMD 309
XFAMS 309
XFDMD 310
XFDMS 310
XFLDD 310
XFLDS 311
XFMMD 311
XFMMS 311
XFSMD 312
XFSMS 312
XFSTD 312
XFSTS 313
XJMP 313
XJSR 313
XLEF 313
XLEFB 314
XNADD 314
XNDIV 314
XNDSZ 315
XNISZ 315
XNLDA 315
XNMUL 315
XNSTA 316
XNSUB 316
XOP0 316
XOR 317
XORI 317
XPEF 318
XPEFB 318
XPSHJ 318
XVCT 318
XWADD 319
XWDIV 319
XWDSZ 319
XWISZ 320
XWLDA 320
XWMUL 320
XWSTA 320
XWSUB 321
ZEX 321

Engineering Publications Comment Form

Please help us improve our future publications by answering the questions below. Use the space provided for your comments.

Title: ____________________

Document No. 014-000648-00

Yes	No	Question	Comments
☐	☐	Is this manual easy to read?	○ You (can,cannot) find things easily. ○ Language (is,is not) appropriate. ○ Technical terms (are,are not) defined as needed. ○ Other:
		In what ways do you find this manual useful?	○ Learning to use the equipment ○ As a reference ○ As an introduction to the product ○ To instruct a class. ○ Other:
☐	☐	Do the illustrations help you?	○ Visuals (are,are not) well designed. ○ Labels and captions (are,are not) clear. ○ Other:
☐	☐	Does the manual tell you all you need to know? What additional information would you like?	
☐	☐	Is the information accurate? (If not please specify with page number and paragraph.)	

Name: __________ Title: __________

Company: __________ Division: __________

Address: __________ City: __________

State: _____ Zip: _____ Telephone: _____ Date: _____

DG-06895

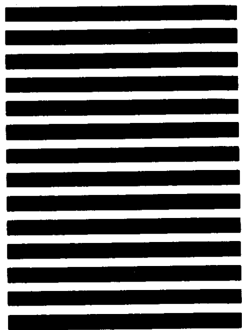
DataGeneral

Data General Corporation, Westboro, Massachusetts 01581

FOLD FOLD

STAPLE STAPLE

FOLD FOLD



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

Postage will be paid by addressee:

ATTN: Users Group Coordinator
4400 Computer Drive
Westboro, MA 01581