Data General

Customer Documentation

ECLIPSE® MV/Family (32-Bit) Systems Instruction Dictionary

ECLIPSE® MV/Family (32-Bit) Systems Principles of Operation

014-001371-01

Ordering No. 014-001371

Printed in the United States of America
Rev. 01, July 1988

Notice

ECLIPSE MV/Family (32-Bit) Systems Principles of Operation
014-001371-01

Revision History:

Original Release - January 1988
First Revision - July 1988

A vertical bar in the margin of a page indicates substantive technical change from the previous revision.

Preface

The *ECLIPSE® MV/Family (32-Bit) Systems Principles of Operation* manual explains the processor-independent concepts and functions of ECLIPSE MV/Family systems to assembly language programmers. The second volume in this two-volume global set, *ECLIPSE® MV/Family (32-Bit) Systems Instruction Dictionary* (DGC No. 014-001372), describes each instruction in the ECLIPSE MV/Family instruction set. Processor-dependent information, available in machine-specific supplements, complements the two-volume global set.

A related manual, the *ECLIPSE® MV/Family Instruction Reference Booklet* (DGC No. 014-000702), provides a brief summary of the instruction set and register information. The reference booklet lists each instruction by assembler-recognizable mnemonic with a shorthand description of its function.

The Assembler mentioned in this manual is Data General's Macroassembler which is detailed in the *AOS/VS Macroassembler (MASM) Reference Manual* (DGC No. 093-000242).

Manual Organization

This manual contains 10 chapters. Chapter 1 gives an overview of the 32-bit ECLIPSE MV/Family system of computers. Chapter 1 of each machine-specific supplement provides a hardware summary for that computer system.

Chapters 2 through 10 present, in a functional framework, processor independent concepts and functions, and introduce the instruction set. Machine-specific supplements contain information specific to that computer system. The chapters explain:

- Fixed-point computation
- Floating-point computation
- Stack management
- Program flow management
- Queue management
- Graphics management
- Device management
- Memory and system management
- ECLIPSE 16-bit compatible instructions

Appendixes A through D in this global manual present information on:

- Register fields
- Fault and status codes
- Reserved memory locations
- Load Control Store instruction

Machine-specific supplements include Appendixes E through G which provide details on:

- Standard I/O device codes
- Context block formats
- Instruction execution times

A Glossary offers brief definitions of terms used to describe the features of ECLIPSE MV/Family computer systems.

The second volume of this set, *ECLIPSE® MV/Family (32-Bit) Systems Instruction Dictionary*, presents each of the instructions in the ECLIPSE MV/Family instruction set alphabetically according to instruction mnemonic.

Standard Symbols

The manual uses certain conventions and abbreviations.

[ ] The square brackets indicate an optional argument. Omit the square brackets when you include an optional argument with an Assembler statement.

UPPERCASE BOLDFACE characters indicate a literal and/or argument in an Assembler statement. When you include a literal argument with an Assembler statement, use the exact form.

lowercase italic characters indicate a variable argument in an Assembler statement. When you include the variable with an Assembler statement, substitute a literal value for the variable argument.

ac The ac abbreviation indicates a fixed-point accumulator.

acs The acs abbreviation indicates a *source* fixed-point accumulator.

acd The acd abbreviation indicates a *destination* fixed-point accumulator.

fpac The fpac abbreviation indicates a floating-point accumulator.

fpacs The fpacs abbreviation indicates a *source* floating-point accumulator.

fpacd The fpacd abbreviation indicates a *destination* floating-point accumulator.

Coordinating Machine-Specific Supplements

The two-volume global set, *ECLIPSE® MV/Family (32-Bit) Systems Principles of Operation* and *ECLIPSE® MV/Family (32-Bit) Systems Instruction Dictionary*, supersedes all previous revisions of the single-volume manual, *ECLIPSE® MV/Family 32-Bit Systems Principles of Operation* (DGC No. 014-000704). The two-volume set contains the most up-to-date information for the ECLIPSE MV/Family computer systems.

The machine-specific supplements are designed to be incorporated into either the two-volume set or the original single-volume manual to create a machine-specific reference for assembly language programmers.

Table P-1 lists the various revisions of the supplement manuals which should be incorporated with the two-volume set (014-001371 and 014-001372). Table P-2 lists the revisions of the supplement manuals which should be incorporated with the original single-volume manual (014-000704). Note that the manual revision numbers may be found on the manual's Notice page. (If your particular machine's supplement or "functional characteristics" manual is not listed, then it is unaffected.)

Table P-1 *Two-Volume Set (014-001371 and 014-001372)*

Manual	Ordering Number	Revision Numbers
ECLIPSE MV/2000™ DC and DS/7500 Series Systems Principles of Operation Supplement	014-001203	03 and up
ECLIPSE MV/7800™ Series Systems Principles of Operation Supplement	014-001180	03 and up
ECLIPSE MV/15000™ Series Systems Principles of Operation Supplement	014-001297	02 and up
ECLIPSE MV/20000™ Series Systems Principles of Operation Supplement	014-001169	02 and up

Table P-2 *Single-Volume (014-000704)*

Manual	Ordering Number	Revision Numbers
ECLIPSE MV/2000™ DC and DS/7500 Series Systems Principles of Operation Supplement	014-001203	00 through 02
ECLIPSE MV/7800™ Series Systems Principles of Operation Supplement	014-001180	00 through 02
ECLIPSE MV/8000™ II System Principles of Operation Supplement	014-001227	00
ECLIPSE MV/10000™ Class Systems Principles of Operation Supplement	014-001228	00
ECLIPSE MV/15000™ Series Systems Principles of Operation Supplement	014-001297	00 through 01
ECLIPSE MV/20000™ Series Systems Principles of Operation Supplement	014-001169	00 through 01

End of Preface

Contents

1 System Overview

2 Fixed-Point Computing

3 Floating-Point Computing

4 Stack Management

5 Program Flow Management

6 Queue Management

7 Graphics Management

8 Device Management

A Register Fields

B Fault and Status Codes

C Reserved Memory Locations

D Load Control Store Instruction

Glossary

Index

Figures

Tables

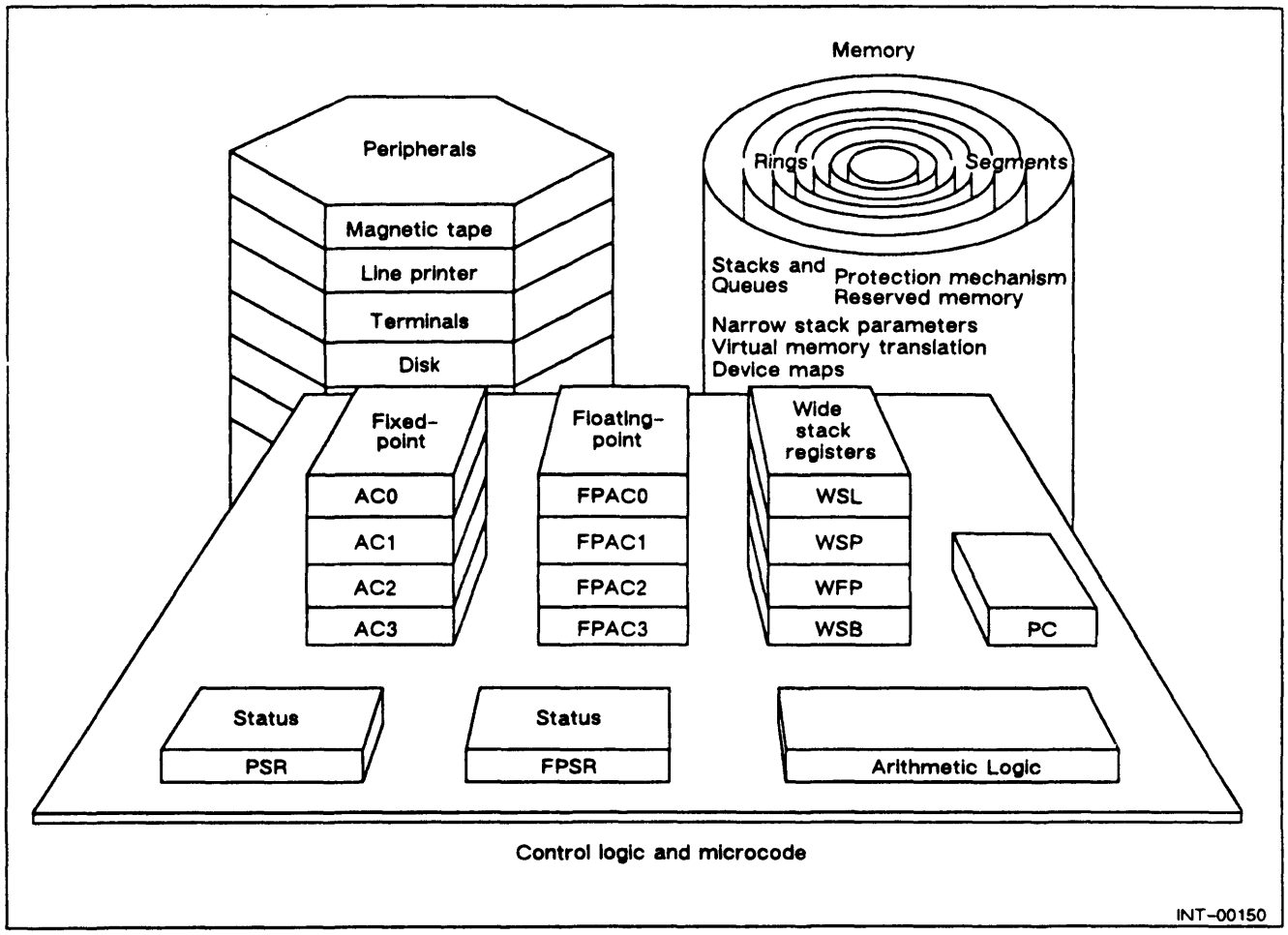


Figure 1-1 *ECLIPSE MV/Family functional components*

1

System Overview

The ECLIPSE 32-bit central processor — hereafter called the processor — provides facilities to manage data, to access memory, and to control program flow. (See Figure 1-1.)

The processor can perform fixed-point or floating-point computation, as well as stack, program, queue, device, system, and memory management. In addition, the processor contains ECLIPSE compatible instructions for 16-bit program development and upward program compatibility. (This manual contains pertinent information for programmers doing cross-development for ECLIPSE 16-bit systems.)

This chapter provides a brief description of the processor's functional capabilities, memory address space, and system protection capabilities. Machine-specific supplements provide hardware summaries.

NOTE: *Each machine-specific supplement contains an appendix,* "Instruction Execution Times," *which lists all instructions supported by that particular machine. If an instruction is not listed in this appendix, it is not supported on that processor.*

Functional Capabilities

The following sections of this chapter describe the functional capabilities of ECLIPSE MV/Family computers.

NOTE: *Data General defines a computer word as 16 bits and a doubleword as 32 bits. For consistency throughout this manual, we specify bit patterns for registers and instructions as 16-bit quantities, regardless of the actual bit length of these quantities. For instance, the bit pattern for a 64-bit floating-point accumulator is represented as four lines of 16 bits each.*

Registers

All ECLIPSE MV/Family computers implement the following registers:

- Four 32-bit fixed-point accumulators
- One 16-bit processor status register
- Four 64-bit floating-point accumulators
- One 64-bit floating-point status register
- Four 32-bit stack management registers
- One 31-bit program counter
- Eight 32-bit segment base registers
- One 1-bit Carry register

Any register bits that are listed as *reserved* must be loaded, or written to, with zeros. The processor may or may not verify that these bits are zero, however, correct operation is only defined when these bits are zero.

Fixed-Point Computation

Fixed-point computation uses fixed-point binary arithmetic with signed and unsigned 16-bit and 32-bit numbers. The processor also performs decimal arithmetic and logical operations, and manipulates 8-bit bytes.

The processor contains five registers relating to fixed-point computation: four 32-bit fixed-point accumulators (AC0, AC1, AC2, and AC3); and a processor status register (PSR). The next two sections summarize these fixed-point registers. Refer to the chapter, "Fixed-Point Computing," for additional information.

NOTE: *The lowest numbered bit of a register (such as bit 0) is the most significant bit. The highest numbered bit (such as bit 31) is the least significant bit.*

Fixed-Point Accumulators

Fixed-point accumulators are accessed with instructions that manipulate a bit, byte, word (16 bits), or doubleword (32 bits). Figure 1-2 indicates the mapping of bytes and words within the fixed-point accumulators.

The majority of operands smaller than the accumulator are right-justified within the accumulator.

In addition to using an accumulator for fixed-point computation:

- The processor returns state information in accumulators under certain conditions, such as an error code after a fault occurs;
- An instruction may be loaded or built in an accumulator, and then executed;
- AC2 or AC3 may be used as index registers for addressing (refer to the section, "Address Modes").

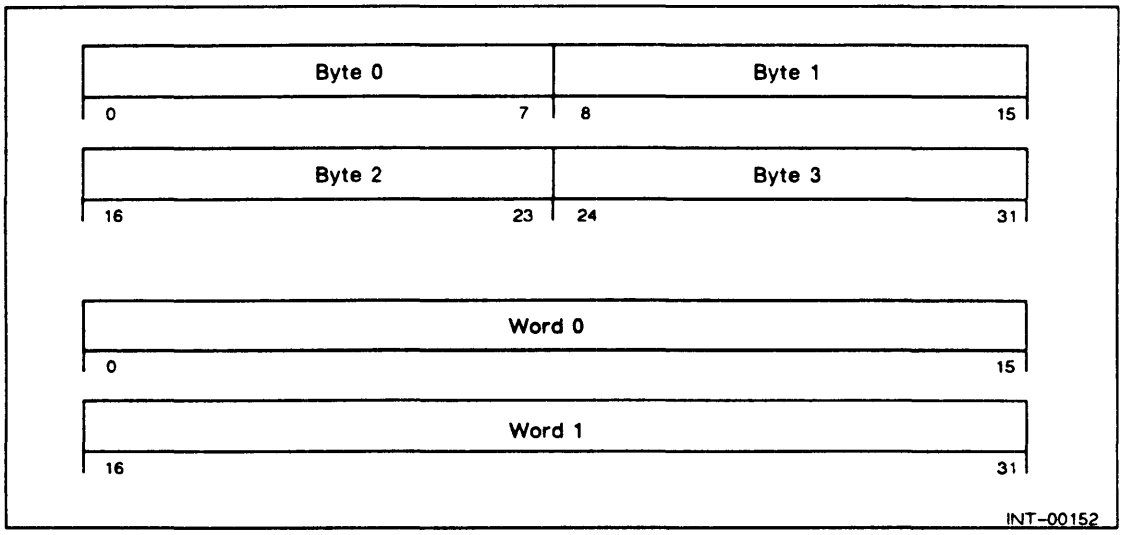


Figure 1-2 *Fixed-point accumulator*

Processor Status Register

The processor status register (PSR) contains status flags such as an overflow fault service mask and a fixed-point overflow fault flag. The overflow fault service mask allows the processor to service a fault. The processor sets the overflow fault flag when the results of a fixed-point computation exceed the system's ability to represent the result of the compute. The remaining flags are processor-dependent.

You can access the PSR bits with instructions that set a bit or that test and skip on condition of a bit. Refer to the chapter, "Fixed-Point Computing," for additional information.

Floating-Point Computation

Floating-point computation consists of floating-point binary arithmetic with signed, single-precision (32-bit) and double-precision (64-bit) numbers.

The processor contains five registers relating to floating-point computation: four 64-bit floating-point accumulators (FPAC0, FPAC1, FPAC2, and FPAC3); and a floating-point status register (FPSR). The next two sections summarize the floating-point registers. Refer to the chapter, "Floating-Point Computing," for additional information.

Floating-Point Accumulators

A floating-point accumulator is accessed with instructions that manipulate single- and double-precision floating-point numbers.

A single-precision number requires a doubleword (two consecutive words), while a double-precision number requires two doublewords (four consecutive words) Figure 1-3 shows how these values map to a floating-point accumulator.

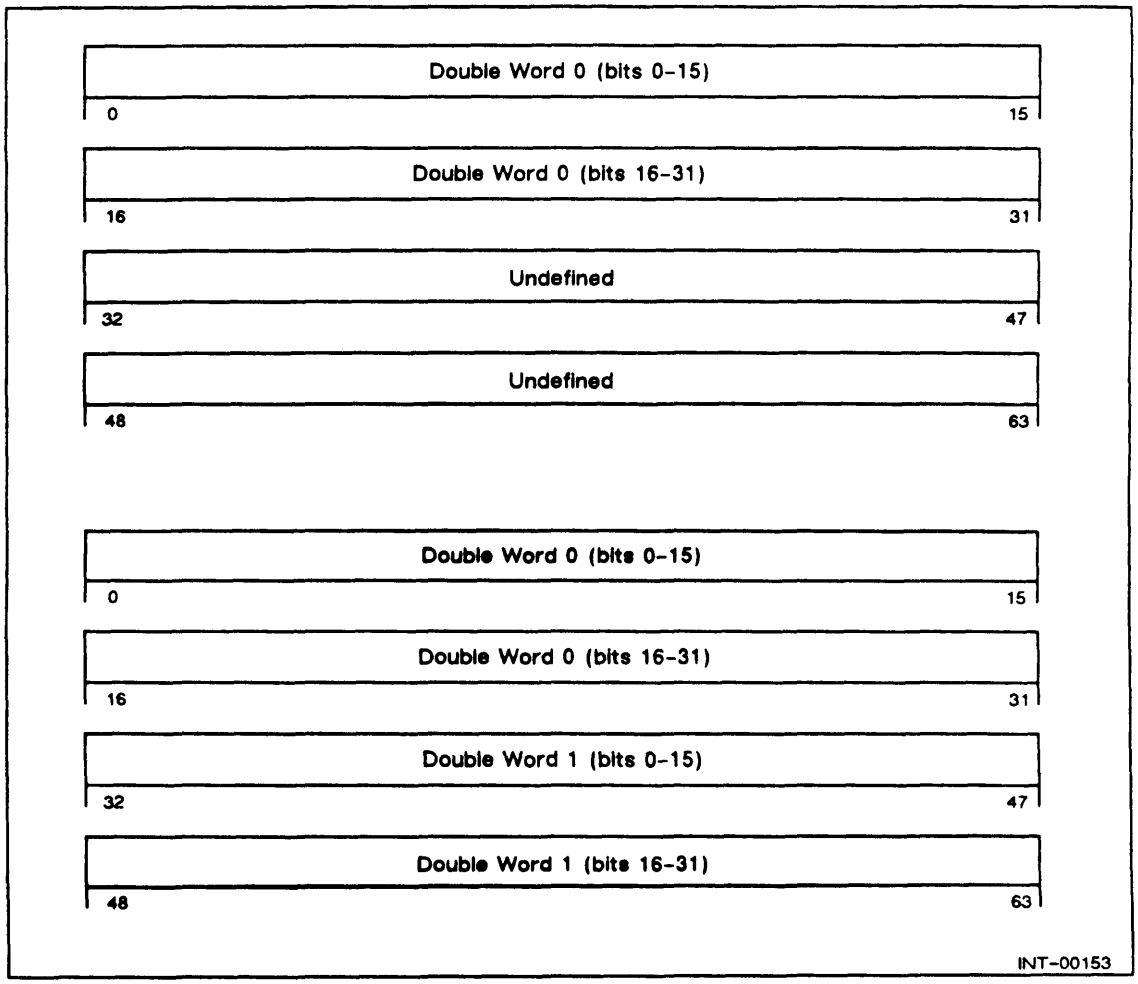


Figure 1-3 *Floating-point accumulator*

Floating-Point Status Register

The floating-point status register (FPSR) contains status and fault flags, such as exponent overflow and underflow fault flags, fault service mask, input argument error flag, rounding flag, and processor status flags.

The processor sets an overflow or underflow fault flag when the result of a floating-point computation exceeds the processor's storage capacity. The fault service mask allows the processor to service a fault. The remaining flags provide information on processor status.

You can access the contents of the FPSR with instructions to initialize it or to test and skip on a condition.

Stack Management

The processor has facilities for narrow and wide stack management. A *stack* is a series of consecutive locations in memory. Typically, a program uses a stack to pass arguments between subroutine calls and to save the program state when servicing a fault. After executing a subroutine or handling a fault, the processor restores the program and continues program execution.

The *narrow stack* is a contiguous set of words that support ECLIPSE 16-bit program development and upward program compatibility. Narrow stack management relies on three 16-bit narrow stack management parameters, per memory segment, which are maintained in memory. Refer to the chapter, "ECLIPSE 16-Bit Programming," for additional information on the narrow stack.

The *wide stack* is a contiguous set of doublewords that support the ECLIPSE MV/Family 32-bit programs. Wide stack management relies on four 32-bit wide stack management parameters, for each memory segment. A memory *segment* is a logically addressable subset of memory. Refer to the section, "Memory Management," for additional information on memory and segments.

Wide stack management for the current segment also includes four 32-bit wide stack management registers. You access a stack management register with instructions that load or store a register value. Refer to the chapter, "Stack Management," for additional information on the wide stack.

The following list summarizes the wide stack management registers.

- *Wide stack base* (WSB) defines the lower limit of the wide stack.
- *Wide stack limit* (WSL) defines the upper limit of the wide stack.
- *Wide stack pointer* (WSP) addresses the current top-most location on the wide stack.
- *Wide frame pointer* (WFP) defines a reference point in the wide stack.

Program Flow Management

In program flow management the processor controls program execution (such as calling a subroutine) and handles faults. This section summarizes program control; the chapter, "Program Flow Management," provides additional information.

The processor controls program flow with a 31-bit program counter (PC). Figure 1-4 shows the format of the program counter. Table 1-1 describes the format.

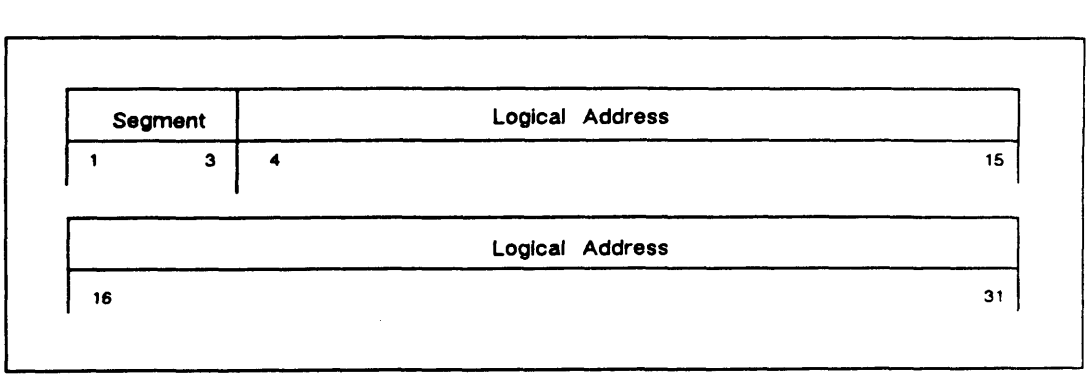


Figure 1-4 *Program counter format*

Table 1-1 *Program counter format*

Name	Bits	Meaning
Segment	1-3	Current segment. The processor provides specific procedures for modifying the current segment field.
Logical Address	4-31	Logical word address within the segment. During normal program flow, the processor increments bits 4-31 of the program counter. Thus, address wraparound occurs within the current segment.

Queue Management

In queue management, elements are inserted, deleted, and searched for in a queue. A *queue* is a variable-length list of linked entries. Typically, an operating system uses queues to track processes that it must perform, such as printing files on a line printer.

Refer to the chapter, "Queue Management," for further information on the queue facilities and management.

Graphics Management

The optional Graphics Instruction Set (GIS) performs high speed graphic functions in ECLIPSE MV/Family systems. GIS supports windowing systems in which several programs share one bitmap. The instruction set includes nonprivileged and privileged instructions. Privileged instructions maintain the various databases. Nonprivileged instructions perform operations such as reading or writing a single pixel, drawing lines, or filling in a bitmap region with a solid color.

Refer to the chapter, "Graphics Management," for more detailed information on GIS as well as the *ECLIPSE MV/Family (32-Bit) Systems Instruction Dictionary* for instruction specifics.

Device Management

In device management, the processor transfers data between memory and a device. The processor transfers data in bytes, words, or blocks of words using three transfer facilities: programmed input/output (I/O); data channel I/O (DCH); or high speed burst multiplexor channel (BMC). The chapter, "Device Management," summarizes the three transfer facilities.

Programmed I/O

Bytes or words may be transferred between an accumulator and a device with the programmed I/O facility. The programmed I/O facility may be used to transfer data with a slow speed device, or to initialize a data channel or a burst multiplexor channel transfer.

Data Channel I/O

The data channel I/O initiates a transfer of words between memory and a device. The data channel accesses memory directly, with or without a device map. Thus, the data transfer bypasses the accumulators.

Burst Multiplexor Channel

The high speed burst multiplexor channel initiates a transfer of blocks of words between memory and a device. Memory is directly accessed, with or without a device map, with the burst multiplexor channel. Thus, the data transfer bypasses the accumulators.

System Management

System management facilities determine processor-dependent configurations, such as processor identification and size of the main memory. Refer to the chapter, "Memory and System Management," for additional information.

Memory Management

The processor provides a *logical address space* of 4 gigabytes (1 gigabyte equals 2^{30} bytes). The address space is divided into eight segments and rings, which facilitate memory management. A *segment* is an addressable unit of memory that contains programs and data. A *ring* is a collection of protection mechanisms, which safeguards the contents of a segment.

As rings and segments are similar and inter-related, this manual uses the term *segment* to indicate either term or both terms.

The processor addresses a segment through a 0–7 numbering system, and each segment contains 512 Mbytes. Figure 1–5 illustrates the concept of the segments and their contents.

- Segment 0. The processor uses this segment to execute privileged and nonprivileged instructions as the kernel operating system.
- Segments 1–7. The processor uses these segments to execute nonprivileged instructions. Refer to the appropriate operating system programmer's manual for information on the implementation-dependent use of segments (typically, segments 1–3 are used by the operating system).

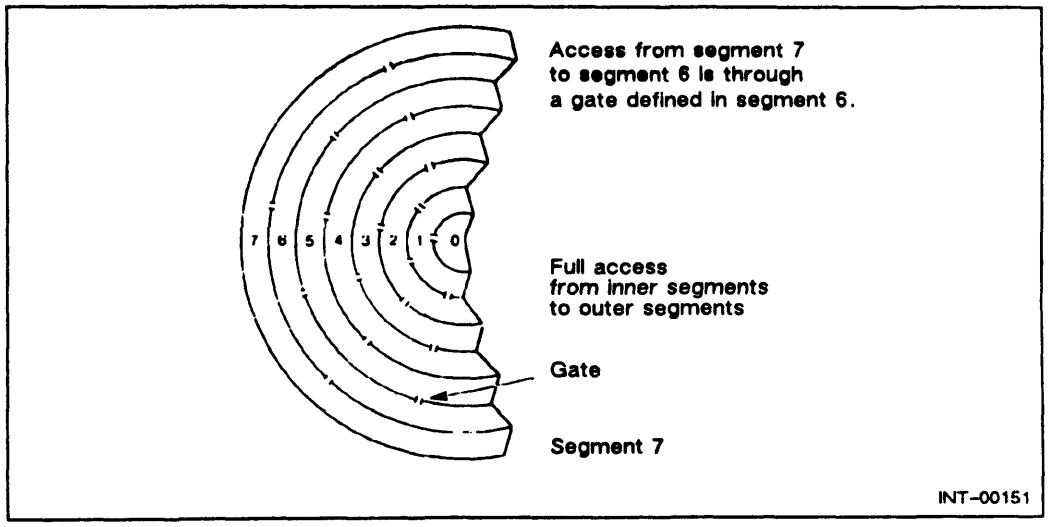


Figure 1–5 *Logical address space*

As the logical address space is larger than physical memory, the processor uses *virtual memory* to provide the full range of logical addresses.

- The virtual memory system translates logical addresses to physical addresses.

  A *logical address* specifies a segment number and a logical location within the segment. You write programs using these logical addresses. The processor converts the logical addresses to physical addresses, and then accesses the contents of the location specified.

- The operating system may store the virtual memory on disk in 2-Kbyte units called *pages*.

  When processing needs a page on disk, the operating system moves the page to physical memory for manipulation. This page-swapping system is called demand paging.

The hardware facilities for address translation include eight segment base registers (SBR0 through SBR7), which define eight memory segments and their access protocols. The processor's address translation capability is explained in the section, "Accessing Memory."

Using a privileged instruction, you can load the contents of a segment base register. Refer to the chapter, "Memory and System Management," for additional information.

ECLIPSE 16-Bit Compatible Instructions

The processor contains an ECLIPSE compatible instruction set (and stack facilities) for 16-bit program development and upward program compatibility. Most programs that execute on ECLIPSE 16-bit computers also execute on ECLIPSE MV/Family computers without recompiling or reassembling. Some ECLIPSE 16-bit instructions that refer to memory locations may need to be modified before using them in ECLIPSE MV/Family system-specific programs. Refer to the chapter, "ECLIPSE 16-Bit Programming," for additional information.

This manual refers to ECLIPSE 16-bit instructions as ECLIPSE or 16-bit instructions, and ECLIPSE MV/Family specific instructions as MV or 32-bit instructions.

Accessing Memory

The processor addresses and accesses memory for an instruction or for an operand. To address memory, the processor uses a 16-bit word as the standard unit of address.

NOTE: *For most efficient performance, 32- and 64-bit data should be aligned on doubleword boundaries.*

The instruction that the processor accesses can be a single word or multiple words. The operand can be a bit, byte, word, doubleword, or multiple words. A *memory reference instruction* refers to a class of instructions that access memory for data or for another instruction. The memory reference instructions contain the information for

- Determining the effective address of an operand. The processor reads or writes an operand.

- Determining the effective address of the next nonsequential instruction. The processor modifies the program counter with the effective address, and then executes the instruction that the program counter identifies.

A memory reference instruction attempts to access memory in the current segment or in another segment. The validity of the access depends on a comparison of the access protocols permitted for the memory page and the type of access that the instruction attempts to perform. The access protocols are explained in the next two sections "Current Segment" and "Other Segments."

Current Segment

When a memory reference instruction addresses the current segment, the processor compares the page protocols with the type of access that the instruction requests, determining the validity of the reference. The *page protocols* are identified as a valid page, read access, write access, and execute access.

For instance, when loading a byte into an accumulator from the current segment, the processor reads the byte from memory if it resides where the page protocols permit a read access.

The processor also compares the segment field of every indirect address reference with the current segment. For accessing data (read or write access), indirect addressing can occur within the current segment or towards a higher numbered segment. For transferring program control (execute access), indirect addressing must occur in the current segment.

If a reference is invalid, the processor aborts the access and services the fault (a protection violation). Refer to the chapter, "Memory and System Management," for further information on page access and protection violation faults.

Other Segments

When executing a memory reference instruction that addresses another segment,

- The processor compares the current segment with the destination segment to determine the directional validity of the reference. The *destination segment* contains the operand or nonsequential instruction.

  Read or write access must be to the current segment or to a higher numbered segment.

- The processor compares the segment and page protocols with the type of access that the instruction requests to determine the access validity of the reference. The processor first checks the segment protocols and then checks the page protocols.

  If read or write access to a higher numbered segment is requested, the segment protocol checks whether the segment is valid. For instance, when loading a byte into an accumulator from a higher numbered segment, the processor reads the byte only if it resides in a valid segment and page protocols permit a read access.

If the reference is invalid, the processor aborts the access and services the protection violation fault. Refer to the chapter, "Memory and System Management," for further details on page accesses and protection violation faults.

Memory Reference Instructions

Memory reference instructions access memory using either word or byte addressing. Figure 1-6 shows the typical memory reference instruction formats for word addressing. Figure 1-7 shows the typical memory reference instruction formats for byte addressing. The mnemonic "op" is the operation indicator. The instruction formats for word addressing contain an indirect (@) field. The instruction formats for word and byte addressing contain *index* and *displacement* fields, and also an optional accumulator (*ac*) field. The optional accumulator field specifies a source or destination accumulator in the range of zero through three.

For instance, if the *ac* field is equal to zero (*ac* = 0) for a load accumulator instruction, the processor loads an operand from memory into the destination accumulator (AC0 or FPAC0).

The combination of *index*, *displacement*, and @ (indirect) fields specifies the effective address that contains the instruction or operand. To resolve the effective address, the processor first identifies the addressing mode; then any indirect address(es), and finally the effective address.

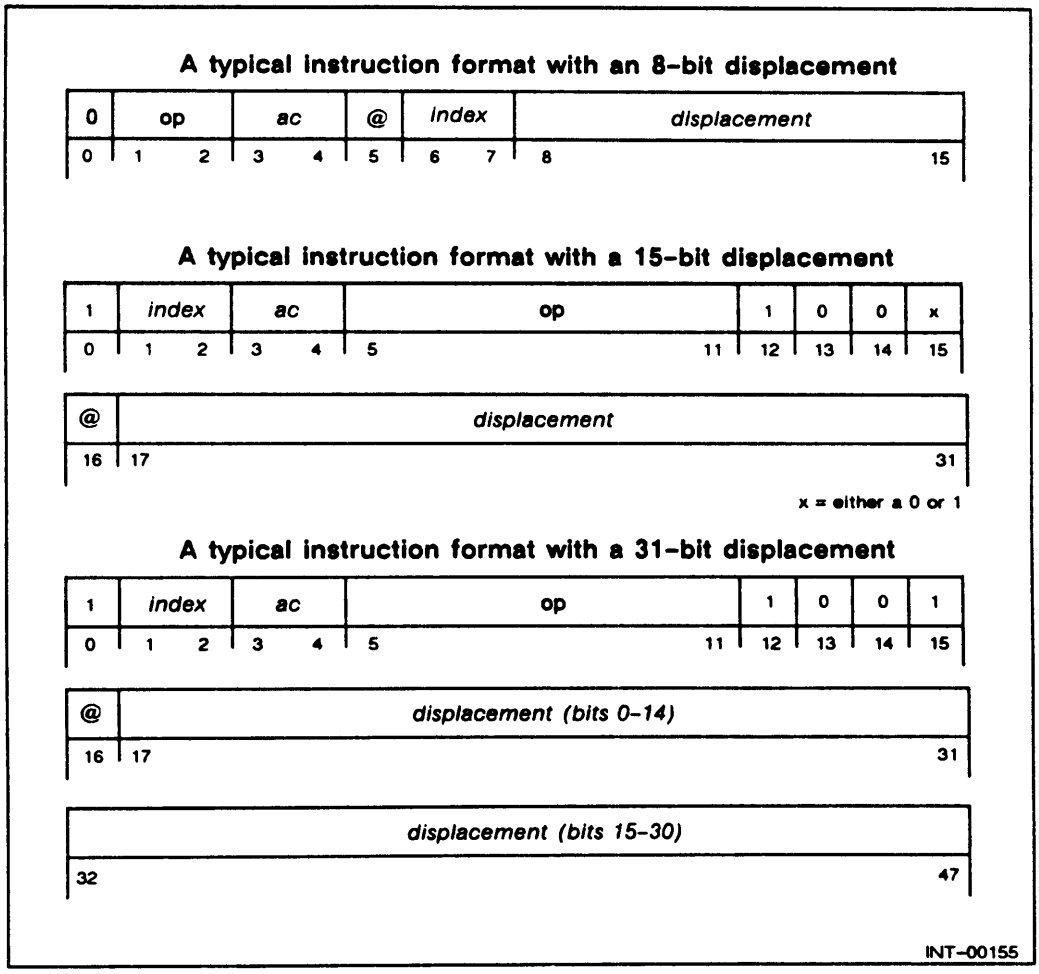


Figure 1-6 *Memory reference instruction word addressing formats*

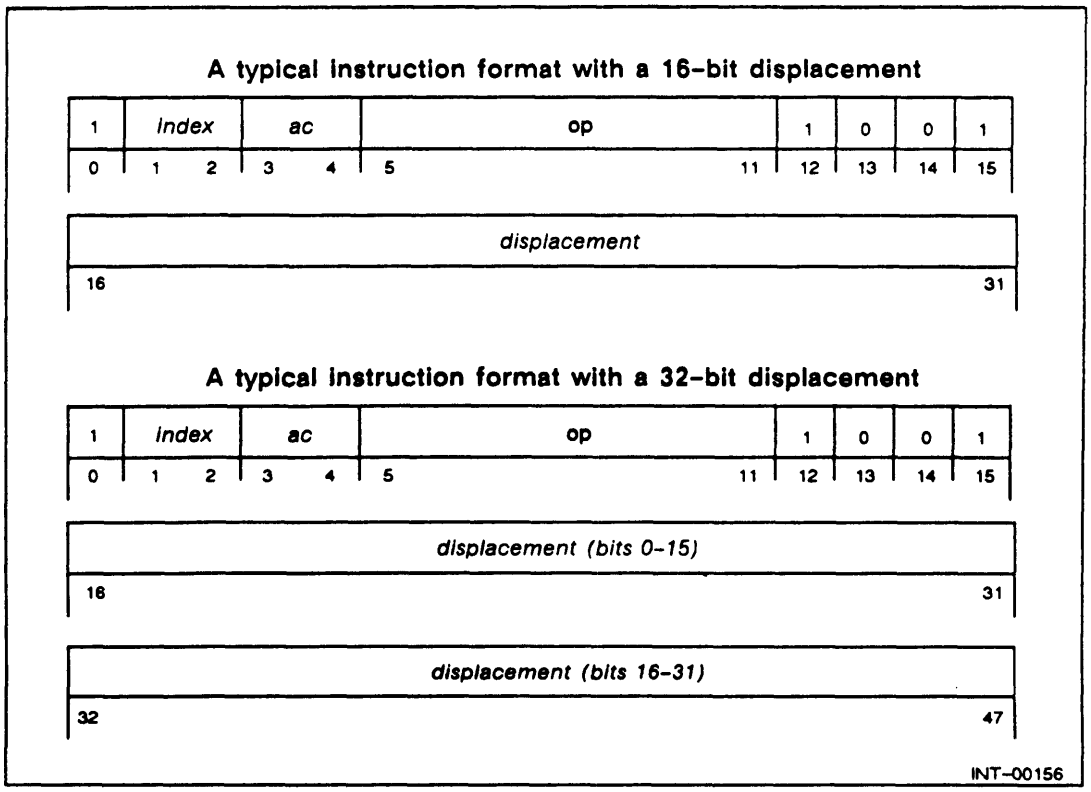


Figure 1-7 *Memory reference instruction byte addressing formats*

Address Modes

Using the index field (Table 1-2), the processor determines if the instruction specifies an absolute or relative addressing mode. The Assembler (in conjunction with the appropriate pseudo-op) produces object code with absolute or relative addressing.

Absolute Addressing

For absolute addressing, the displacement field contains an indirect or an effective address. The address, expressed as an unsigned integer (8, 15, or 31 bits wide), specifies an addressing range as shown in Table 1-2.

With a few exceptions (**LDA, LDB, LDI, LDIX, LEF, LSN,** and **XOP0**), an assembler mnemonic of a memory reference instruction indicates the size and the range of the displacement. For instance, a memory reference instruction

- Without the X or L prefix uses a standard displacement of 8 bits.
- With the X or E prefix uses an extended displacement of 15 bits.
- With the L prefix uses a long displacement of 31 bits.

NOTE: *When using an 8- or 15-bit displacement in absolute addressing, the processor zero-extends the displacement to 28 bits, and uses the current segment for the 3 high-order bits of the effective address.*

Thus, the displacement becomes an indirect address or an effective absolute address.

Relative Addressing

For relative addressing, the index field defines a register (Table 1-2) the contents of which become a base address. The processor adds the base address to the displacement (8-, 15-, or 31-bit two's-complement integer). When using an 8- or 15-bit displacement, the processor sign-extends the displacement to 31 bits.

In addition, if executing an instruction with an extended (15-bit) or long (31-bit) displacement, the processor adds a constant to the sum for program relative addressing. The additional increment adjusts the sum to address the first word of the displacement, which begins following the word that contains the instruction opcode. An instruction with an 8-bit displacement contains the displacement in the same word as the opcode.

Thus, the address becomes an indirect or effective relative address.

Indirect and Effective Addresses

When the indirect field equals zero, the absolute or relative address becomes the effective address. The processor translates an effective address to a physical address, and accesses the physical address.

When the indirect field equals one, the absolute or relative address becomes an indirect address (or pointer). The processor translates the indirect address to a physical address and uses the contents of that physical address as another indirect or direct address.

NOTE: *For an ECLIPSE 16-bit compatible instruction, the processor accesses a single word in memory as an indirect pointer; otherwise, the processor accesses a doubleword.*

The processor tests bit 0 of the pointer contents, which defines additional (if any) indirect addressing.

- When bit 0 equals zero, the contents become the effective address.

  The processor translates the effective address to a physical address and accesses it.

- When bit 0 equals one, the contents become another pointer.

  The processor continues to resolve pointers until bit 0 equals zero.

  With the address translation unit (ATU) enabled, the processor can resolve up to 15_{10} pointers. However, for an instruction that can specify two indirect-addressing chains (such as **WBLM**), the total number of pointers for the two chains should be equal to or less than 15.

NOTE: *If the processor attempts to resolve more than 15 indirect addresses (with the ATU enabled), a protection violation occurs. Though the actual number of indirections (for one or two chains of the same instruction) may vary slightly with different processors, at some level of indirection the processor will trap. All ECLIPSE MV/Family processors will detect infinite indirection.*

Table 1-2 *Effective addressing*

Address Mode	Index Bits	Intermediate Logical Address*	Prefix †	Displacement Range Octal Words (decimal)
Absolute	00	D		0 to 377 (0 to 255) (current ring)
		D	E or X	0 to 077777 (0 to 32,767) (current ring)
		D	L	0 to 17777777777 (0 to 2,147,483,647)
PC Relative	01	PC+D		- 200 to + 177 (- 128 to + 127) (current ring)
		PC+*n*+D	E or X	- 40000 to + 37777 (- 16,384 to + 16,383)
		PC+*n*+D	L	-10000000000 to + 07777777777 (-1,073,741,824 to 1,073,741,823)
AC2 Relative	10	AC2+D		- 200 to + 177 (- 128 to + 127) (current ring)
		AC2+D	E or X	- 40000 to + 37777 (- 16,384 to + 16,383)
		AC2+D	L	-10000000000 to + 07777777777 (-1,073,741,824 to + 1,073,741,823)
AC3 Relative	11	AC3+D		- 200 to + 177 (- 128 to + 127) (current ring)
		AC3+D	E or X	- 40000 to + 37777 (- 16,384 to + 16,383)
		AC3+D	L	-10000000000 to + 07777777777 (-1,073,741,824 to + 1,073,741,823)

* *The processor ignores bit 0 of PC, AC2, and AC3 when calculating the intermediate logical address.*

† *E, X or L corresponds to the prefix of an instruction mnemonic, which identifies an instruction containing an ECLIPSE 16-bit extended (E) displacement field or ECLIPSE 32-bit extended (X) or long (L) displacement field.*

n *The n variable in the PC relative addressing mode equals the number of words that precede the first word of the displacement for the current instruction.*

Operand Access

Before accessing a memory operand, the processor first resolves an effective address.

The processor accesses an operand as a bit, byte, several bytes, word, doubleword, or several doublewords. Tables 1-3 and 1-4 show the relations between instructions that load an accumulator with word-oriented and byte data, respectively. The following sections explain the word, byte, and bit accesses. (To access several bytes, the processor must first access a byte; to access several words or doublewords, it must first access a word.)

Table 1-3 *Word-oriented data*

Instruction	Word Address Width (Bits)	Displacement Width (Bits)
LDA	15	8
ELDA	15	15
XNLDA	31	15
XWLDA	31	15
LNLDA	31	31
LWLDA	31	31

Table 1-4 *Byte data*

Instruction	Byte Pointer Address Width (Bits)	Displacement Width (Bits)	Index =
STB	16	0	Byte address in any accumulator
ESTB	16	16	Word address (absolute, PC-relative, AC-relative)
WSTB	32	0	Byte address in any accumulator
XSTB	32	16	Word address (absolute, PC-relative, AC-relative)
LSTB	32	32	Word address (absolute, PC-relative, AC-relative)

Word

The processor accesses a word operand for computations involving narrow (16-bit) data. An instruction mnemonic with a prefix of N (such as **NADD**) indicates a narrow or one word operand. An instruction that requests a word in memory (such as **XNLDA**) supplies the effective address parameters to the processor. The processor then resolves the effective address.

Doubleword

The processor accesses a doubleword operand for computations involving data widths of 32 or more bits. A fixed-point instruction mnemonic with a prefix of W (such as **WADD**) indicates a wide or two-word operand. A single-precision floating-point instruction (such as **LFAMS**) requires one doubleword, while a double-precision instruction (such as **LFAMD**) requires two doublewords.

An instruction that requests a doubleword (such as **XWLDA**) supplies the effective address parameters to the processor. The processor then resolves the effective address, which points to the first word of the doubleword operand.

Byte

An instruction that requests a byte forms a byte pointer from the contents of an accumulator or from the contents of the index field and the 16- or 32-bit displacement. (The accumulator specified by the index field holds a word pointer. The processor multiplies this word pointer by two to create a byte pointer.) A byte pointer consists of an effective address and a byte indicator. The least significant bit of the byte pointer contains the byte indicator.

NOTE: *Byte addressing excludes indirect addressing.*

The processor identifies a byte as follows:

- 16-Bit displacement

  For an instruction with a 16-bit displacement (such as **XLDB**), the processor extends the displacement to 29 bits (absolute addressing) or 32 bits (relative addressing), calculates the effective address, and then identifies the byte. Note that when the processor extends the displacement to 29 bits, it also appends the current ring of execution (as bits 0-2) to create a 32-bit address.

- 32-Bit displacement

  For an instruction with a 32-bit displacement (such as **LLDB**), the processor calculates the effective address, and then identifies the byte.

- Accumulator

  For an instruction that requires a byte pointer in an accumulator, you must first use a load effective byte address instruction (such as **LLEFB**). The load effective byte address instruction calculates an effective byte address, and then loads it into an accumulator.

  Although identification of the bit numbers depend on the byte pointer location, the format of a byte pointer remains identical, regardless of its location. Figure 1-8 shows the formats for a byte pointer; Table 1-5 describes the formats.

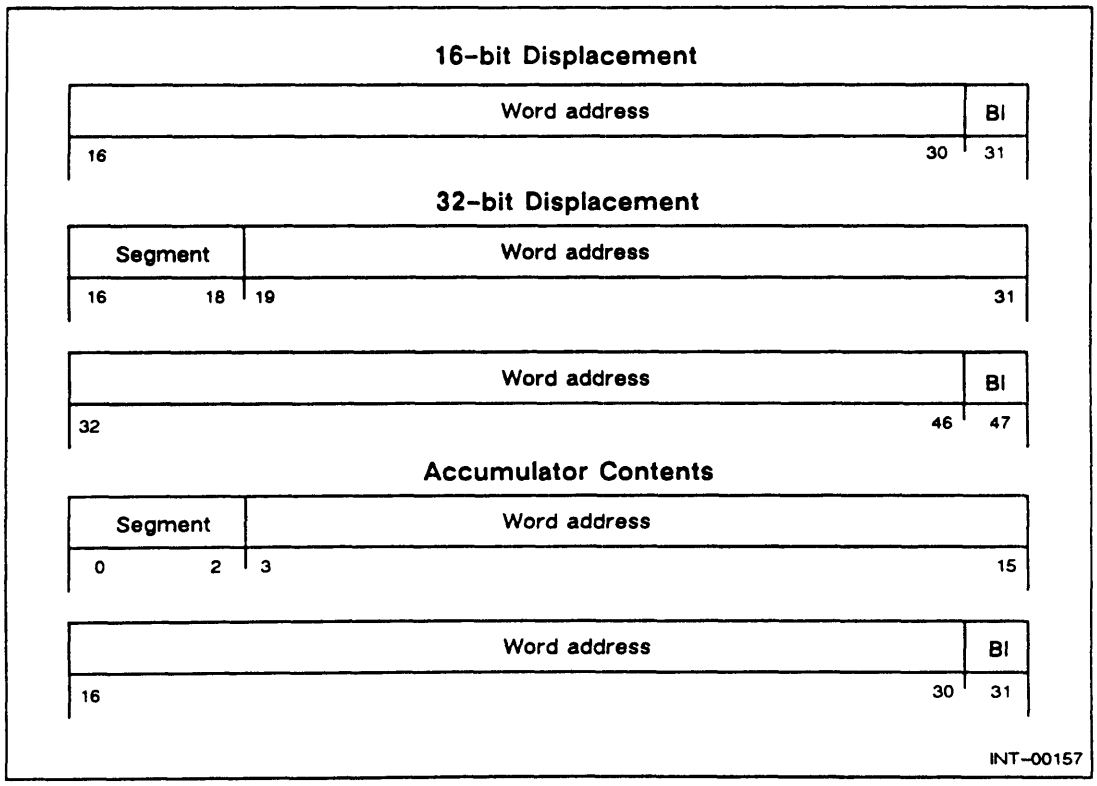


Figure 1-8 *Byte pointer format*

Table 1-5 *Byte pointer contents*

Name	Meaning
Segment	This field identifies either the current segment or an outward memory segment.
Word address	This field identifies a 16-bit word in the memory segment.
BI	This field identifies the byte. When the BI field equals 0, the processor accesses the most significant byte (bits 0-7). When the BI field equals 1, the processor accesses the least significant byte (bits 8-15).

The processor accesses the word and then locates the byte as Figure 1-9 shows.

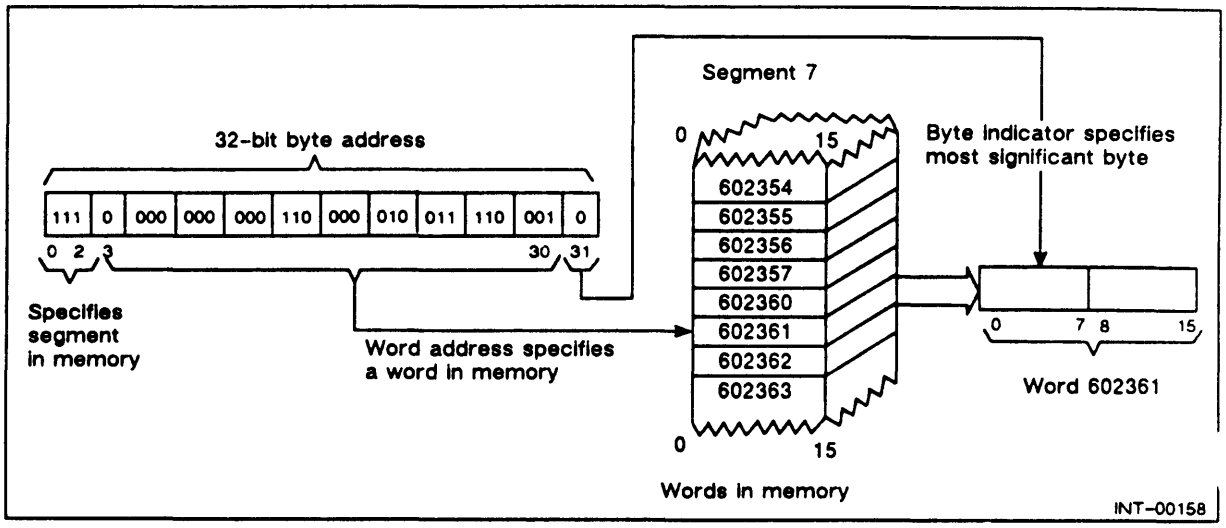


Figure 1-9 *Byte addressing*

Bit

An instruction that accesses a bit in memory (such as **WBTO, WBTZ, WSNB, WSZB,** and **WSZBO**) forms a bit pointer from the contents of two accumulators. The bit pointer is composed of a word pointer and a bit identifier. The word pointer consists of an effective address (in the *acs* accumulator) and a word offset (in the *acd* accumulator). The bit identifier is located in the least significant bits of the *acd* accumulator.

Figure 1-10 shows the accumulator formats for the **WBTO, WBTZ, WSNB, WSZB,** and **WSZBO** instructions; Table 1-6 describes the formats.

Table 1-6 *Bit pointer contents*

Name	Meaning
@	Specifies indirection. When it equals 1, it identifies an indirect address; when it equals 0, it identifies a direct address.
Segment	Identifies either the current segment or an outward memory segment.
Base word address	Identifies a 16-bit word in the specified memory segment.
Base word offset	Contains an unsigned integer which the processor adds to the effective address to arrive at a final word address (see Figure 1-11).
Bit identifier	Specifies the bit position (0-15) in the final word.

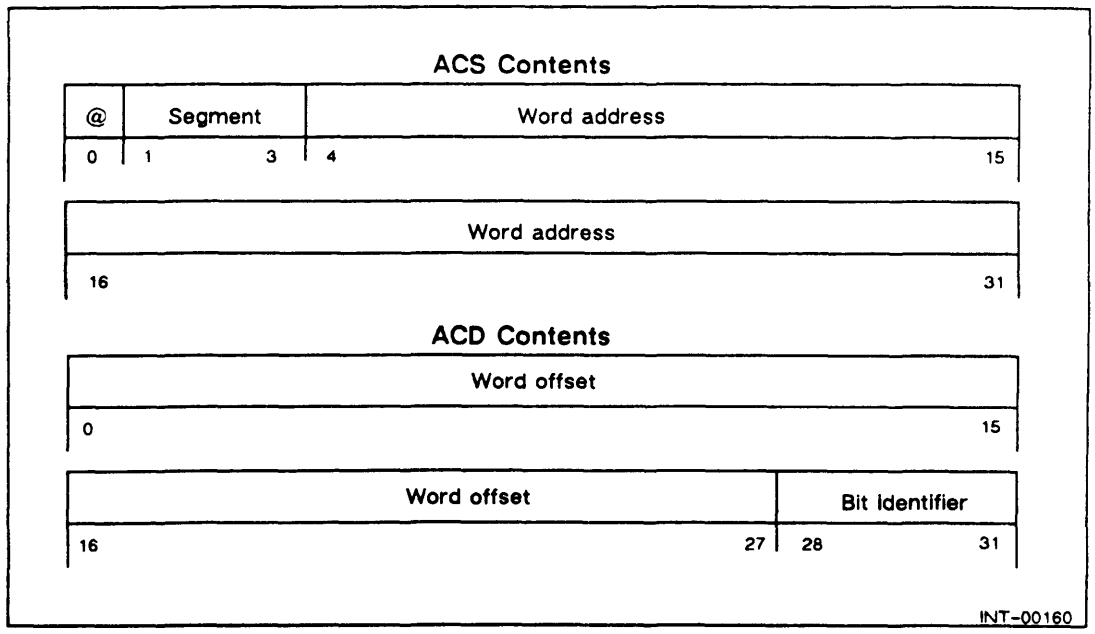


Figure 1-10 *Bit pointer format*

The processor uses the *acs* accumulator contents to calculate an effective address. If a bit instruction specifies the two accumulators as the same accumulator, then the base word address is zero in the current segment.

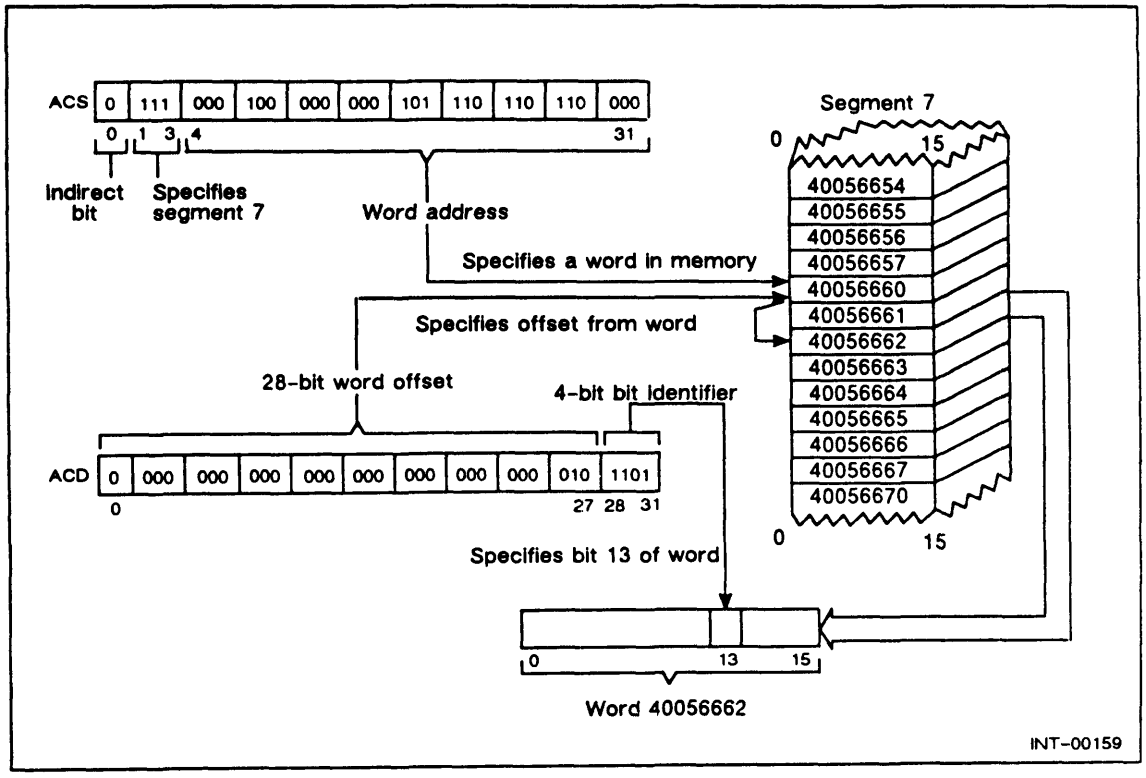


Figure 1-11 *Bit addressing*

Protection Capabilities

While executing an instruction, the processor checks the validity of a memory reference or an I/O operation (protection violation), a page reference (nonresident page), a stack operation, a computation, and a data format. Table 1-7 lists the validity checks (or faults).

Table 1-7 *Faults*

Fault	Type
Nonresident page	Privileged
Protection violation	Privileged
Stack operation	Nonprivileged
Fixed-point computation	Nonprivileged
Floating-point computation	Nonprivileged
Invalid decimal or ASCII data format	Nonprivileged

If the processor detects an error, a nonprivileged or privileged fault occurs before the next instruction is executed. A nonprivileged fault occurs when the processor detects a computation error. The processor limits I/O access on a per ring basis, and limits memory access using a hierarchical protection mechanism. For instance,

- Before executing an I/O instruction, the processor checks the I/O validity flag in the current segment.
- Before executing a memory reference instruction, the processor checks the validity of the reference.

The processor executes an I/O or memory reference instruction when validity checks permit the access. Otherwise, the processor initiates a protection violation. Thus, an operating system can restrict access to the devices to specific segment(s).

Accessing and changing a protection mechanism requires a privileged instruction (executable only in segment 0) or data access, typically controlled by the operating system. Refer to the chapter, "Program Flow Management," for further details on servicing a nonprivileged fault.

A privileged fault occurs when an operation is not permitted by the address translation mechanism (page not resident, I/O protection) or by the ring structure (privileged instruction, outward call). Refer to the chapters, "Memory and System Management" and "Program Flow Management," for further details on servicing a privileged fault.

End of Chapter

2

Fixed-Point Computing

Using fixed-point computation the processor can add, subtract, multiply, and divide 16- and 32-bit signed (two's-complement) and unsigned binary data. The processor can also perform logical operations on 16- and 32-bit data.

In addition to binary arithmetic and logical operations, the processor can manipulate 8-bit bytes (as alphanumeric ASCII data) and can perform binary coded decimal (BCD) arithmetic. The processor performs the byte manipulation with fixed-point operations, and performs BCD arithmetic with fixed-point and floating-point operations.

Following a computation, the processor can shift (arithmetically or logically) the contents of an accumulator and can skip on a condition (the result of the computation and/or shift). Finally, the processor can store the result in an accumulator or memory.

This chapter explains the various computations (binary, logical, decimal and byte) and the processor status register.

Binary Operations

The processor performs fixed-point binary arithmetic in the arithmetic logic unit (ALU). Move, arithmetic, shift, and skip instructions control processor and arithmetic logic unit operations.

Data Formats

The majority of fixed-point arithmetic instructions require two's-complement (signed) binary numbers. For instance, the **ADD** instruction adds two 16-bit two's-complement binary numbers. The 16- and 32-bit numbers must begin on word boundaries. Figure 2-1 shows the fixed-point accumulator formats for the 16- and 32-bit two's-complement numbers; Table 2-1 describes the formats. Table 2-2 shows the precision of 16- and 32-bit two's complement (signed) binary numbers.

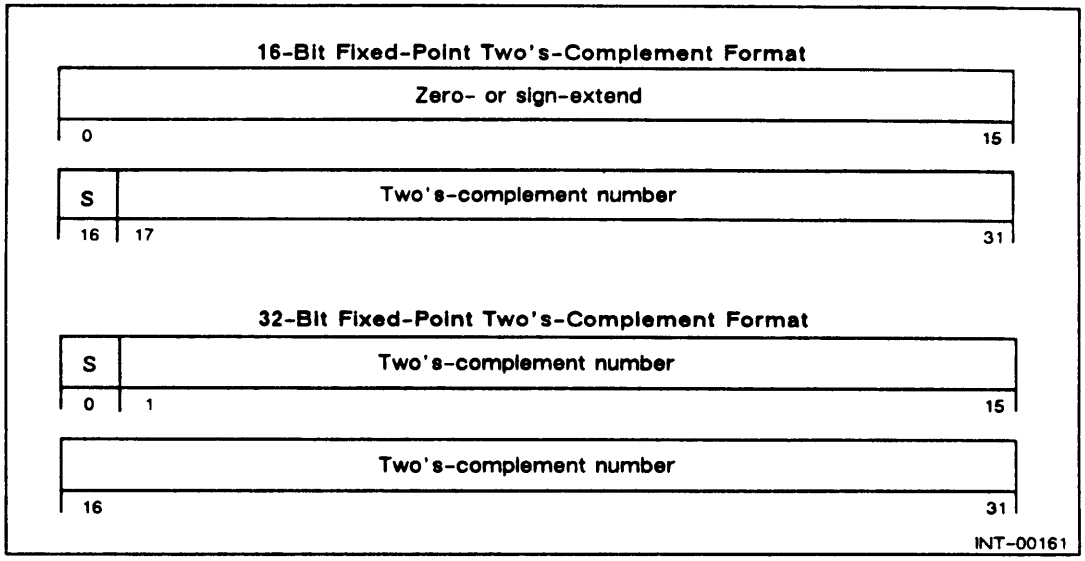


Figure 2-1 *Fixed-point two's-complement data formats*

Table 2-1 *Fixed-point two's-complement formats*

Name	Contents
Zero- or sign-extend	These bits contain either 16 zeros or 16 ones. For moving and computing narrow data (depending on the instruction), the processor sign-extends the narrow data either when loading it into an accumulator, or (when converting it to wide data) before or after the narrow data operations.
S	The sign bit. Bit 16 contains the sign bit for narrow data; bit 0 contains the sign bit for wide data. The sign bit equals zero for a positive number and equals one for a negative number.
Two's-Complement Number	The processor requires two's-complement binary numbers for the majority of fixed-point arithmetic computation.

Table 2-2 *Range of 16- and 32-bit fixed-point numbers (in octal)*

Form of Data	16-bit Precision	32-bit Precision
Signed (two's complement)	-32,768 to +32,767	-2,147,483,648 to +2,147,483,647
Unsigned	0 to 65,535	0 to 4,294,967,295

Table 2-3 lists the instructions that explicitly convert 16-bit data to or from 32-bit data. Other tables in this chapter list the instructions that convert the precision before or after another function. For instance, when loading narrow data (16-bit) from memory into an accumulator, the processor sign-extends the number before loading it. When executing a narrow fixed-point instruction (NADD), the processor sign-extends the result.

Table 2-3 *Fixed-point precision conversion*

Instruction	Operation
CVWN	Convert from 32-bit to 16-bit
SEX	Sign-extend 16-bits to 32-bits
ZEX	Zero-extend 16-bits to 32-bits

Move Instructions

The move instructions transfer data between accumulators, load or store data between memory and an accumulator, calculate and load a value into an accumulator, or move blocks of words between memory locations. Table 2-4 lists these instructions according to type of operation.

The block move instructions (such as WBLM) require an effective address in one or more accumulators. Use a load effective address instruction to calculate and load the effective address into an accumulator.

Table 2-4 *Fixed-point data movement instructions*

Instruction Type Instruction	Operation
Between Accumulators	
MOV *	Move and skip
WMOV	Wide move
WXCH	Wide exchange accumulators
XCH *	Exchange accumulators
Betwen memory and an accumulator	
ELDA *	Extended load accumulator
ESTA *	Extended store accumulator
LDA *	Load accumulator
LDATS	Load accumulator with doubleword addressed by WSP
LNLDA	Narrow load accumulator (long displacement)
LNSTA	Narrow store accumulator (long displacement)
LWLDA	Wide load accumulator
LWSTA	Wide store accumulator
NLDAI	Narrow load immediate
STA *	Store accumulator
STATS	Store accumulator into doubleword addressed by WSP
XNLDA	Narrow load accumulator
XNSTA	Narrow store accumulator
XWLDA	Wide load accumulator
XWSTA	Wide store accumulator
WLDAI	Wide load with wide immediate
Calculate and load into an accumulator	
ELEF *	Extended load effective address
LEF *	Load effective address
LLEF	Load effective address (long displacement)
XLEF	Load effective address
Between memory locations	
BAM *	Block add and move
BLM *	Block move
WBLM	Wide block move

* *ECLIPSE compatible instruction*

Arithmetic Instructions

The arithmetic instructions perform computations on fixed-point values. The basic operations include addition, subtraction, multipication, and division. An additional arithmetic operation is incrementing or decrementing a value and then skipping the next sequential 16-bit word in the instruction stream depending on the results of the new value.

Arithmetic functions are performed on either two values in accumulators, an immediate value and an accumulator, an immediate value and a memory location, or a memory location and an accumulator. Tables 2-5 through 2-9 list the arithmetic instructions according to arithmetic function and type of operation.

NOTE: *The ECLIPSE 16-bit compatible instructions (such as,* **ADC, ADD, MUL,** *and* **DIVS***) ignore bits 0-15 of the source accumulator. The results of ECLIPSE 16-bit compatible instructions leave bits 0-15 of the destination accumulator undefined, except where noted otherwise.*

Table 2-5 *Fixed-point addition instructions*

Operation Type		Operand Values		
Instruction	Operation	Source	Destination	Result
Accumulator to Accumulator				
ADC *	Add complement and skip	U16	U16	U16
ADD *	Add and skip	U16	U16	U16
NADD	Narrow add	S16	S16	S32
WADC	Wide add complement	S32	S32	S32
WADD	Wide add	S32	S32	S32
ADDI *	Extended add immediate	S16	S16	S16
ADI *	Add immediate	2I	U16	U16
Immediate to Accumulator				
INC *	Increment and skip	1	U16	U16
NADDI	Narrow extended add immediate	S16	S16	S32
NADI	Narrow add immediate	2I	S16	S32
WADDI	Wide add with wide immediate	S32	S32	S32
WADI	Narrow add immediate	2I	S32	S32
WINC	Wide increment	1	U32	U32
WNADI	Wide add with narrow immediate	S16	S32	S32
Immediate to Memory				
LNADI	Narrow add immediate	2I	S16	S16
LWADI	Wide add immediate	2I	S32	S32
XNADI	Narrow add immediate to memory word	2I	S16	S16
XWADI	Add immediate to memory doubleword	2I	S32	S32
Memory to Accumulator				
LNADD	Narrow add memory word to accumulator	S16	S16	S32
LWADD	Wide add memory to accumulator	S32	S32	S32
XNADD	Narrow add memory to accumulator	S16	S16	S32
XWADD	Wide add memory to accumulator	S32	S32	S32

S16 = Signed 16-bit integer
U16 = Unsigned 16-bit integer
S32 = Signed 32-bit integer
U32 = Unsigned 32-bit integer
2I = 2-bit integer in range 1 to 4

** ECLIPSE compatible instruction*

Table 2-6 *Fixed-point subtraction instructions*

Operation Type Instruction	Operation	Operand Values Source	Destination	Result
Accumulator from Accumulator				
NSUB	Narrow subtract	S16	S16	S32
SUB *	Subtract and skip	U16	U16	U16
WSUB	Wide subtract	S32	S32	S32
Immediate from Accumulator				
NSBI	Narrow subtract immediate	2I	S16	S32
SBI *	Subtract immediate	2I	U16	U16
WSBI	Wide subtract immediate	2I	S32	S32
Immediate from Memory				
LNSBI	Narrow subtract immediate	2I	S16	S16
LWSBI	Wide subtract immediate	2I	S32	S32
XNSBI	Narrow subtract immediate	2I	S16	S16
XWSBI	Wide subtract immediate	2I	S32	S32
Memory from Accumulator				
LNSUB	Narrow subtract memory immediate	S16	S16	S32
LWSUB	Wide subtract memory word	S32	S32	S32
XNSUB	Narrow subtract memory word	S16	S16	S32
XWSUB	Wide subtract memory	S32	S32	S32

S16 = Signed 16-bit integer
U16 = Unsigned 16-bit integer
S32 = Signed 32-bit integer
U32 = Unsigned 32-bit integer
2I = 2-bit integer in range 1 to 4

* *ECLIPSE compatible instruction*

Table 2-7 *Fixed-point multiplication instructions*

Operation Type Instruction	Operation	Operand Values Source	Destination	Result
Accumulator by Accumulator				
MUL *	Unsigned multiply	U16	U16	U32
MULS *	Signed multiply	S16	S16	S32
NMUL	Narrow sign-extend multiply	S16	S16	S32
WMUL	Wide multiply	S32	S32	S32
WMULS	Wide signed multiply	S32	S32	S64
Accumulator by Memory				
LNMUL	Narrow wide multiply memory word	S16	S16	S32
LWMUL	Wide multiply memory word	S32	S32	S32
XNMUL	Narrow multiply memory word	S16	S16	S32
XWMUL	Wide multiply memory word	S32	S32	S32

S16 = Signed 16-bit integer
U16 = Unsigned 16-bit integer
S32 = Signed 32-bit integer
U32 = Unsigned 32-bit integer
2I = 2-bit integer in range 1 to 4

* *ECLIPSE compatible instruction*

Table 2-8 *Fixed-point division instructions*

Operation Type Instruction	Operation	Operand Values Divisor	Dividend	Quotient/Remainder
Accumulator by Accumulator				
DIV *	Unsigned divide	U16	U32	U16/U16
DIVS *	Signed divide	S16	S32	S16/S16
DIVX *	Sign-extend and divide	S16	S16	S16/S16
HLV *	Halve signed	2	S16	S16/—
NDIV	Narrow sign-extend divide	S16	S16	S32/—
WDIV	Wide divide	S32	S32	S32/—
WDIVS	Wide signed divide	S32	S64	S32/S32
WHLV	Wide halve signed	2	S32	S32/—
Accumulator by Memory				
LNDIV	Narrow divide memory word	S16	S16	S32/—
LWDIV	Wide divide memory word	S32	S32	S32/—
XNDIV	Narrow divide by memory word	S16	S16	S32/—
XWDIV	Wide divide by memory doubleword	S32	S32	S32/—

S16 = Signed 16-bit integer
U16 = Unsigned 16-bit integer
S32 = Signed 32-bit integer
U32 = Unsigned 32-bit integer
— = Not applicable

* *ECLIPSE compatible instruction*

Table 2-9 *Fixed-point increment or decrement value and skip instructions*

Instruction	Operation
DSZ *	Decrement and skip if zero
DSZTS	Decrement the doubleword addressed by WSP (skip if zero)
EDSZ *	Extended decrement and skip if zero
EISZ *	Extended increment and skip if zero
ISZ *	Increment and skip if zero
ISZTS	Increment the doubleword addressed by WSP (skip if zero)
LNDSZ	Narrow decrement and skip if zero
LNISZ	Narrow increment and skip if zero
LWDSZ	Wide decrement and skip if zero
LWISZ	Wide increment and skip if zero
XNDSZ	Narrow decrement and skip if zero
XNISZ	Narrow increment and skip if zero
XWDSZ	Wide decrement and skip if zero
XWISZ	Wide increment and skip if zero

All of the instructions above are atomic (if the wide operand is aligned on a doubleword boundary) with the exception of **DSZTS** *and* **ISZTS**. *Note that a performance increase may be realized (particularly in a multiple-processor configuration) if you use instructions which are not atomic in place of atomic ones, such as* **LWADI** *(Wide Add Immediate) or* **ISZTS** *instead of* **LWISZ**.

* *ECLIPSE compatible instruction*

Carry Operations

For fixed-point arithmetic operations, the processor maintains a carry flag (Carry) that contains a value of 0 or 1. If an instruction adds 16-bit data, any carry occurs from bit 16; if an instruction adds 32-bit data, any carry occurs from bit 0.

The value of Carry can be initialized before a binary operation by executing an explicit carry instruction. Table 2-10 lists the instructions that initialize Carry. The processor retains the value of Carry for use with another instruction.

The processor changes the value of Carry as a result of executing an ECLIPSE MV/Family arithmetic instruction or an ECLIPSE 16-bit compatible fixed-point instruction. For an ECLIPSE MV/Family arithmetic instruction, the processor loads the result of a carry into Carry; it is not relative to its former value (as it is with an ECLIPSE 16-bit compatible instruction). For an ECLIPSE 16-bit compatible instruction, the processor complements Carry during

- Addition, when the most significant bit of each operand and the carry from the adjacent bit produce a carry;
- Subtraction, when borrowing from the most significant bit.

Table 2-10 *Carry initializing instructions*

Instruction	Operation
ADC *	Add complement with optional Carry initialization
AND *	AND with optional Carry initialization
ADD *	Add with optional Carry initialization
COM *	One's complement with optional Carry initialization
CRYTC	Complement Carry
CRYTO	Set Carry to one
CRYTZ	Set Carry to zero
INC *	Increment with optional Carry initialization
MOV *	Move with optional Carry initialization
NEG *	Negate with optional Carry initialization
SUB *	Subtract with optional Carry initialization

* *ECLIPSE compatible instruction*

Shift Instructions

ECLIPSE MV/Family shift instructions operate on either the entire 32 bits of an accumulator (wide shift) or bits 16-31 of an accumulator (ECLIPSE 16-bit instructions).

Wide arithmetic shift instructions (**WASH** and **WASHI**) move 32 bits of an accumulator left or right (0 to 31 bit positions), depending on an 8-bit two's-complement number. The 8 bits in the source accumulator for the **WASH** instruction or the 8 bits in the immediate displacement of the **WASHI** instruction contain the 8-bit number.

- For a left shift, the 8-bit number is positive. The processor shifts from 0 to 31 bit positions and zero-extends the vacated bit positions. A fixed-point overflow occurs if the sign bit changes.

  NOTE: *Shifting a negative value more than 31 bit positions to the left guarantees a fixed-point overflow.*

- For no shifting, the 8-bit number must be 0.

- For a right shift, the 8-bit number must be negative. The processor shifts from 0 to 31 bits and sign-extends the vacated bit positions. The processor drops the bits shifted from the least significant bit position. Negative values shifted right are rounded towards zero.

For instance, when the processor shifts -3 to the right one bit position, the result yields -1; shifting +1 to the right one bit position yields 0. The **WASH** and **WASHI** instructions provide the capability of multiplying or dividing by a power of 2.

The ECLIPSE 16-bit compatible arithmetic and logical instructions (**ADC, ADD, AND, COM, INC, MOV, NEG,** and **SUB**) can shift an intermediate result one bit position or swap the two bytes (Figure 2-2). Accumulator bit 31 is the least significant bit, and bit 16 is the most significant bit. The shift can be

- One bit to the left.

  Carry assumes the state of the most significant bit, and the least significant bit assumes the state of Carry.

- One bit to the right.

  Carry assumes the state of the least significant bit, and the most significant bit assumes the state of Carry.

- A swap of the most significant byte with the least significant byte.

  The processor preserves the state of the Carry flag.

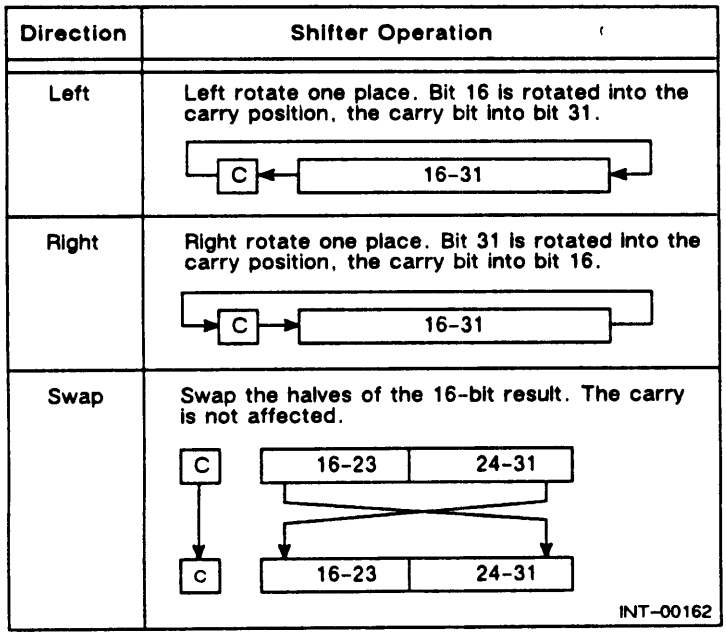


Figure 2-2 *ECLIPSE compatible shift operations*

Skip Instructions

In a skip instruction, the processor tests the result of an operation for a specific condition and directs the processor to skip or execute the word after the skip instruction.

For an instruction that includes a skip option (such as the ECLIPSE 16-bit compatible arithmetic and logical instructions), the processor tests the result during its temporary storage. The processor can then save the result of the computation or ignore it. For an instruction that excludes a skip option (such as **NADD**), the processor stores the result in memory or an accumulator. You can then test the result with an explicit test and skip on condition instruction (such as Skip on OVR Reset — **SNOVR**).

The ECLIPSE 16-bit compatible instructions test for a zero or nonzero value for either or both of the temporary result or Carry. Other instructions compare immediate values to the contents of memory locations or an accumulator, or compare the contents of two accumulators.

Table 2-11 lists the fixed-point skip on condition instructions according to type of operation. When a skip occurs, the processor increments the program counter by one and executes the second word after the skip instruction.

NOTE: *Be sure that a skip does not transfer control to the middle of a 32-bit or longer instruction.*

Table 2-11 *Fixed-point skip on condition instructions*

Instruction	Operation
ECLIPSE 16-bit compatible instructions	
ADC *	Add complement with optional skip
ADD *	Add with optional skip
INC *	Increment with optional skip
MOV *	Move with optional skip
NEG *	Negate with optional Carry initialization
SUB *	Subtract with optional skip
Compare immediate to accumulator	
WSEQI	Wide skip if equal to immediate
WSGTI	Wide skip if AC greater than immediate
WSLEI	Wide skip if AC less than or equal to immediate
WSNEI	Wide skip if AC not equal to immediate
WUGTI	Wide unsigned skip if AC greater than immediate
WULEI	Wide unsigned skip if AC less than or equal to immediate
Compare immediate to memory location	
CLM *	Compare to limits
WCLM	Wide compare to limits and skip
Compare accumulator with accumulator	
SGE *	Skip if ACS greater than or equal to ACD
SGT *	Skip if ACS greater than ACD
WSEQ	Wide skip if ACS equal to ACD
WSGE	Wide signed skip if ACS greater than or equal to ACD
WSGT	Wide signed skip if ACS greater than ACD
WSLE	Wide signed skip if ACS less than or equal to ACD
WSLT	Wide signed skip if ACS less than ACD
WSNE	Wide skip if ACS not equal to ACD
WUSGE	Wide unsigned skip if ACS greater than or equal to ACD
WUSGT	Wide unsigned skip if ACS greater than ACD

* *ECLIPSE compatible instruction*

Overflow Fault

The processor checks for a fixed-point overflow when attempting division or when calculating a fixed-point result. An overflow occurs if the divisor is zero, or if the result is too large to store in memory or in a fixed-point accumulator. At the end of the current instruction cycle, the processor sets the processor status register's overflow flag (OVR) to one. OVR remains set until cleared by another instruction. Refer to the chapter, "Program Flow Management," for information on fault handling.

Processor Status Register

The processor contains a 16-bit processor status register (PSR), which retains information about the status of fixed-point computations. You access the register with instructions that test and set the register contents. Refer to the "Skip Instructions" section for a list of the instructions that test the register contents. Table 2-12 lists the instructions that manipulate the register contents.

Table 2-12 *PSR manipulation instructions*

Instruction	Operation
BKPT	Breakpoint (sets the PSR to 0; tests IXCT upon return from interrupt)
FXTD	Disable fixed-point trap (resets OVK and disables traps)
FXTE	Enable fixed-point trap (sets OVK and enables traps)
LCALL	Call subroutine (sets OVR to 0)
LPSR	Load PSR into AC0
PBX	Pop block and execute (may set IXCT)
SNOVR	Skip on OVR reset
SPSR	Store PSR from AC0
WDPOP	Wide pop context block (restores PSR from context block)
WPOPB	Wide pop block (restores PSR from wide stack)
WRSTR	Wide restore (restores PSR from wide stack)
WRTN	Wide return (restores PSR from wide stack)
WSAVR	Wide save and set OVK to zero
WSAVS	Wide save and set OVK to one
WSSVR	Wide special save and set OVK to zero
WSSVS	Wide special save and set OVK to one
XCALL	Call subroutine (sets OVR to 0)
XVCT	Vector I/O interrupt (initializes PSR with contents of device control table)

Figure 2-3 shows the format of the processor status register; Table 2-13 describes its contents.

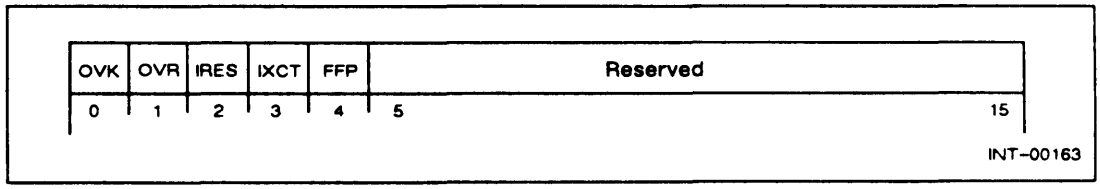


Figure 2-3 *Processor status register format*

NOTES: *The IRES, IXCT and FFP bits are for hardware use. Do not modify the state of these bits; otherwise, results are unpredictable.*

Refer to a machine-specific supplement for information on implemented bits.

Table 2-13 *Processor status register contents*

Bit	Mnemonic	Function
0	OVK	OVK is an overflow mask. To enable fixed-point overflow detection and servicing, set OVK to 1 (with the **FXTE**, **SPSR**, **WSAVS**, and **WSSVS** instructions — see Table 2-12). The processor saves or restores the status of OVK when going to or returning from a subroutine or fault handler. For the processor to detect and service an overflow fault, OVK must be set to 1 before the processor sets OVR to 1.
1	OVR	OVR is an overflow flag. The processor sets OVR to 1 when it detects a fixed-point overflow condition. The processor detects a fixed-point overflow condition when the result exceeds the 16-bit precision (for narrow data instruction) or 32-bit precision (for wide data instruction). The overflow condition (*overflow*) exists for the duration of the fixed-point instruction that causes the overflow. The processor saves the transient *overflow* condition by performing a logical inclusive OR of *overflow* and OVR before completing the instruction. OVR remains set to 1 until any of the following events occur: • An I/O interrupt request is acknowledged. Refer to the chapter, "Device Management," for additional details. • A fault is detected and serviced. Refer to the chapter, "Program Flow," for additional details. • A powerup, I/O reset, or system reset is performed. • The processor executes an instruction listed in Table 2-12.
2	IRES	IRES is an interrupt resume flag. The processor sets IRES when it interrupts a resumable instruction that requires the processor to save its state on the user stack. For example, when the processor interrupts a Wide Edit (WEDIT) instruction, the processor sets IRES and saves the microstate on the user stack. When a resumable instruction begins execution, it first tests IRES. If IRES is 0, the instruction begins an initial execution. If IRES is 1, the instruction restores the state, resets IRES to 0, and resumes execution. NOTE: *Although the processor can interrupt some instructions, implementations may choose to run them through to completion. Refer to a machine-specific supplement for additional information.*
3	IXCT	IXCT is an interrupt-executed opcode flag. When the processor executes the BKPT instruction, it pushes a wide return block onto the current stack. Then, when returning program control, the PBX instruction (located at the end of the breakpoint handler) pops the wide return block and continues the normal program flow with the saved instruction in AC0. If an interrupt occurs while executing the saved instruction (PC points to the BKPT instruction), the processor sets IXCT and pushes the opcode of the saved instruction onto the wide stack. Upon returning from the interrupt handler, the BKPT instruction tests IXCT. If IXCT is set, the BKPT instruction resets IXCT to 0, pops the saved opcode of the interrupted instruction off the wide stack, and executes it.
4	FFP	FFP is the floating-point fault pending flag. This bit contains the state of the Trap Enable (TE) flag of the floating-point status register (FPSR) only if the FPSR error bit (ANY) is also set to indicate a floating-point error. FFP is applicable to systems with floating-point units which are capable of operating in parallel with the CPU. FFP is valid only in the PSR within the context block. Before handling either an interrupt or page fault, the processor must wait for any floating-point instruction executing in a parallel floating-point unit to complete. To guarantee that any floating-point fault is serviced in the proper context, the processor inhibits the floating-point trap until the completion of the page fault or the interrupt service. To accomplish this, the processor sets FFP to reflect the current value of TE in the FPSR. The processor then clears TE (if FFP is set) to inhibit floating-point faults and services the page fault or interrupt. Upon return from the service routine, the processor restores the FPSR TE bit from the PSR FFP and clears FFP. If the restored FPSR TE bit is 1, the processor services any pending floating-point traps after the next instruction boundary is crossed, such as after a WDPOP or WRSTR instruction.

(continued)

Table 2-13 *Processor status register contents (concluded)*

Bit	Mnemonic	Function
5-15	Reserved	The processor sets the reserved bits to zero when storing them in memory. The processor ignores the reserved bits when loading the PSR. CAUTION: *Do not set the PSR bits 5 through 13 to store transient data while they are in memory (such as in a return block); these reserved bits must remain unused.* When stored in memory, bits 14 and 15 are reserved for Data General software.

Logical Operations

The processor performs fixed-point logical arithmetic in the arithmetic logic unit. You control the processor and arithmetic logic unit operations with the move, logic, shift, and skip instructions.

The processor performs the logical functions with **ADC**, **AND**, **COM**, **IOR**, and **XOR** instructions. It can then store the result in memory or can test the result with a skip instruction, which either continues normal program flow or changes it.

Data Formats

Fixed-point logical instructions require the binary data to begin on word boundaries. For instance, an Inclusive OR instruction (**IOR**) performs a logical OR of two 16-bit binary values; a Wide Inclusive OR instruction (**WIOR**) performs a logical OR of two 32-bit binary values. Figure 2-4 shows the 16- and 32-bit formats.

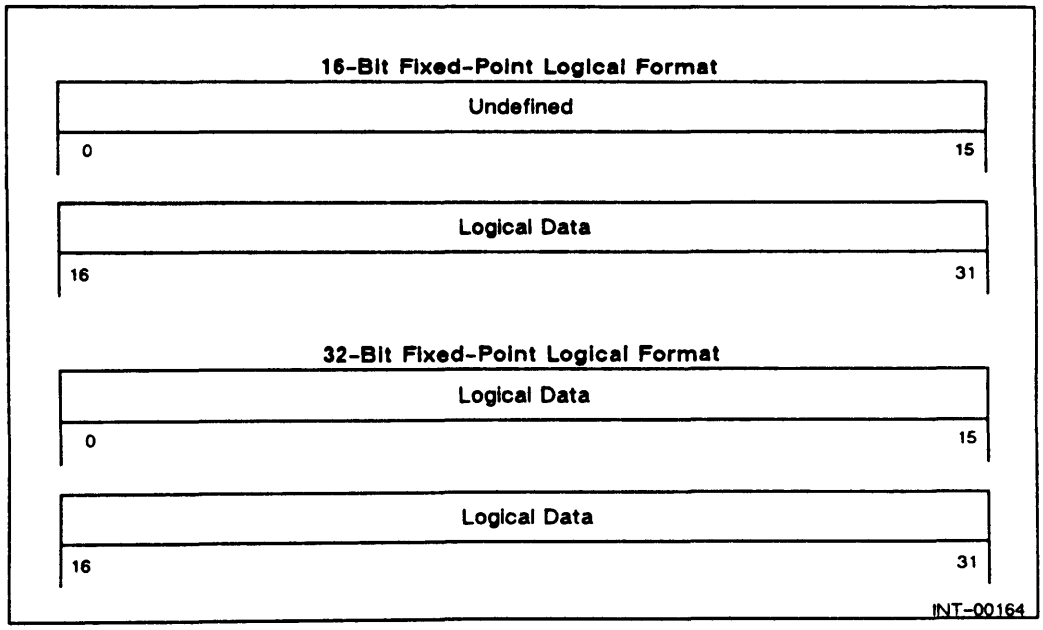


Figure 2-4 *Fixed-point logical data formats*

Logical Instructions

The instructions, listed in Table 2-14, perform logical manipulation on binary data — one's complement, negate, AND, inclusive OR, or exclusive OR. The instructions operate on one or two accumulators, or an accumulator and an immediate value.

Table 2-14 *Logical Instructions*

Instruction	Operation
ANC *	AND with one's complemented source
AND *	AND
ANDI *	AND Immediate
COM *	Complement (one's complement)
IOR *	Inclusive OR
IORI *	Inclusive OR Immediate
NEG *	Negate
NNEG	Narrow Negate
WANC	Wide AND with one's complemented source
WAND	Wide AND
WANDI	Wide AND Immediate
WCOM	Wide complement (one's-complement)
WIOR	Wide Inclusive OR
WIORI	Wide Inclusive OR Immediate
WNEG	Wide Negate
WXOR	Wide exclusive OR
WXORI	Wide exclusive OR Immediate
XOR *	Exclusive OR
XORI	Exclusive OR Immediate

* *ECLIPSE compatible instruction*

Bit Manipulation

The instructions listed in Table 2-15 operate on one or more bits. A wide set bit instruction (**WBTO** and **WBTZ**) requires an effective address in an accumulator. Use a load effective address instruction (**LLEF** or **XLEF**) to calculate and to load the effective address into an accumulator.

Table 2-15 *Bit Instructions*

Instruction	Operation
COB *	Count bits
LOB *	Locate lead bit
LRB *	Locate and reset lead bit
WBTO	Wide set bit to one
WBTZ	Wide set bit to zero
WLOB	Wide locate lead bit
WLRB	Wide locate and reset lead bit

Shift Instructions

Table 2-16 lists the logical shift instructions. The processor can also shift an intermediate result for the **ADC**, **ADD**, **INC**, **MOV**, **NEG** and **SUB** instructions.

Table 2-16 *Logical shift instructions*

Instruction	Operation
DHXL *	Double hex shift left
DHXR *	Double hex shift right
DLSH *	Double logical shift
HXL *	Hex shift left
HXR *	Hex shift right
LSH *	Logical shift
WLSH	Wide logical shift
WLSHI	Wide logical shift immediate
WLSI	Wide logical shift left immediate
WMOVR	Wide move right

* *ECLIPSE compatible instruction*

Skip Instructions

Table 2-17 lists the logical skip on condition instructions. The ECLIPSE 16-bit logical instructions test the result during its temporary storage. Other instructions check the setting of one or more bits either in an accumulator, a memory location, or the processor status register. When a skip occurs, the processor increments the program counter by one, and executes the second word after the skip instruction.

NOTE: *Verify that a skip does not transfer control to the middle of a 32-bit or longer instruction.*

Table 2-17 *Fixed-point logical skip instructions*

Instruction	Operation
ECLIPSE 16-bit instructions)	
ANC *	AND (complemented source) with optional skip
AND *	AND with optional skip
COM *	One's complement with optional skip
NEG *	Negate with optional skip
Bit check (in accumulator or PSR)	
NSALA	Narrow skip on all bits set in accumulator
NSANA	Narrow skip on any bit set in accumulator
WSALA	Wide skip on all bits set in accumulator
WSANA	Wide skip on any bit set in accumulator
WSKBO	Wide skip on AC bit set to one
WSKBZ	Wide skip on AC bit set to zero
SNOVR	Skip on OVR reset
Bit check (in memory location)	
NSALM	Narrow skip on all bits set in memory location
NSANM	Narrow skip on any bit set in memory location
SNB *	Skip on nonzero bit
SZB *	Skip on zero bit
SZBO *	Skip on zero bit and set to one
WSALM	Wide skip on all bits set in doubleword memory location
WSANM	Wide skip on any bit set in doubleword memory location
WSNB	Wide skip on nonzero bit
WSZB	Wide skip on zero bit
WSZBO	Wide skip on zero bit and set bit to one

* *ECLIPSE compatible instruction*

Decimal and Byte Operations

The processor performs decimal arithmetic (packed and unpacked) and 8-bit byte (or ASCII) manipulation. You control the various operations with the move, arithmetic, skip, and shift instructions. The move instructions include the instructions that convert, compare, and insert data.

The decimal arithmetic operations consist of

- Converting and moving decimal numbers between a floating-point accumulator and memory, and translating, scaling, and moving decimal strings between memory locations. The move instructions that convert one data type to another require an explicit data type description.
- Performing floating-point computations on the converted decimal numbers. Refer to the chapter, "Floating-Point Computing," for information on the floating-point arithmetic instructions.

The byte operations consist of

- Moving bytes from one memory location to another.
- Inserting bytes. To insert one or more bytes into a string, move the beginning part of the string to another location. Bytes to be inserted are moved to the other location, and finally, the remainder of the string is moved to the other location.
- Deleting bytes. To delete one or more bytes from a string, move the beginning part of the string to another location. Then, skip the bytes to be deleted, and finally, move the remainder of the string to the other location.
- Converting from one data type to another data type. The move instructions that convert one data type to another require an explicit data type description.
- Comparing one data type to another data type or searching the string for a specific character. The skip instructions include the byte compare instructions even though they do not perform the skip function. A byte compare instruction stores the result of the comparison in an accumulator. Use a skip on condition instruction to test the comparison.

Data Formats

The processor must know the format of the data before accessing it. Most instructions (such as fixed-point and floating-point instructions) imply a data format. However, for packed decimal (BCD) and unpacked decimal (ASCII) arithmetic with certain instructions (such as **WEDIT, WLDI, WDMOV**), the processor requires (in AC0 and/or AC1) an explicit data type indicator, as shown in Figure 2-5 and described in Table 2-18.

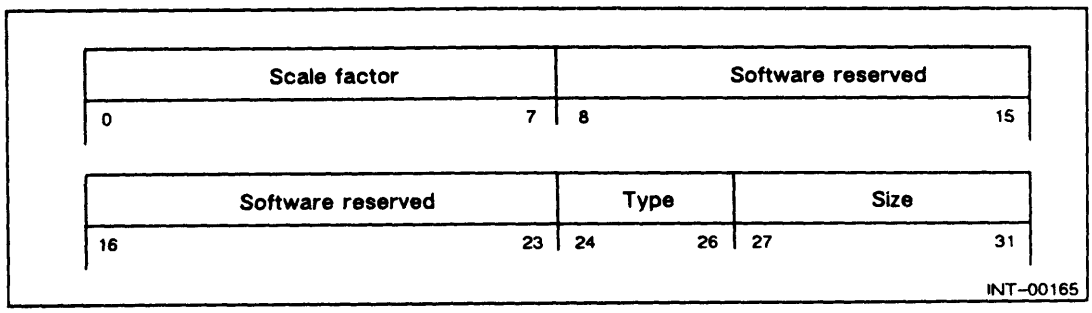


Figure 2-5 *Explicit data type indicator*

Table 2-18 *Data type indicator description*

Mnemonics	Bits	Description
Scale factor	0-7	The scale factor determines how the decimal integer in memory will be interpreted by some instructions. Decimal instructions which do not require this field ignore its contents. The scale factor (sf) is interpreted as an 8-bit, two's complement integer in the range -128 <= sf <= 127. If the decimal integer represented in memory is X, then the "scaled" decimal integer is equal to X * 10^(-sf). For example, if the decimal string in memory represents the number 932, then the "scaled" decimal integer is equal to: 9320 if sf = -1 932 if sf = 0 93.2 if sf = 1 0.00932 if sf = 5
Software reserved	8-23	The reserved field indicates that Data General reserves these bits for future software use.
Type	24-26	The type field identifies the type of data, as listed in Table 2-19.
Size	27-31	The size field is interpreted as a 5-bit, unsigned integer in the range 0 <= size <= 31, and indicates the length of the integer data in memory. For all data types except 5, the size field is one less than the number of bytes of memory occupied by the integer. For data type 5, the size field is equal to the number of digits in the integer. The processor expects an odd number for a size specification. If an even size is specified, the processor adds one to it (to make the size odd) and uses a zero for the most significant digit. Refer to to Table 2-19 for examples.

Decimal strings in memory are either packed or unpacked (see Figure 2-6).

An unpacked decimal digit is the ASCII code for the digit that is represented (see Table 2-21 for valid unpacked digits). An unpacked decimal string consists of a series of unpacked digits and a sign, as follows:

Data types 0 and 1 combine the sign of the integer with one of the decimal digits. This overpunched sign occupies one byte in the integer field in memory, and all other bytes in the integer field consist of unpacked digits.

Data types 2 and 3 require an unpacked sign that occupies a separate byte in the integer field. All other bytes in the integer field consist of unpacked digits. The unpacked sign can be either the ASCII plus sign (+) — 053_8 — or the ASCII minus sign (-) — 055_8.

Refer to Table 2-20 for a list of the sign-positioned ASCII characters. Table 2-21 lists the nonsign-positioned ASCII characters.

A packed decimal string contains two BCD digits per byte (Figure 2-6). The lowest order byte in the decimal string contains the least significant decimal digit packed with the sign of the integer. The packed sign, which occupies four bits, can be either 14_8 or 17_8 (C_{16} or F_{16}) for positive (+); or 15_8 (D_{16}) for negative (-).

Table 2-19 on the following page gives examples of eight data types.

Table 2-19 *Explicit data types*

Data Type	Meaning	Decimal Example	Characters in Each Byte in Memory (Octal) or [Hex]	Data Type Indicator (Octal)
0	*Unpacked decimal*: last byte combines the sign and the last digit	-397	(063) (071) (120) [33] [39] [50]	000002
		+397	(063) (071) (107) [33] [39] [47]	
1	*Unpacked decimal*: first byte combines the sign and the first digit	-397	(114) (071) (067) [40] [39] [37]	000042
		+397	(103) (071) (067) [43] [39] [37]	
2	*Unpacked decimal*: last byte contains the unpacked sign	-397	(063) (071) (067) (055) [33] [39] [37] [2D]	000103
		+397	(063) (071) (067) (053) [33] [39] [37] [2B]	
3	*Unpacked decimal*: first byte contains the unpacked sign	-397	(055) (063) (071) (067) [2D] [33] [39] [37]	000143
		+397	(053) (063) (071) (067) [2B] [33] [39] [37]	
4	*Unpacked decimal*: and unsigned	+397	(063) (071) (067) [33] [39] [37]	000202
5	*Packed decimal*: two BCD digits (or one digit and sign) per byte	-397	(071) (175) [39] [7D]	000243
		+397	(071) (174) [39] [7C]	
6	*Two's complement*: byte-aligned	-397	[FE] [73]	000301
		-397	[FF] [FE] [73]	000302
		+397	[01] [8D]	000301
		+397	[00] [01] [8D]	000302
7	*Floating-point:* byte-aligned	-397	[C3] [18] [D0]	000342
		-397	[C3] [18] [D0] [00]	000343
		+397	[43] [18] [D0]	000342
		+397	[43] [18] [D0] [00]	000343

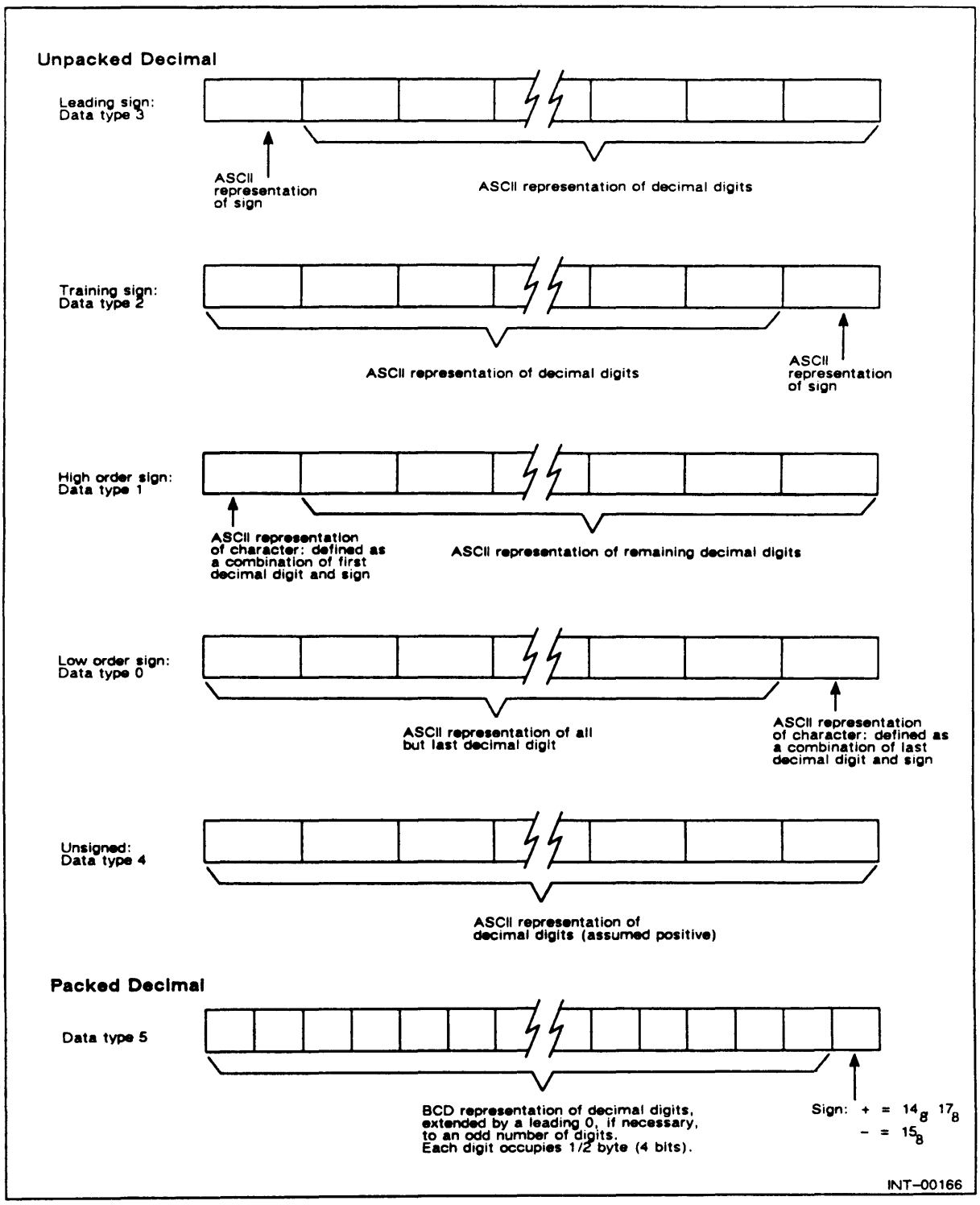


Figure 2-6 *Packed and unpacked decimal data*

Table 2-20 *Sign and number combination for unpacked decimal*

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

NOTE: *Though all four forms of 0+ and both forms of 0- are accepted, brackets ("{" or "}") are always generated.*

Table 2-21 *Nonsign-positioned numbers for unpacked decimal*

Digit	ASCII Character (octal code)		Digit	ASCII Character (octal code)		Digit	ASCII Character (octal code)	
space		(040)	3	3	(063)	7	7	(067)
0	0	(060)	4	4	(064)	8	8	(070)
1	1	(061)	5	5	(065)	9	9	(071)
2	2	(062)	6	6	(066)			

Move Instructions

Move instructions transfer formatted data between memory and a fixed-point accumulator (ac) or floating-point accumulator (fpac) or between two memory locations. In addition to moving data, several instructions also convert, compare, or insert data.

Table 2-22 lists the instructions that move bytes of data. If an instruction loads a byte into the least significant bits of a fixed-point accumulator, the processor zero-extends the remaining bits. If an instruction stores a byte into memory, the processor changes the byte being addressed, but the other byte in the memory word remains intact.

Table 2-22 *Fixed-point byte movement instructions*

Instruction	Operation
LDB *	Load byte
LLDB	Load byte (long displacement)
LSTB	Store byte (long displacement)
STB *	Store byte
WCMT	Wide character move until true
WCMV	Wide character move
WCTR	Wide character translate and compare
WDMOV	Wide decimal move
WEDIT	Convert and insert string of decimal or ASCII characters
WLDB	Wide load byte
WSTB	Wide store byte
XLDB	Load byte
XSTB	Store byte

* *ECLIPSE compatible instructions*

The decimal move and convert instructions (Table 2-23):

- Convert packed decimal dat. ' floating-point format when loading a decimal number into a floating-point accumulator.
- Convert floating-point data to packed decimal format when storing a decimal number in memory.

Table 2-23 *Fixed-point to floating-point conversion and store instructions*

Instruction	Operation
LDI, WLDI	Convert a decimal and load into an fpac
LDIX, WLDIX	Convert a decimal, extend it, and load it into four fpacs
STI, WSTI	Convert fpac data and store into memory
STIX, WSTIX	Convert the four fpacs' data and load into memory

The edit (WEDIT) instruction (with an edit subprogram) converts a decimal integer to a string of bytes, moves a string of bytes, or inserts additional bytes. Table 2-24 lists the edit subprogram instructions (these are all ECLIPSE 16-bit compatible instructions).

Table 2-24 *Edit subprogram instructions*

Instruction	Operation
DADI	Add signed integer to destination indicator
DAPS	Add signed integer to opcode pointer if sign flag is zero
DAPT	Add signed integer to opcode pointer if trigger is one
DAPU	Add signed integer to opcode pointer
DASI	Add signed integer to source indicator
DDTK	Decrement a word in the stack by one and jump if word is nonzero
DEND	End edit subprogram
DICI	Insert characters immediate
DIMC	Insert character *j* times
DINC	Insert character once
DINS	Insert one of two characters depending on sign flag
DINT	Insert one of two characters depending on trigger
DMVA	Move a number of alphabetical characters
DMVC	Move a number of characters
DMVF	Move a number of digits depending on trigger
DMVN	Move a number of numbers
DMVO	Move digit with overpunch
DMVS	Move number with zero suppression
DNDF	End float
DSSO	Set sign flag to one
DSSZ	Set sign flag to zero
DSTK	Store in stack
DSTO	Set trigger to one
DSTZ	Set trigger to zero

Arithmetic Instructions

With the ECLIPSE 16-bit compatible fixed-point add and subtract instructions, the processor computes the sum or difference of two unsigned BCD numbers in bits 28-31 of two accumulators. A carry, if any, is a decimal carry. With the wide decimal instructions, the processor adds one to, or subtracts one from, a decimal string or compares two decimal strings. Table 2-25 lists the arithmetic instructions.

Table 2-25 *Arithmetic instructions*

Instruction	Operation
DAD *	Decimal add
DSB *	Decimal subtract
WDCMP	Wide decimal compare
WDDEC	Wide decimal decrement
WDINC	Wide decimal increment

* *ECLIPSE compatible instruction*

Shift Instructions

With the ECLIPSE 16-bit compatible hex shift instructions, the processor can move decimal results (in bits 16-31 of a fixed-point accumulator) either to the left or to the right. Table 2-26 lists the hex shift instructions.

Table 2-26 *Hex shift instructions*

Instruction	Operation
DHXL *	Double Hex Shift Left
DHXR *	Double Hex Shift Right
HXL *	Hex Shift Left
HXR *	Hex Shift Right

* *ECLIPSE compatible instruction*

Effective Address Instructions

Load effective address instructions (see Table 2-27) calculate a byte or word address that can be used with other instructions to manipulate data. When the processor executes a character manipulation instruction (such as WCMV) with an illegal address, a protection fault occurs.

Table 2-27 *Load effective address instructions*

Instruction	Operation
ELEF *	Extended Load Effective Address
LEF *	Load Effective Address
LLEF	Load Effective Address (Long Displacement)
LLEFB	Load Effective Byte Address (Long Displacement)
LPEF	Push Address (Long Displacement)
LPEFB	Push Byte Address (Long Displacement)
WMOVR	Wide Move Right (convert byte pointer to word pointer)
XLEF	Load Effective Address (Extended Displacement)
XLEFB	Load Effective Byte Address (Extended Displacement)
XPEF	Push Effective Address (Extended Displacement)
XPEFB	Push Effective Byte Address (Extended Displacement)

* *ECLIPSE compatible instruction*

Skip Instructions

A skip instruction normally tests for a condition and then modifies the program counter. When a skip occurs, the processor increments the program counter by one and executes the second word after the skip instruction. However, the Wide Character Compare (WCMP), the Wide Character Translate and Compare (WCTR), and the Wide Load Sign (WLSN) instructions test for a condition, and then load a 0, -1, or +1 into AC1. You can then use a Wide Skip If Accumulator Equal instruction (WSEQ and WSEQI) to test the result.

The Wide Character Scan Until True instruction (WCST) searches a string of bytes for one or more specified characters. When the instruction locates a byte, it stores the byte address in an accumulator.

NOTE: *Be sure that a skip does not transfer control to the middle of a 32-bit or longer instruction.*

Data Type Faults

The processor checks for a valid decimal or ASCII data type and for valid data when executing an instruction that requires an explicit data type description (such as WEDIT, WCTR, WSTI, or WCST). If either the data type or the data is invalid, the processor does not perform the instruction, but instead services the fault before executing another instruction.

Table 2-28 lists the decimal and ASCII fault codes. The first and second columns list the code that appears in AC1. The third column lists the type of return block pushed. The fourth and fifth columns list instructions and conditions that can cause faults.

Refer to the chapter, "Program Flow Management," for more information on fault handling.

Table 2-28 *Decimal and ASCII fault codes*

Code Returned in AC1		Return Block	Faulting	
Narrow	Wide	Type	Instruction	Meaning
000000	100000	2	EDIT, WEDIT	An invalid digit or alphabetic character encountered during execution of one of the following subopcodes: DMVA, DMVF, DMVN, DMVO,DMVS
000001	100001	1	LDIX, STIX	Invalid data type (6 or 7)
		3	EDIT, WEDIT, WLDIX, WSTIX, WDMOV, WDDEC, WDINC, WDCMP	Invalid data type (6 or 7)
000002	100002	2	EDIT, WEDIT	DMVA or DMVC subopcode with source data type 5; AC2 contains the data size and precision
000003	100003	2	EDIT, WEDIT	An invalid opcode; AC2 contains the data size and precision
000004	100004	1	LDI, STI, STIX, WLDI, WSTI, WSTIX, WLDIX	Number too large to convert to specified data type. |number|> (10^{16}) - 1 Number too large to convert to specified data type. Number > (10^{32}) - 1
000005	—	3	EDIT, LDI, LDIX, STI, STIX	Invalid microinterrupt return block (Applies only to ECLIPSE interrupt-resumable instructions)
000006	100006	1	WLSN, WLDI, LSN, LDI, LDIX, WLDIX	Sign code is invalid for this data type *
		3	EDIT, WEDIT,WDINC, WDMOV, WDCMP, WDDEC	
000007	100007	1	WLSN, WLDI, WLDIX, LSN, LDI, LDIX	Invalid digit *
		3	WDMOV, WDCMP, WDINC, WDDEC	

* *A value containing both an invalid sign and one or more invalid digits produces a decimal/ASCII fault which may indicate either type of error.*

Decimal Arithmetic Example

Figure 2-7 illustrates an example of code written for execution under AOS/VS. The program does the following:

1. Accepts the decimal number from a terminal (in ASCII format).
2. Converts it to single-precision floating-point format.
3. Performs the floating-point addition.
4. Converts the sum to ASCII format.
5. Displays it on the terminal.

```
          .TITL     DECIMAL
          .ENT      START
          .NREL
          :CONSTANTS
          .ENABLE SWORD

    CON:  .TXT      ``@CONSOLE''              ;Generic console name
   FCON:  202                                 ;Type 4 and 3 decimal digits.
   IBUF:  .BLK 5                              ;Reserve 5 words for number buffer.

          :PARAMETER PACKETS

  :READ CONSOLE PACKET TO OPEN. READ. & WRITE

CONSOLE:  .BLK  22
          .LOC  CONSOLE+?ISTI
          ?RTDS+OFIO                          ;Data sensitive I/O.
          .LOC  CONSOLE+?ISTO
          0
          .LOC  CONSOLE+?IMRS
          -1
          .LOC  CONSOLE+?IBAD                 .?IBAD contains byte pointer to data packet.
          IBUF*2
          .LOC  CONSOLE+?IRCL
          -1
          .LOC  CONSOLE+?IFNP
          CON*2
          .LOC  CONSOLE+?IRNW
          0
          .ENABLE DWORD
          .LOC  CONSOLE+?IDEL
          -1
          .LOC  CONSOLE+?ETSP
          0
          .LOC  CONSOLE+?ETFT
          0
          .LOC  CONSOLE+?ETLT
          0

:END OF CONSOLE PACKET

  START:  ?OPEN  CONSOLE                      ;Open console to read and write.
          .
          .
          ?READ  CONSOLE                      ;Accept a number from the keyboard.
          .
          .
          XNLDA  1.FCON                       ;Initialize for data type 4.
          XNLDA  3.CONSOLE+?IBAD              ;Get byte pointer from console packet.
          WLDI   0
          FAS    0,0                          ;Single-precision floating-point add.
          XNLDA  1.FCON
          XNLDA  3.CONSOLE+?IBAD

  AGAIN:  WSTI   0
          INC    1,1,SZC                      ;Increment byte count and skip
                                              ;if WSTI truncates.
          WBR    AGAIN                        ;Repeat WSTI.
          ?WRITE  CONSOLE                     ;Display the sum on the console.
          .
          .
          ?CLOSE  CONSOLE                     ;Close the console.
          .
          .
          ?RETURN                             ;Return to CLI.
          .
          .
          .END START                                                    INT-00167
```

Figure 2-7 *Decimal arithmetic example*

End of Chapter

3

Floating-Point Computing

Using floating-point computation, the processor can add, subtract, multiply, and divide 32-bit (single-precision) and 64-bit (double-precision) sign magnitude data.

The optional Intrinsic Instruction Set (IIS), allows the processor to perform trigonometric and logarithmic functions, exponentiation, and square root evaluation on 32-bit and 64-bit data.

Following a computation, the processor can convert a double-precision value to a single-precision value, or it can convert a single-precision value to a fixed-point or decimal value. Then, the processor can test and skip on a condition that results from the computation or conversion. Finally, the processor can store the result in an accumulator or memory.

This chapter explains the various computations (convert, move, arithmetic, and skip), the Intrinsic Instruction Set, and the floating-point status register (FPSR).

Data Formats

All floating-point instructions require normalized floating-point values in order to produce valid results. (Exceptions to this rule are given in the instruction descriptions, such as the Floating-Point Normalize (**FNOM**) instruction, the Read High Word (**FRH**) instruction, and any instructions which load or store floating-point values or push or pop values using the stack.) Floating-point arithmetic and intrinsic instructions require normalized, sign magnitude numbers. You can use the Floating-Point Normalize instruction to normalize raw floating-point data, which may or may not already be normalized.

In addition, if a mantissa equals zero, the processor expects it to equal true zero. A *true zero* value exists when the sign bit, exponent, and mantissa equal zero; that is, all bits equal zero.

The processor operates most efficiently when single-precision numbers are on even-word boundaries, and double-precision numbers are on even doubleword boundaries. These numbers must be within the value range of $5.4(10^{-78})$ to $7.2(10^{75})$. Figure 3-1 shows the floating-point formats; Table 3-1 describes the formats.

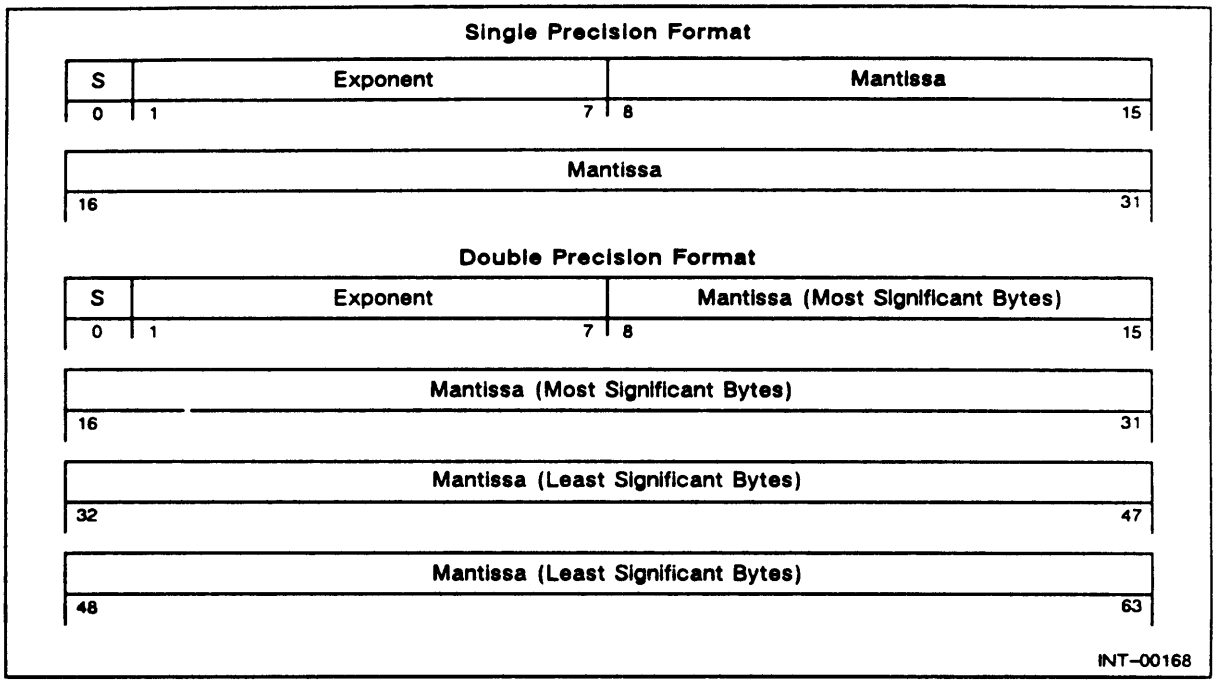


Figure 3-1 *Floating-point data formats*

Table 3-1 *Floating-point data formats description*

Name	Contents
S	The S bit is the sign bit of the mantissa. The sign bit equals 0 for a positive number, and equals 1 for a negative number.
Exponent	The exponent, expressed as an unsigned integer, equals 64_{10} greater than the true value of the exponent (excess-64 representation). The following exponents illustrate excess-64 representation numbers. Exponent — True Value of Exponent 0 — -64_{10} 64_{10} — 0 127_{10} — $+63_{10}$
Mantissa	The mantissa, expressed as a fraction, implies that the location of the binary point is between bits 7 and 8. For normalized floating-point numbers, the range of the mantissa for single-precision is 1/16 to $1-(2^{-24})$ the range for double-precision is 1/16 to $1-(2^{-56})$

Conversion Instructions

Floating-point conversion instructions change various values to floating-point representation. Table 3-2 lists the instructions that convert and move data between fixed-point and floating-point accumulators, convert a mixed number to a fraction, and scale a floating-point number.

Table 3-2 *Floating-point binary conversion instructions*

Instruction	Operation
FEXP *	Load exponent (AC0 17-23 to fpac 1-7)
FAB *	Compute absolute value (set sign of fpac to zero)
FFAS *	Fix to AC (fpac to ac)
FFMD *	Fix to memory
FINT *	Integerize (fpac)
FLAS *	Float from AC (ac to fpac)
FNEG *	Negate
FNOM *	Normalize (fpac)
FRDS	Floating-point round double to single
FRH *	Read high word (fpac 0-15 to AC0 16-31)
FSCAL *	Scale floating-point
WFFAD	Wide fix from fpac
WFLAD	Wide float from AC

* *ECLIPSE compatible instruction*

Table 3-3 lists the instructions that convert and move a fixed-point decimal between memory and a floating-point accumulator. Refer to the chapter, "Fixed-Point Computing," for further information on the load and store integer instructions.

Table 3-3 *Floating-point decimal conversion instructions*

Instruction	Operation
LDI, WLDI	Convert a decimal and load it into an fpac
LDIX, WLDIX	Convert a decimal, extend it, and load it into four fpacs
STI, WSTI	Convert fpac data and store it into memory
STIX, WSTIX	Convert the data in four fpacs and store them into memory

Move Instructions

All single-precision operations that specify an accumulator operate with the most significant 32 bits of the floating-point accumulator (bits 0-31) and ignore the least significant 32 bits (bits 32-63). Upon completion of the specified operation, the processor returns the result to the most significant portion of the floating-point accumulator. The processor loads the least significant 32 bits of the floating-point accumulator with zeros. Table 3-4 lists the instructions which load and store values between the floating-point accumulators and memory or between accumulators.

Table 3-4 *Floating-point data movement instructions*

Instruction	Operation
FLDD *	Load Floating-Point Double
FLDS *	Load Floating-Point Single
FMOV *	Move Floating-Point (FPAC to FPAC)
FPSH	Narrow Floating-Point Push
FPOP	Narrow Floating-Point Pop
FSTD *	Store Floating-Point Double
FSTS *	Store Floating-Point Single
LFLDD	Load Floating-Point Double (Long Displacement)
LFLDS	Load Floating-Point Single (Long Displacement)
LFSTD	Store Floating-Point Double (Long Displacement)
LFSTS	Store Floating-Point Single (Long Displacement)
WFPOP	Wide Floating-Point Pop
WFPSH	Wide Floating-Point Push
XFLDD	Load Floating-Point Double (Extended Displacement)
XFLDS	Load Floating-Point Single (Extended Displacement)
XFSTD	Store Floating-Point Double (Extended Displacement)
XFSTS	Store Floating-Point Single (Extended Displacement)

* *ECLIPSE compatible instruction*

Floating-Point Arithmetic Operations

To perform a floating-point arithmetic operation, the processor executes a floating-point arithmetic instruction. In executing the instruction, the processor

1. Appends guard digits.
2. Aligns the mantissas (for addition and subtraction).
3. Calculates and normalizes the result.
4. Adjusts the result by truncating or rounding it.
5. Stores the result in a floating-point accumulator or memory.

Appending Guard Digits

To increase the accuracy of the result, the processor appends guard digits to the mantissa of one operand before performing the arithmetic calculations. A *guard digit* is one hex digit (four bits) that initially contains zero. The processor modifies the guard digits during the arithmetic calculations, which increases the accuracy of the result.

The processor appends one or two guard digits to the least significant hex digit of a mantissa, depending on the RND flag (bit 8) in the floating-point status register. Use the Load Floating-Point Status Register instruction (**LFLST**) to change the RND flag.

- When the RND flag equals zero, the processor appends one guard digit in preparation for truncating the mantissa of the intermediate result.
- When the RND flag equals one, the processor appends two guard digits in preparation for rounding the mantissa of the intermediate result.

An *intermediate result* includes the exponent and the mantissa.

NOTE: *The floating-point conversion and single-precision store instructions (***FINT, FSCAL, LFSTS, WFFAD, WFLAD,** *and* **XFSTS***) ignore the RND flag.*

Aligning the Mantissas

For floating-point addition and subtraction, the processor first aligns the smaller mantissa to the larger mantissa. To align the mantissas, the processor takes the absolute value of the difference between the two exponents. If the difference equals nonzero, the processor shifts the mantissa with the smaller exponent to the right until the difference equals zero or until the processor shifts out the significant digits of the mantissa. The mantissas are aligned when the difference equals zero.

If the processor shifts out the significant digits, the operation is equivalent to adding zero to the number with the larger exponent. To shift out the significant digits, the processor must shift at least 7 or 8 hex digits for single-precision (for truncating or rounding, respectively) or shift at least 15 or 16 hex digits for double-precision.

Calculating and Normalizing the Result

The processor performs the floating-point arithmetic operation, uses algebraic rules to determine the signs of the intermediate result, and then normalizes it. The processor normalizes an intermediate mantissa by shifting it left one hex digit at a time until the most significant hex digit represents a nonzero quantity. For each hex digit shifted left, the processor decrements the intermediate exponent by one. The processor zero fills the guard digit of the intermediate mantissa.

Truncating or Rounding the Result

As determined by the RND flag, the processor truncates or rounds the intermediate mantissa.

- When the RND flag equals zero, the processor truncates the intermediate mantissa by removing the guard digit.
- When the RND flag equals one, the processor rounds the intermediate mantissa by removing and analyzing the two guard digits.

  When the two guard digits are

  - Within the range of 0 to $7F_{16}$ inclusive, the intermediate result becomes the final result (without change).
  - Equal to 80_{16}, the processor adds the least significant bit of the intermediate mantissa to the intermediate mantissa.

    The processor forces an even mantissa to be rounded down to the nearest integer and an odd mantissa to be rounded up to the nearest integer. If the processor rounded down or rounded up without an intermediate mantissa overflow, the operation produces the final result.
  - Within the range of 81_{16} to FF_{16} inclusive, the processor adds 1 to the intermediate mantissa.

  If the processor rounded up the intermediate mantissa without an overflow, the operation produces the final result.

  If rounding up causes a mantissa overflow, the processor performs the following actions:

  1. Shifts the intermediate mantissa right one hex digit.
  2. Places 1_{16} into the most significant hex digit.
  3. Adds 1_{16} to the intermediate exponent.
  4. Truncates the rightmost hex digit so that the intermediate mantissa is 24 or 56 bits, which becomes the final result.

Storing the Result

The processor stores the final result in the specified memory location or floating-point accumulator. The processor then checks for a possible exponent underflow or overflow. If no underflow or overflow exists, the instruction execution is complete. If an underflow or overflow exists, the processor sets the appropriate error flag in the floating-point status register. The value of the exponent is undefined.

Arithmetic Instructions

Floating-point arithmetic instructions perform single- and double-precision addition, subtraction, multiplication, and division. Unnormalized floating-point numbers may produce undefined results (use **FNOM** to normalize floating-point numbers).

Addition

The processor adds the two mantissas together, producing an intermediate result. The processor determines the sign of the intermediate result from the signs of the two operands by the rules of algebra.

If the mantissa addition produces a carry from the most significant bit, the processor shifts the intermediate mantissa to the right one hex digit and increments the exponent by one.

- If incrementing the exponent produces no exponent overflow and the intermediate mantissa equals a nonzero, the processor normalizes the intermediate mantissa, rounds or truncates it, and stores the final result in memory or in a floating-point accumulator.
- If incrementing the exponent produces an exponent overflow, the processor sets the OVF error flag in the FPSR to one and terminates the instruction.

If there is no mantissa overflow, but the intermediate mantissa contains all zeros, the processor places a true zero in memory or in a floating-point accumulator.

Table 3-5 lists the floating-point add instructions.

Table 3-5 *Floating-point addition instructions*

Instruction	Operation
FAD *	Add Double (FPAC to FPAC)
FAS *	Add Single (FPAC to FPAC)
FAMD *	Add Double (Memory to FPAC)
FAMS *	Add Single (Memory to FPAC)
LFAMD	Add Double (Memory to FPAC) (Long Displacement)
LFAMS	Add Single (Memory to FPAC) (Long Displacement)
XFAMD	Add Double (Memory to FPAC) (Extended Displacement)
XFAMS	Add Single (Memory to FPAC) (Extended Displacement)

* *ECLIPSE compatible instruction*

Subtraction

For floating-point subtraction, the processor temporarily complements the sign of the source mantissa and performs a floating-point addition. Upon completion, the difference is stored in the destination floating-point accumulator. Also the source mantissa returns to its original value when the source accumulator (fpacs) is different from the destination accumulator (fpacd). Table 3-6 lists the floating-point subtract instructions.

Table 3-6 *Floating-point subtraction instructions*

Instruction	Operation
FSD *	Subtract Double (FPAC from FPAC)
FSS *	Subtract Single (FPAC from FPAC)
FSMD *	Subtract Double (Memory from FPAC)
FSMS *	Subtract Single (Memory from FPAC)
LFSMD	Subtract Double (Memory from FPAC) (Long Displacement)
LFSMS	Subtract Single (Memory from FPAC) (Long Displacement)
XFSMD	Subtract Double (Memory from FPAC) (Extended Displacement)
XFSMS	Subtract Single (Memory from FPAC) (Extended Displacement)

* *ECLIPSE compatible instruction*

Multiplication

For floating-point multiplication, the processor multiplies one floating-point mantissa by the other floating-point mantissa to produce an intermediate floating-point mantissa. The processor adds the two exponents, subtracts 64_{10} to maintain excess 64 notation, and produces an intermediate floating-point exponent. The processor then normalizes the intermediate mantissa, rounds or truncates it, and stores the final result. Table 3-7 lists the floating-point multiplication instructions.

Table 3-7 *Floating-point multiplication instructions*

Instruction	Operation
FMD *	Multiply Double (FPAC by FPAC)
FMS *	Multiply Single (FPAC by FPAC)
FMMD *	Multiply Double (FPAC by Memory)
FMMS *	Multiply Single (FPAC by Memory)
LFMMD	Multiply Double (FPAC by Memory) (Long Displacement)
LFMMS	Multiply Single (FPAC by Memory) (Long Displacement)
XFMMD	Multiply Double (FPAC by Memory) (Extended Displacement)
XFMMS	Multiply Single (FPAC by Memory) (Extended Displacement)

* *ECLIPSE compatible instruction*

Division

For floating-point division, the processor tests the divisor for zero. (The source location contains the divisor and the destination location contains the dividend.) If the divisor is zero, the processor sets the INV error flag in the FPSR to one, places error code zero in the INP bits and the address of the instruction in the FPPC, and ends the instruction. If the divisor is nonzero, the processor compares the two mantissas. If the dividend mantissa is greater than or equal to the divisor mantissa, the processor aligns the two mantissas in the following process:

1. Shifts the dividend mantissa to the right one hex digit.
2. Places 0_{16} in the most significant digit of the dividend mantissa.
3. Adds 1_{16} to the dividend exponent.

When the dividend mantissa is less than the divisor mantissa, the processor performs the following actions:

1. Divides the mantissas to produce an intermediate floating-point mantissa.
2. Subtracts the divisor exponent from the dividend exponent, and adds 64_{10} to the difference (maintaining the excess 64 notation), which produces an intermediate floating-point exponent.
3. Normalizes and rounds or truncates the intermediate mantissa, which produces the final result (exponent and mantissa).
4. Stores the final result in memory or a floating-point accumulator.

Table 3-8 lists the floating-point divide instructions.

Table 3-8 *Floating-point division instructions*

Instruction	Operation
FDD *	Divide Double (FPAC by FPAC)
FDS *	Divide Single (FPAC by FPAC)
FDMD *	Divide Double (FPAC by Memory)
FDMS *	Divide Single (FPAC by Memory)
FHLV *	Halve (fpac/2)
LFDMD	Divide Double (FPAC by Memory) (Long Displacement)
LFDMS	Divide Single (FPAC by Memory) (Long Displacement)
XFDMD	Divide Double (FPAC by Memory) (Extended Displacement)
XFDMS	Divide Single (FPAC by Memory) (Extended Displacement)

* *ECLIPSE compatible instruction*

Skip Instructions

A skip instruction tests the result of an operation for a specific condition by checking a bit or combination of bits in the FPSR. These instructions then direct the processor either to skip the word or to execute the word after the skip instruction. (The **FCMP** instruction compares two floating-point numbers and sets the Z and N status flags in the FPSR reflecting the relationship. You can then use a skip instruction to test the status flags.)

Table 3-9 lists the floating-point skip on condition instructions. When a skip occurs, the processor increments the program counter by one and executes the second word after the skip instruction.

NOTE: *Be sure that a skip does not transfer control to the middle of a 32-bit or larger instruction.*

Table 3-9 *Floating-point skip on condition instructions*

Instruction	Operation
FCMP *	Compare two floating-point numbers (set N and Z)
FSEQ *	Skip on zero (Z = 1)
FSGE *	Skip on greater than or equal to zero (N = 0)
FSGT *	Skip on greater than zero (N and Z = 0)
FSLE *	Skip on less than or equal to zero (N or Z = 1)
FSLT *	Skip on less than zero (N = 1)
FSND *	Skip on no zero divide (INV = 0)
FSNE *	Skip on nonzero (Z = 0)
FSNER *	Skip on no error (ANY = 0)
FSNM *	Skip on no mantissa overflow (MOF = 0)
FSNO *	Skip on no overflow (OVF = 0)
FSNOD *	Skip on no overflow and no zero divide (OVF and INV = 0)
FSNU *	Skip on no underflow (UNF = 0)
FSNUD *	Skip on no underflow and no zero divide (UNF and INV = 0)
FSNUO *	Skip on no underflow and no overflow (UNF and OVF = 0)

* *ECLIPSE compatible instruction*

Intrinsic Instruction Set

The optional Intrinsic Instruction Set (IIS) performs trigonometric and logarithmic functions, exponentiation, and square root evaluation on single-precision and double-precision data.

These instructions assume the single argument to be operated on is in FPAC0 and the answer will be returned to FPAC0. For instructions which require two arguments (**WFATN2D**, **WFATN2S**, **WFPWRD**, and **WFPWRS**), FPAC1 contains the second argument. When the instruction completes, the contents of the remaining floating-point accumulators are undefined.

All floating-point inputs are assumed to be normalized. Any input with a zero mantissa is also assumed to have a zero sign bit and an all zero exponent (true zero).

The trigonometric instructions (sine, cosine, tangent) require a floating-point input in radians while the inverse trigonometric instructions (arcsine, arccosine, arctangent) return the result in radians.

IIS instructions always update the Z and N flags of the FPSR so that they reflect the result — either zero or negative.

If traps are enabled, an IIS instruction can cause a floating-point trap for either invalid input or for a result that overflows or underflows.

- Invalid input. If an invalid normalized number or an illegal argument is used as input to an IIS instruction, a trap occurs. An illegal argument causes the processor to place the instruction address in the FPSR floating-point program counter (FPPC), set the INV bit in the FPSR to one, and place an error code in FPSR bits 28–31 (INP). The error code indicates what type of input error occurred. For example, a negative value input to a square root instruction causes an invalid input argument trap with the error code 2 returned to the INP bits. The FPSR description in the section, "Faults and Status," of this chapter explains the various codes and their meanings.
- Overflow or underflow. If the result of an IIS instruction has overflowed or underflowed, a floating-point trap occurs with the relevant error bits (OVF or UNF) updated in the FPSR. Overflow and underflow errors behave identically to the standard floating-point instruction errors.

The four-word format for all IIS instructions is shown in Figure 3–2; Table 3–10 describes the format.

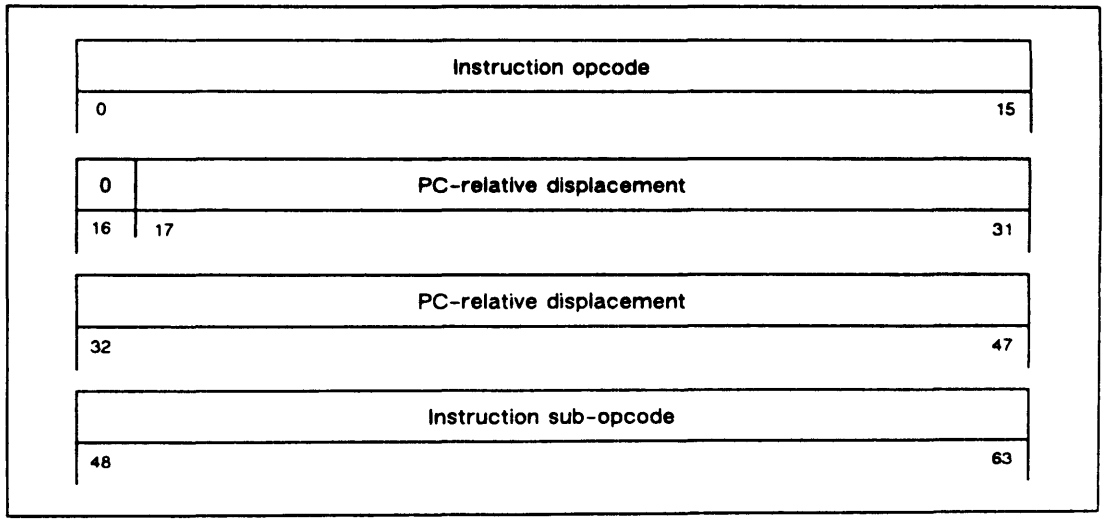


Figure 3–2 *Intrinsic instruction set format*

Table 3-10 *Intrinsic instruction set format description*

Name	Bits	Contents
Instruction opcode	0-15	This identifies an instruction as belonging to the Intrinsic Instruction Set. For all IIS instructions, this value is 107171_8.
PC-relative displacement	16-47	Each IIS instruction has a displacement coded with it. Machines that implement these instructions in hardware ignore the displacement. (Addresses must resolve to a valid word in the current ring.) The coded displacement is a PC-relative, nonindirectable address of a routine in an optional runtime library in the current ring. The routine emulates the function of the instruction if hardware support does not exist. Note that bit 16 is always 0.
Instruction sub-opcode	48-63	This field identifies the specific IIS instruction. The descriptions in the *Instruction Dictionary* give the sub-opcodes for all IIS instructions.

All IIS instructions are interruptible. Table 3-11 lists the intrinsic instructions.

Table 3-11 *Floating-point intrinsic instructions*

Instruction	Function
WFACOSD	Arccosine Double
WFACOSS	Arccosine Single
WFASIND	Arcsine Double
WFASINS	Arcsine Single
WFATAND	Arctangent Double
WFATANS	Arctangent Single
WFATN2D	Arctangent Double (Two-Accumulator)
WFATN2S	Arctangent Single (Two-Accumulator)
WFCOSD	Cosine Double
WFCOSS	Cosine Single
WFEXPD	Exponential Double
WFEXPS	Exponential Single
WFLG2D	Binary Logarithm Double
WFLG2S	Binary Logarithm Single
WFLNGD	Natural Logarithm Double
WFLNGS	Natural Logarithm Single
WFLOGD	Common Logarithm Double
WFLOGS	Common Logarithm Single
WFPWRD	Power Double (Two-Accumulator)
WFPWRS	Power Single (Two-Accumulator)
WFSIND	Sine Double
WFSINS	Sine Single
WFSQRD	Square Root Double
WFSQRS	Square Root Single
WFTAND	Tangent Double
WFTANS	Tangent Single

Faults and Status

The processor checks for a floating-point fault, mantissa status, or an exception condition (such as an overflow or underflow) during or immediately after executing a floating-point instruction. The processor stores the result of this check in the 64-bit floating-point status register (FPSR). When the processor detects a floating-point fault, it sets the appropriate FPSR bits. These bits are cumulative and remain set until they are changed by another instruction.

The processor cannot service the fault unless it first determines the state of the trap enable (TE) mask (bit 5 of the FPSR). If TE equals

- 0 — the processor continues normal program execution with the next sequential instruction. Program flow remains unchanged.

- 1 — the processor disrupts normal program execution by performing an indirect jump to the floating-point fault handler to service the fault. Refer to the chapter, "Program Flow Management," for further information on floating-point fault handling.

NOTE: *The* **FSST**, **LFSST**, **FPSH**, *and* **WFPSH** *instructions store the contents of the FPSR. However, they will not store an FPSR value with any combination of bit 5 and bits 1 through 4 concurrently set.*

The FPSR is accessed with instructions that initialize the register or that test the register's bits. The section, "Skip Instructions," in this chapter lists instructions that test the bits. Table 3-12 lists the instructions that initialize the register and that store or load the register contents.

Table 3-12 *Floating-point status register instructions*

Instruction	Operation
FCLE *	Clear errors (FPSR)
FLST *	Load FPSR
FPOP *	Narrow floating-point pop
FPSH *	Narrow floating-point push
FSST *	Store FPSR
FTD *	Floating-point trap disable (resets TE)
FTE *	Floating-point trap enable (sets TE)
LFLST	Load FPSR (Long Displacement)
LFSST	Store FPSR (Long Displacement)
WFPSH	Wide push floating-point state
WFPOP	Wide pop floating-point state

* *ECLIPSE compatible instruction*

The floating-point status register contains flags indicating a fault (ANY, OVF, UNF, INV, MOF), enabling fault detection (TE), the mantissa status (Z and N), result rounding or truncating (RND), the floating-point identification (ID), the floating-point unit operation (PAR), an invalid input argument indicator (INP), and the floating-point program counter (FPPC). Figure 3-3 shows the format of the FPSR; Table 3-13 describes the FPSR bits.

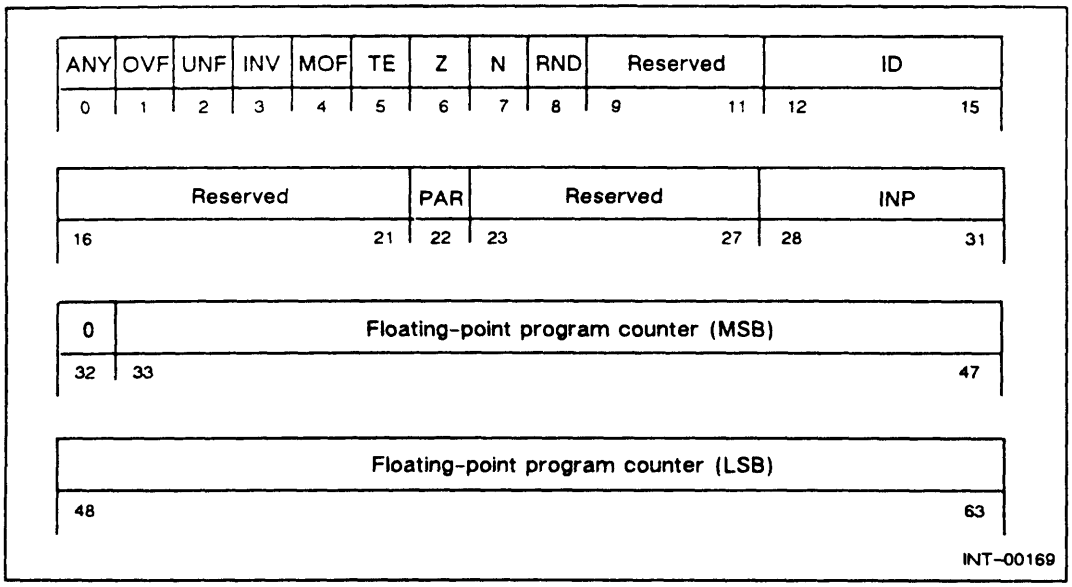


Figure 3-3 *Floating-point status register format*

Table 3-13 *Floating-point status register format description*

Name	Bits	Contents or Function
ANY	0	Error status flag. The processor sets ANY to 1 when it sets either OVF, UNF, INV, or MOF to 1. ANY is never set from memory, but is always recomputed from these bits.
OVF	1	Exponent overflow flag. The processor sets OVF while executing a floating-point instruction and an exponent overflow occurs. The result is correct except the exponent is 128_{10} too small.
UNF	2	Exponent underflow flag. The processor sets UNF while executing a floating-point instruction and an exponent underflow occurs. The result is correct except the exponent is 128_{10} too large.
INV	3	Input argument error flag. The processor sets INV while attempting to execute an instruction with an invalid argument as input. If INV is 1, INP further defines the input error. The processor then aborts the operation, the state of FPAC0 is undetermined, and the remaining operands are unchanged. NOTE: *The previous definition of this flag, Divide by Zero (DVZ), has been expanded to include other input argument errors.*
MOF	4	Mantissa overflow flag. The processor sets MOF while executing a floating-point instruction when it detects a mantissa overflow. If it occurs during a FSCAL instruction, the processor shifts out the most significant bit. If it occurs during an FFAS, FFMD, or WFFAD instruction, the result is too large and the processor truncates the result before storing it.

(continues)

Table 3-13 *Floating-point status register format description*

Name	Bits	Contents or Function
TE	5	Trap enable mask. The processor or you enable floating-point fault detection and servicing by setting TE to 1, and disable floating-point detection and servicing by setting TE to 0. TE can be set to 1 with the FTE instruction, and set to 0 with the FTD instruction. Unless your system runs with a parallel floating-point unit, the processor does not save or restore the status of TE when going to or returning from a subroutine or fault handler. Refer to the "Processor Status Register" description in the chapter, "Fixed-Point Computing," for further information. The processor cannot detect and service a fault unless TE is set to 1 before the processor sets ANY to 1. If TE is set to 1, a 1 in any of bits 1 through 4 results in a floating-point trap, except where noted.
Z	6	True zero flag. The processor sets Z if the result of executing a floating-point instruction produces a true zero.
N	7	Negative flag. The processor sets N if the result of executing a floating-point instruction produces a value less than 0.
RND	8	Round flag. You set RND (with the LFLST, FLST, WFPOP, and FPOP instructions) to direct the processor to round (RND = 1) or to truncate (RND = 0) the intermediate result when executing a floating-point instruction.
Reserved	9-11	These bits are ignored and returned as zeros.
ID	12-15	Floating-point identification code that reflects the floating-point revision.
Reserved	16-21	These bits are ignored and returned as zeros.
PAR	22	Floating-point operation flag. This bit is applicable only to processors which support a floating-point unit capable of operating in both serial and parallel with the main CPU. If PAR is 1, the floating-point unit operation is serial. If PAR is 0, the floating-point unit operation is parallel.
Reserved	23-27	These bits are ignored and returned as zeros.
INP	28-31	These bits contain an indicator of an invalid input argument. The definition of INP depends on the setting of the invalid input argument (INV) bit. If INV is 0, INP is undefined. If INV is 1, the value contained in INP indicates an attempt to use an invalid input. A code value greater than zero applies to the floating-point Intrinsic Instruction Set (IIS) option. The currently defined values are listed below.

(continues)

Table 3-13 *Floating-point status register format description (concluded)*

Name	Bits	Contents or Function		
INP	28-31	Code (octal)	Instruction	Description
		0	FDS, FDD, FDMS, FDMD, XFDMS, XFDMD, LFDMS, LFDMD	Attempt to execute a floating-point divide instruction with a divisor equal to 0.
		NOTE: *When the processor detects a 0 divisor input, the contents of the remaining floating-point accumulators are unmodified, Carry is unchanged, and* overflow *is unaffected.*		
		1	WFLOGS, WFLOGD WFLG2S, WFLG2D WFLNGS, WFLNGD	FPAC0 contains a value less than or equal to 0.
		2	WFSQRS, WFSQRD	FPAC0 contains a value less than 0.
		3	WFASINS, WFASIND WFACOSS, WFACOSD	The absolute value of FPAC0 is greater than 1.
		4	WFPWRS, WFPWRD	The value in FPAC0 is less than 0 and the value in FPAC1 is not equal to 0, or the value in FPAC0 is equal to 0, and the value in FPAC1 is less than or equal to 0.
		5	WFEXPS, WFEXPD	The number in FPAC0 will cause an overflow.
			WFPWRS, WFPWRD	The numbers in FPAC0 and FPAC1 will cause an overflow.
		6	WFTANS, WFTAND	The number in FPAC0, which is an odd integer multiple of values near PI/2, will cause an overflow. (Even integer multiples of PI/2 return 0)
		7	WFATN2S, WFATN2D	The numbers in both FPAC0 and FPAC1 equal 0.
		10	WFEXPD, WFEXPS	The number in FPAC0 will cause an underflow.
			WFPWRD, WFPWRS	The numbers in either FPAC0 or FPAC1 will cause an underflow.
		NOTES: *If floating-point traps are disabled (TE equals 0), more than one invalid input argument error may occur before floating-point traps are again enabled. In this case, INP will contain the error code for the FIRST instruction which caused an invalid input argument error.* *When the processor detects an invalid input error while executing an IIS instruction, the state of all floating-point accumulators is undefined, Carry is unchanged, and* overflow *is unaffected.*		
0	32	This bit must be zero (processor specific).		
Floating-Point Program Counter	33-63	The floating-point program counter (FPPC) contains the address of the first floating-point instruction to set an error bit in the FPSR (after an FCLE or IORST instruction) unless specifically set by a Load Floating-Point Status instruction (LFLST). FPPC is undefined if ANY=0.		

NOTES: *All reserved bits in the FPSR must be 0. Instructions which load the FPSR from memory (such as* **LFLST** *or* **WFPOP***) must load "reserved" bits as zeros otherwise the results are unpredictable.*

The OVF, UNF, INV, and MOF status bits are cumulative. These bits are only cleared by FPSR loads which restore them, or by an instruction which explicitly clears them (such as **FCLE***).*

End of Chapter

4

Stack Management

A *stack* is a series of consecutive locations in memory. In the simplest form, stack instructions add items in sequential order to the top of the stack and retrieve them in the reverse order. A program can access several stack areas, but can use only one stack area at any time. The processor, using the push-down stack concept, pushes (stores) data onto the stack (toward higher addresses) and pops (retrieves) data from it in the reverse order (toward lower addresses).

For instance, the processor can push or pop the contents of up to four accumulators with the **WPSH** or **WPOP** instruction. In addition, the processor can push a return block for a subroutine call, an I/O interrupt request, or a fault. Then a return block would be popped upon returning from the call, interrupt, or fault handler.

The 32-bit processor provides facilities for wide and narrow stack operations. The wide stack, a series of doublewords, supports 32-bit programs. The system includes four 32-bit stack management registers to manage wide stack operations. The narrow stack, a series of single words, supports 16-bit programs (for ECLIPSE 16-bit program development and upward program compatibility). The system uses three words per ring in reserved memory to manage narrow stack operations.

This chapter presents the wide stack operations and instructions. Refer to the chapter, "ECLIPSE 16-Bit Compatibility," for further information on the narrow stack. The chapter, "Program Flow Management," presents wide and narrow stack fault handling.

Wide Stack Operations

Each segment contains a set of wide stack parameters that the processor manages in the current segment with four 32-bit stack registers. You can modify the contents of the stack registers with instructions that move data between an accumulator and a stack register.

When transferring program control to another segment, the processor stores the contents of the stack registers in page zero of the current segment and initializes the contents of the stack registers from page zero of the destination segment.

CAUTION: *A program must not refer to or modify the stack parameters in page zero of the current segment.*

When a program executes in one ring, the stack must reside either in that ring or a higher ring and may span ring boundaries. Extreme care should be taken when using a stack that crosses an upper ring boundary as certain locations, such as page zero, may be affected (a ring crossing to that ring may then produce undefined results).

NOTE: *A segment violation fault is NOT generated when executing a stack-related instruction (such as* **WPOPJ***) which affects that portion of the stack extending into the next-higher ring (this occurs when popping the stack contents from the higher ring back into the ring of execution).*

Figure 4-1 shows the four stack parameters. Items (1) and (2) identify the lower stack limit (base) and upper stack limit, which define the locations that the stack occupies. Items (3) and (4) identify the wide stack pointer and the wide frame pointer, which address the data in the stack.

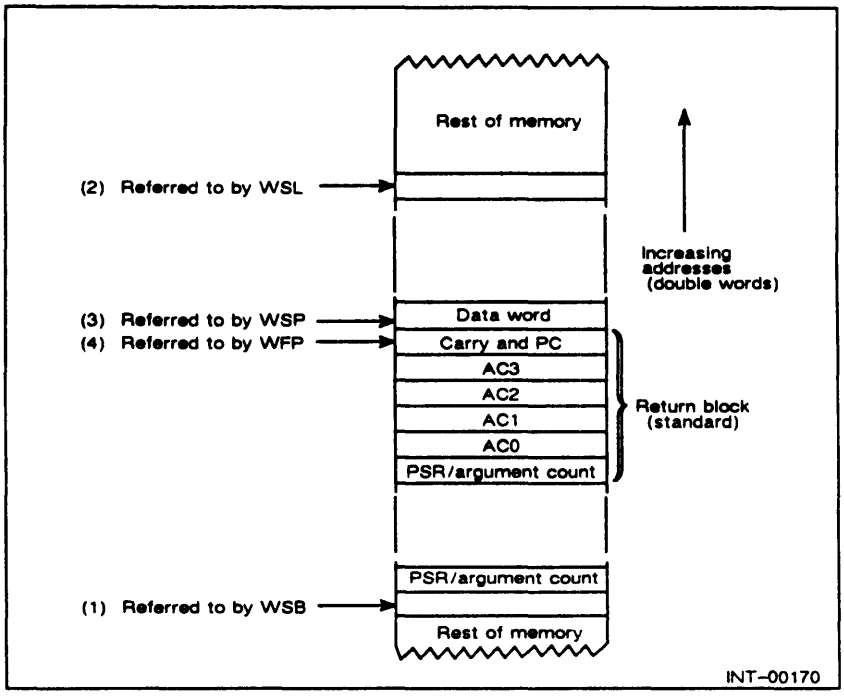


Figure 4-1 *Typical wide stack*

Wide Stack Registers

For most efficient operation, the contents of the wide stack registers should be initialized to address locations that are aligned on doubleword boundaries (even addresses).

The processor accesses the stack management registers to save or restore them when changing program flow between segments. When the processor transfers program control to another segment, it initializes all four wide stack registers using the contents of reserved memory locations in page zero of the new segment. Figure 4-2 shows the format of these registers; Table 4-1 describes the format.

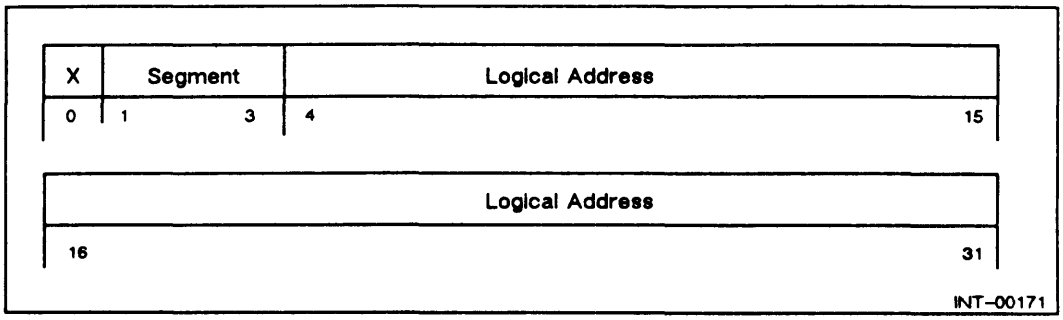


Figure 4-2 *Wide stack management register format*

Table 4-1 *Wide stack management register format description*

Name	Bit	Contents
x	0	Initially this value must be zero (in the stack limit this bit is set to one when a stack fault occurs).
Segment	1-3	Specifies the segment location of the stack.
Logical Address	4-31	Specifies a logical address within the segment.

Wide Stack Base

The wide stack base (WSB) defines the lower limit of the wide stack. When you initialize a wide stack, the wide stack base should be one doubleword below the actual address of the first doubleword in the wide stack.

The processor uses the contents of the wide stack base when it pops data from the wide stack. For instance, when returning from a subroutine, the processor pops a wide return block and then checks for a wide stack underflow. If the wide stack pointer value is less than the wide stack base value, an underflow condition exists. Refer to the section, "Wide Stack Faults," for further information on handling an underflow fault.

Wide Stack Limit

The wide stack limit (WSL) defines the upper limit of the wide stack. When you initialize a wide stack, the wide stack limit should be 18 doublewords below the actual address of the last doubleword in the wide stack, so that there will be space for handling a stack overflow if one occurs.

The processor pushes one or more doublewords onto the wide stack (such as a wide return block when calling a subroutine), and then for most operations checks for a stack overflow fault. (However, the processor checks for overflow before pushing data onto the stack when using the wide save instructions — **WSAVR**, **WSAVS**, **WSSVR**, and **WSSVS** — or when crossing to a subroutine in a lower-numbered segment.)

To check for a wide stack overflow fault, the processor compares the wide stack pointer contents to the wide stack limit contents. If the wide stack pointer contents are greater than the wide stack limit contents, an overflow condition exists. Refer to the section, "Wide Stack Faults," for further information on handling an overflow fault.

Wide Stack Pointer

The wide stack pointer (WSP) addresses the top location of the wide stack, either the location of the last doubleword placed on the stack (when adding data to the stack) or the next doubleword available from the stack (when removing data from the stack). When initializing a wide stack, set the wide stack pointer so that it is equal to the address in the wide stack base register.

To push a doubleword, the processor increments the wide stack pointer by two and stores a doubleword onto the stack. A pop operation retrieves one or more doublewords from the wide stack and decrements the wide stack pointer by two for each doubleword it pops.

NOTE: *The area between the wide stack pointer and the wide stack limit can be modified by the processor. For example, the* **WEDIT** *instruction may implicitly push and pop temporary* **WEDIT** *data.*

Wide Frame Pointer

The wide frame pointer (WFP) -- unchanged by push and pop operations -- defines a reference point in the wide stack. When setting up a wide stack, initialize the wide frame pointer so that it has the same value as the wide stack pointer. This preserves the original value of the wide stack pointer.

The processor stores and resets the value of the wide frame pointer when entering or leaving subroutines. Thus, the wide frame pointer identifies the boundary between words placed on the wide stack before a subroutine call, and between words placed on the wide stack during a subroutine execution. Using the wide frame pointer as a reference, the processor can move back into the wide stack and retrieve arguments stored there by a preceding routine.

Wide Stack Register Instructions

The instructions listed in Table 4-2 load (or modify) a wide stack register with data from an accumulator or store data in an accumulator from a wide stack register. In addition, if the **LCALL** or **XCALL** instruction transfers program control to another segment, the processor initializes all four wide stack registers (using the contents of reserved memory locations in page zero of the new segment). When transferring program control back to the original segment (with the **WRTN** instruction), the processor initializes the stack registers from the reserved memory locations of the original segment.

Table 4-2 *Wide stack register instructions*

Instruction	Operation
LCALL	Call subroutine (return from call with WRTN)
LDAFP	Load accumulator with the WFP register contents
LDASB	Load accumulator with the WSB register contents
LDASL	Load accumulator with the WSL register contents
LDASP	Load accumulator with the WSP register contents
STAFP	Store accumulator in the WFP register
STASB	Store accumulator in the WSB register
STASL	Store accumulator in the WSL register
STASP	Store accumulator in the WSP register
WMSP	Wide modify WSP register
WRTN	Wide return control from subroutine (LCALL, XCALL)
XCALL	Call subroutine (return from call with WRTN)

Wide Stack Data Instructions

The wide stack data instructions access a doubleword or a block of doublewords. All the wide stack data instructions which push or pop data on the stack increment or decrement the wide stack pointer.

- Instructions which use the wide stack pointer only to access data (such as **LDATS** and **DSZTS**) do not modify WSP.
- Instructions that access a doubleword modify the wide stack pointer by two.
- Instructions that access a block of doublewords modify the wide stack pointer by four or more (depending upon the size of the data block or return block).

The instructions in Table 4-3 access a doubleword or a block of doublewords.

Table 4-3 *Wide stack doubleword access instructions*

Instruction	Operation
DSZTS *	Decrement the doubleword addressed by WSP (skip if zero)
ISZTS *	Increment the doubleword addressed by WSP (skip if zero)
LDATS *	Load accumulator with doubleword addressed by WSP
LPEF	Push address
LPEFB	Push byte address
LPSHJ	Push jump to subroutine (pop with WPOPJ)
NBStc *	Narrow backward search queue and skip
NFStc *	Narrow forward search queue and skip
STATS *	Store accumulator into doubleword addressed by WSP
WBStc *	Wide backward search queue and skip
WFPOP	Wide floating-point pop
WFPSH	Wide floating-point push
WFStc *	Wide forward search queue and skip
WPOP	Wide pop accumulators (push with WPSH)
WPOPJ	Wide pop PC and jump (push with LPSHJ or XPSHJ)
WPSH	Wide push accumulators (pop with WPOP)
XPEF	Push address
XPEFB	Push byte address
XPSHJ	Push jump to subroutine (pop with WPOPJ)

* *Instruction uses but does not modify WSP.*

The instructions in Table 4-4 push or pop a return block. Although the return block can take several forms, it usually consists of six doublewords. Table 4-5 lists the standard return block with the order of items pushed and popped.

Table 4-4 *Wide stack return block instructions*

Instruction	Operation
BKPT	Breakpoint handler (return from breakpoint handler with PBX)
PBX	Pop block and execute (return from breakpoint handler)
WPOPB	Wide pop block
WRSTR	Wide restore from an interrupt
WRTN	Wide return via wide save (WSAVR, WSAVS, WSSVR, and WSSVS)
WSAVR	Wide save (reset overflow mask), used with LCALL and XCALL
WSAVS	Wide save (set overflow mask), used with LCALL and XCALL
WSSVR	Wide special save (reset overflow mask), used with LJSR & XJSR
WSSVS	Wide special save (set overflow mask), used with LJSR & XJSR
WXOP	Extended operation (return with WPOPB; used to expand instruction set)

Table 4-5 *Standard wide return block*

Doubleword Number in Block			
Pushed	Popped	Name	Contents
1	6	PSR/ARG_COUNT	Bits 0-15 contain the PSR. Bits 16-31 contain either all zeros or an argument count from the LCALL or XCALL instruction.
2	5	AC0	Contents of accumulator 0
3	4	AC1	Contents of accumulator 1
4	3	AC2	Contents of accumulator 2
5	2	AC3	Contents of accumulator 3 (or WFP before the push)
6	1	CRY/PC	Bit 0 contains Carry. Bits 1-31 contain the PC return address.

The *Instruction Dictionary* presents the return block contents with each subroutine instruction description. The chapter, "Program Flow Management," identifies the return blocks for the nonprivileged faults, while the chapter, "Device Management," presents the return block for an I/O interrupt. The chapter, "Memory and System Management," identifies the return blocks for privileged operations.

Initializing A Wide Stack

Figure 4-3 illustrates assembler code for initializing a wide stack. The stack resides in locations 256_{10} through 355_{10}. In this example, the processor detects a stack overflow 18 doublewords before the actual end of the stack — if a stack overflow does occur, the 18 reserved doublewords (ENDZ) provides stack space for a wide stack fault return block.

```
        .NREL
BASE:   .BLK    66.             ;Reserve 66 words for the wide stack
ENDZ:   .BLK    36.             ;Reserve 36 words for wide stack end zone
        .
        .
        .
        XLEF    0,ENDZ          ;Initialize WSL for a stack
        STASL   0               ;  overflow when WSP = BASE+66
        XLEF    0,BASE-2
        STASB   0               ;Initialize WSB
        STASP   0               ;Initialize WSP
        STAFP   0               ;Initialize WFP
        .
        .
        .
        XPEFB   BYTZ*2          ;Calculate and store the byte address
        .                       ;  for BYTZ on the stack.
        .
        .
```

INT-00172

Figure 4-3 *Sample code for initializing a wide stack*

Figure 4-4 illustrates the result of executing the assembler code in Figure 4-3. The **XPEFB** instruction calculates and pushes a byte address onto the stack.

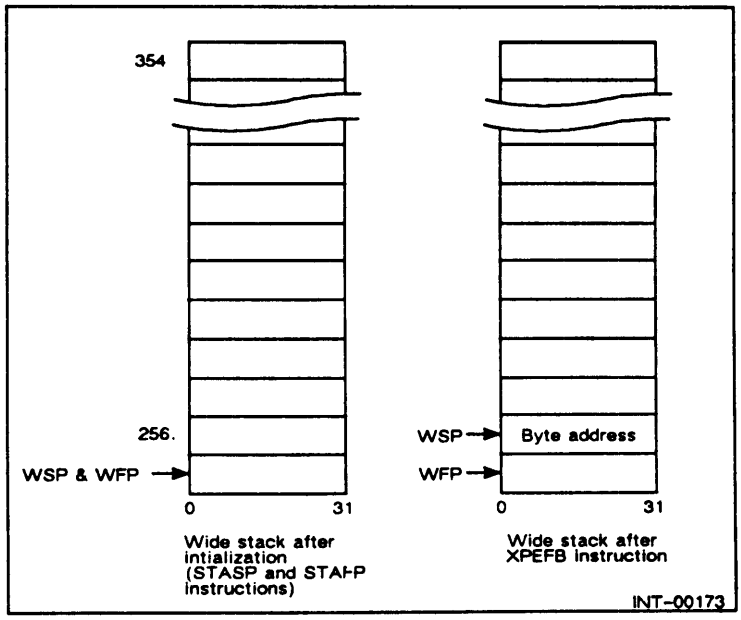


Figure 4-4 *Example of wide stack operations*

Wide Stack Faults

Stack overflow and underflow are stack faults. Stack overflow occurs when a program pushes data into the area beyond that allocated for the stack. Stack underflow occurs when a program pops data from the area beyond that allocated for the stack. Once detected, the processor always processes a stack fault.

After pushing data onto the stack, the processor checks for a stack overflow by comparing the value of the wide stack pointer to the value of the wide stack limit. If the value of the wide stack pointer is greater, a stack overflow exists. Loading the value 37777777777_8 into the wide stack limit register disables wide stack overflow fault detection.

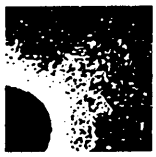
After popping data from the stack, the processor checks for a stack underflow by comparing the value of the wide stack pointer to the value of the wide stack base. If the value of the wide stack pointer is less, a stack underflow exists. Loading the value 20000000000_8 into the wide stack base register disables wide stack underflow fault detection.

Table 4-6 lists some instructions that push or pop one or more doublewords onto the wide stack. The table also lists the number of words required beyond the wide stack limit for a stack fault return block. Refer to the chapter, "Program Flow Management," for a description of stack fault servicing.

Table 4-6 *Instructions affecting the wide stack*

Instruction	Description	Doublewords Pushed or (Popped)	Doublewords Required Beyond WSL for Stack Fault
BKPT	Breakpoint handler	6	12
LCALL	Subroutine call	1	7
LFAMD, etc.	Floating-point arithmetic with TE enabled	0	12
LPEF	Push address	1	7
LPEFB	Push byte address	1	7
LPSHJ	Push jump	1	7
PBX	Pop block and execute	(6)	6
WADD, etc.	Fixed-point arithmetic with OVK enabled	0	12
WEDIT	Wide edit	9	15
WFPOP	Wide floating-point pop	(10)	6
WFPSH	Wide floating-point push	10	16
WPOP	Wide pop accumulators	(1-4)	6
WPOPB	Wide pop block	(6)	6
WPOPJ	Wide pop PC and jump	(1)	6
WPSH	Wide push accumulators	1-4	10
WRSTR	Wide restore	(11)	6
WRTN	Wide return	(6)	6
WSAVR	Wide save/reset OVK	5	11
WSAVS	Wide save/set OVK	5	11
WSSVR	Wide special save/reset OVK	6	12
WSSVS	Wide special save/set OVK	6	12
WXOP	Extended operation	6	12
XCALL	Subroutine call	1	7
XPSHJ	Push jump to subroutine	1	7
XVCT	Vector on I/O interrupt	6 or 11	17

End of Chapter



5

Program Flow Management

This chapter explains program flow, related instruction groups, transferring program control to another segment, and handling faults. Refer to the chapter, "Device Management," for I/O interrupt processing.

The program counter (PC) specifies the logical address of the instruction to execute. Thus, it controls the execution sequence of instructions. Since only bits 4 through 31 of the program counter are incremented, address wraparound occurs within the current segment.

NOTE: *If the address translation unit is off, addresses are physical ones. In this case, the program counter is undefined for values greater than 512 Mbytes.*

To address the next instruction (for normal program flow), the processor increments the program counter by

- one, when executing a one-word instruction (such as **NADI**).
- two, when executing a two-word instruction (such as **NADDI**).
- three, when executing a three-word instruction (such as **LNADI**).
- four, when executing a four-word instruction (such as **LCALL**).

NOTE: *The* **WCLM** *instruction increments the program counter by five when both source and destination accumulators are specified as the same accumulator (the instruction then includes two 32-bit immediate values).*

Any of the following events alter the normal program flow.

- Executing the **XCT** instruction.
- Executing a jump instruction.
- Executing a skip instruction.
- Executing a subroutine call or return instruction.
- Detecting a fault.
- Detecting an I/O interrupt request.

Related Instruction Groups

The next section explains related instruction groups such as the Execute Accumulator, jump, skip, and subroutine call or return instructions.

Execute Accumulator

The Execute Accumulator instruction (**XCT**) executes bits 16-31 of an accumulator as an instruction. If these bits are the first 16 bits (word) of a multiword instruction, the additional required words of the instruction are fetched from words immediately following the **XCT** instruction. After executing the accumulator contents, program flow continues at one of the following locations.

- The first location after the **XCT** instruction (assuming a 16-bit instruction was executed).
- The second, third, or fourth location after the **XCT** instruction, if the contents of the accumulator is the first of a two-, three-, or four-word instruction.
- The effective address, if the accumulator contains an instruction that alters normal program flow, such as a jump or skip instruction.

Jump

A jump instruction loads the effective address into the program counter. Program flow continues at the effective address. A jump instruction does not save a return address. The jump instructions are listed in Table 5-1.

Table 5-1 *Jump instructions*

Instruction	Operation
DSPA *	Dispatch (with narrow displacement)
JMP *	Jump (with narrow displacement)
LDSP	Dispatch (with long displacement)
LJMP	Jump (with long displacement)
WBR	Branch (PC-relative jump)
XJMP	Jump (with extended displacement)

* *ECLIPSE compatible instruction*

Skip

A skip instruction causes the processor to either execute the next word in the instruction stream, or to jump over this word and execute the second word following the skip instruction. To skip, the processor adds one to the program counter. During most skip instructions, the processor first tests a machine condition or status, and on the basis of test results, executes the first or second word as an instruction.

When using a skip instruction, verify that the skip does not transfer control to the middle of a two (or more) word instruction. For instance, the two lines of code starting at "NOGO:" in Figure 5-1 perform an undesired skip because the program counter contains the address of the first word of the **LFDMD** two-word displacement. The three lines of code starting at "SKIPOK:" in Figure 5-1 perform the skip properly.

```
NOGO:   FSEQ                    ;Skip on zero
        LFDMD   0,@OPAND        ;Floating-point divide with a two-word displacement
        .
        .
        .
SKIPOK: FSNE                    ;Skip on nonzero and execute the LFDMD instruction
        WBR     NEXT            ;Zero -- skip the LFDMD instruction
        LFDMD   0,@OPAND        ;Floating-point divide with a two-word displacement
NEXT:   .
        .
        .
                                                                    INT-00174
```

Figure 5-1 *Illegal and legal skip instruction sequences*

Certain skip instructions modify the program counter by other than one (or zero) word. Table 5-2 lists these instructions; the remainder of this section describes how the DO-loop and search queue instructions modify the program counter.

Table 5-2 *Skip instructions*

Instruction	Operation
FNS *	No skip
FSA *	Skip always
LNDO	Narrow do until greater than
LWDO	Wide do until greater than
XNDO	Narrow do until greater than
XWDO	Wide do until greater than
NBStc	Narrow search queue backward
NFStc	Narrow search queue forward
WBStc	Wide search queue backward
WFStc	Wide search queue forward

* *ECLIPSE compatible instruction*

A DO-loop instruction (**LNDO**, **LWDO**, **XNDO**, and **XWDO**) increments a loop variable by one and then compares it to a value in a specified accumulator. The processor executes the

- First instruction in the DO-loop sequence while the incremented variable equals (or remains less than) the value in the accumulator.
- Instruction following the DO-loop sequence when the incremented variable becomes greater than the value in the accumulator.

The processor skips the DO-loop sequence of instructions by adding one plus the termination offset (for skipping the DO-loop sequence) to the current program counter value. The processor then loads this sum into the program counter.

For example, the lines of code in Figure 5-2 perform a valid DO-loop sequence.

```
        WSUB    0,0             ;Get a 0
        XNSTA   0,INDEX         ;Initialize the counter in memory
        NLDAI   5,0             ;Maximum index value
LOOP:   XNDO    0,END -.,INDEX  ;Start of the Do-loop
          .                     ;New index value is in AC) and may be
          .                     ;used by computations in the loop
          .
        WBR     LOOP
END:    . . . .                 ;Loop was executed 5 times
          .
          .
INDEX:  .WORD 0                 ;Index value
                                                            INT-00175
```

Figure 5-2 *DO-loop instruction sequence*

A search queue instruction (**NBStc**, **NFStc**, **WBStc**, and **WFStc**) causes the program counter to skip one, two, or three words when an explicit queue element exists. The first word following the search queue instruction normally contains a jump instruction to a routine that handles an unsuccessful return from the queue search, while the second word is a jump to a routine for an interrupted queue search. The third word following the search queue instruction is the location where a successful queue search returns. Refer to the chapter, "Queue Management," for more information on the search queue instructions.

In addition to the program flow and search queue instructions, additional skip instructions are available for fixed-point, floating-point, and I/O operations. For more information, refer to the following chapters.

- "Fixed-Point Computing" for the fixed-point skip instructions.
- "Floating-Point Computing" for the floating-point skip instructions.
- "Device Management" for the I/O skip instructions.

Subroutine

Some instructions that call a subroutine push arguments and/or a return block onto the wide stack before actually jumping to the subroutine. These instructions also require specific instructions to properly return from a subroutine. Table 5-4 lists the subroutine, save, and return instructions. Table 5-5 illustrates the relationships between the various subroutine instructions and their corresponding return instructions. A description of the subroutine instructions and an example with wide stack operations follows the tables.

For instructions which do not explicitly push information onto the stack, the recommended procedure for calling and returning from a subroutine is:

1. Push any arguments (to be passed to the subroutine) onto the stack.
2. Jump to (using an **LJSR** or **XJSR** instruction) or call (with an **LCALL** or **XCALL** instruction) the subroutine. This places the effective address into the program counter.
3. Use a save instruction as the first instruction in the subroutine (to push a return block and other return information onto the stack).
4. Return from the subroutine using the appropriate subroutine return instruction (see Table 5-5). The subroutine return instruction (generally a **WRTN** instruction) pops the wide return block from the stack, thus, restoring the Carry bit, program counter, and accumulators. Program flow continues with the instruction following the subroutine call (unless the popped return block updates the program counter).

Although a wide return block can take several forms, it usually consists of six doublewords, as shown in Table 5-3. The fifth doubleword contains the contents of AC3 (for a **BKPT** or **WXOP** instruction) or the previous wide frame pointer (for **XCALL** or **LCALL** and **WSAVS** or **WSAVR** instructions). Bit 0 of the sixth doubleword contains Carry; bits 1-31 contain the program counter.

Table 5-3 *Standard wide return block*

Doubleword Number In Block Pushed	Popped	Name	Contents
1	6	PSR/ARG_COUNT	Bits 0-15 contain the PSR. Bits 16-31 contain either all zeros or an argument count from the LCALL or XCALL instruction.
2	5	AC0	Contents of accumulator 0
3	4	AC1	Contents of accumulator 1
4	3	AC2	Contents of accumulator 2
5	2	AC3	Contents of accumulator 3 (or WFP before the push)
6	1	CRY/PC	Bit 0 contains Carry. Bits 1-31 contain the PC return address.

Table 5-4 *Subroutine instructions*

Instruction	Operation
BKPT	Breakpoint handler
LCALL	Call subroutine (long displacement)
LJSR	Jump to subroutine (long displacement)
LPSHJ	Push jump (long displacement)
PBX	Pop block and execute
WEDIT	Wide edit of alphanumeric data
WPOPB	Wide pop block
WPOPJ	Wide pop PC and jump
WRTN	Wide return
WSAVR	Wide save and reset overflow mask
WSAVS	Wide save and set overflow mask
WSSVR	Wide special save and reset overflow mask
WSSVS	Wide special save and set overflow mask
WXOP	Wide extended operation
XCALL	Call subroutine (extended displacement)
XJSR	Jump to subroutine (extended displacement)
XPSHJ	Push jump (extended displacement)

Table 5-5 *Sequence of subroutine instructions*

Call Instruction or Sequence	Segment Crossing Permitted	Associated Save Instruction	Return Instruction
BKPT	No		PBX/WPOPB*
LCALL	Yes	WSAVR	WRTN
	Yes	WSAVS	WRTN
LJSR	No	WSSVR	WRTN
	No	WSSVS	WRTN
LPSHJ	No		WPOPJ
WEDIT	No		DEND
WXOP	No		WPOPB
XCALL	Yes	WSAVR	WRTN
	Yes	WSAVS	WRTN
XJSR	No	WSSVR	WRTN
	No	WSSVS	WRTN
XPSHJ	No		WPOPJ

* *Use the* BKPT/WPOPB *instruction sequence when removing the* BKPT *instruction before returning from the breakpoint handler.*

Jump to Subroutine

A jump to a subroutine (**LJSR** or **XJSR**) instruction transfers program control to a subroutine in the current segment. These instructions store the return address in AC3 and transfer program control to the effective address. As the first instruction of the subroutine, a wide special save (**WSSVR** or **WSSVS**) instruction pushes a standard wide return block onto the wide stack, manipulates some of the PSR bits, and reserves stack space for local variables. As the last instruction of the subroutine, the Wide Return (**WRTN**) instruction returns program control from the subroutine.

A push and jump to a subroutine (**LPSHJ** or **XPSHJ**) instruction pushes a return address onto the wide stack and transfers program control to the effective address in the current segment. As the last instruction of the subroutine, the **WPOPJ** instruction returns program control from the subroutine.

Call Subroutine

A call to a subroutine (**LCALL** or **XCALL**) instruction transfers program control to a subroutine in the current segment or in another segment and pushes (or copies) a doubleword onto the destination wide stack. As the first instruction of the subroutine, a wide save (**WSAVR** or **WSAVS**) instruction pushes a standard wide return block onto the wide stack in the destination segment, sets or resets the OVK bit of the PSR, and reserves stack space for local variables. As the last instruction of the subroutine, the Wide Return (**WRTN**) instruction returns program control from the subroutine. (Refer to the next section for a complete description of transferring program control to another segment.)

Breakpoint Instruction

The Breakpoint (**BKPT**) instruction pushes a wide return block and transfers program control to the breakpoint handler. The Pop Block and Execute (**PBX**) instruction returns program control from the breakpoint handler.

Before executing **BKPT**, first save elsewhere in memory the one-word opcode from the location that the **BKPT** instruction will occupy. Then, store the **BKPT** instruction in that one-word location.

When the processor executes the **BKPT** instruction, it pushes a wide return block onto the current stack and jumps to the breakpoint handler. When returning program control, the breakpoint handler must load the one-word opcode from memory into AC0. Then it executes the **PBX** instruction, which temporarily

1. Disables the interrupt system for one instruction execution;
2. Saves the one-word opcode in AC0 (bits 16–31) and performs the function of a **WPOPB** instruction;
3. Replaces the **BKPT** instruction with the temporarily saved one-word opcode and then continues normal program flow.

If an interrupt occurs while the processor is executing the saved instruction (the program counter points to the **BKPT** instruction), the processor sets the IXCT flag in the PSR and pushes the opcode of the saved instruction on the wide stack. Upon returning from the interrupt handler, the **BKPT** instruction tests the IXCT flag. If the flag is set, the **BKPT** instruction resets the flag to 0, pops the saved opcode of the interrupted instruction off the wide stack, and executes it.

NOTE: *If you remove the* **BKPT** *instruction (replacing it with another instruction) before returning from the breakpoint handler, you must use the* **WPOPB** *instruction to return from the handler.*

Wide Edit Instruction

The Wide Edit (**WEDIT**) instruction transfers control to an edit subprogram without changing the program counter. The **WEDIT** instruction subprogram then uses a byte pointer as its program counter. The subprogram End Edit (**DEND**) instruction returns program control to the instruction following the **WEDIT** instruction. Refer to the **WEDIT** description in the *Instruction Dictionary* for complete information.

Example With Wide Stack Operations

The following illustrates the effects of a jump to subroutine (**XJSR**) instruction on a wide stack. The jump occurs within the current segment. The routine passes arguments to the subroutine by pushing the arguments onto the stack before executing the **XJSR** instruction.

Figure 5-3 illustrates the lines of processor-related assembler code for beginning and ending a subroutine. The first instruction of the subroutine is a wide special save instruction (**WSSVS**) and the last instruction of the subroutine is a Wide Return instruction (**WRTN**). The second instruction of the subroutine (**XPEF**) further illustrates wide stack operations.

```
        .
        XJSR    SUB     ;Jump to subroutine.
        .
SUB:    WSSVS   0       ;Save a wide return block.
HERE:   XPEF    HERE    ;Calculate and push this address into the
        .               ;wide stack.
        .
        .
        WRTN            ;Return from subroutine call.
```

INT-00176

Figure 5-3 *Example of subroutine code for* **XJSR**

Figures 5-4 and 5-5 illustrate the result of executing the assembler code in Figure 5-3. During the **XJSR** instruction, the processor stores the return address into AC3 and jumps to the subroutine. With **WSSVS** as the first instruction of the subroutine, the processor stores the PSR, AC0-AC2, the old WFP, Carry (C), and AC3 (return address) into the wide stack.

Although Figures 5-4 and 5-5 illustrate that the wide stack resides between 256_{10} and 355_{10}, the wide stack can be of any size and can reside almost anywhere within the segment of execution (or a higher numbered segment). Refer to the chapter, "Stack Management," for further wide stack information.

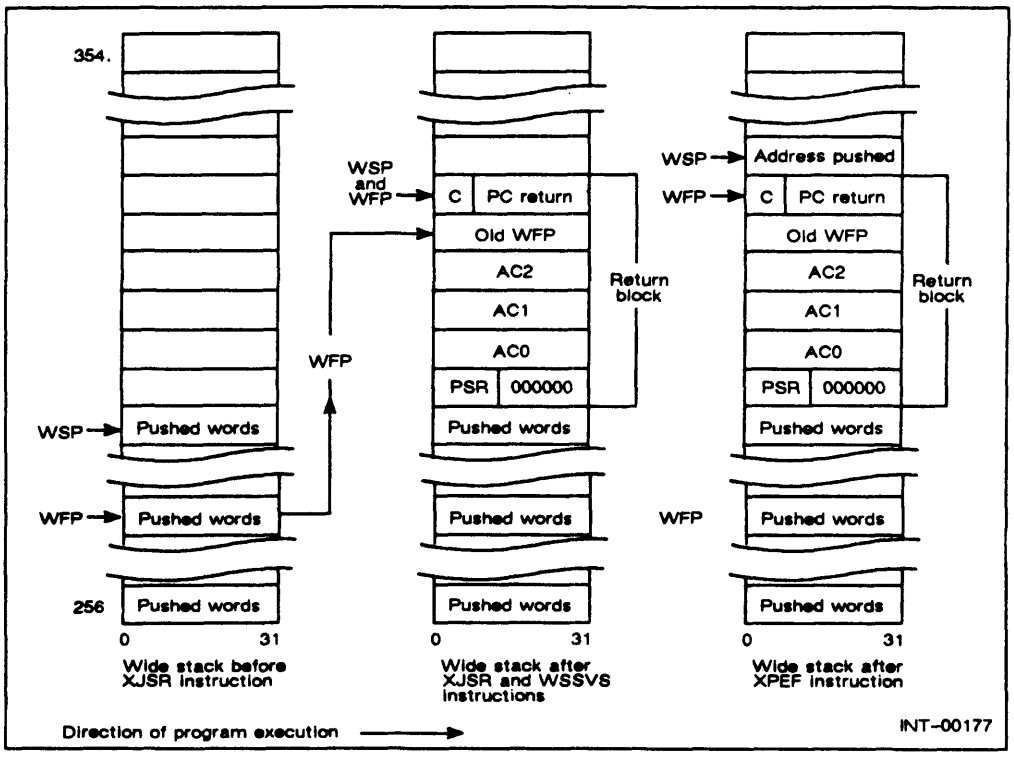


Figure 5-4 *Wide stack operations from* **XJSR**, **WSSVS**, *and* **XPEF** *instructions*

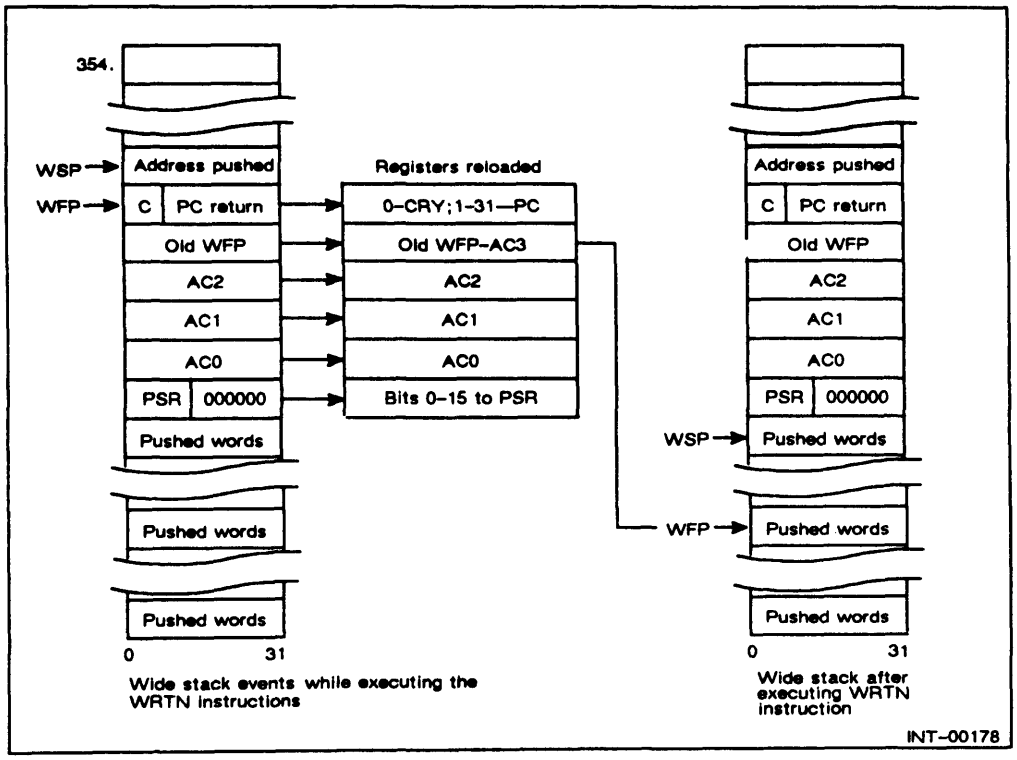


Figure 5-5 *Wide stack operations from* **WRTN** *instruction*

Transferring Program Control To Another Segment

The ring structure of ECLIPSE MV/Family processors limits the transfer of program control (for most instructions) to the current ring of execution (CRE). In order to transfer program control to another segment, access to the other segment must be through a gate array using a call subroutine instruction. The instructions listed in Table 5-6 transfer program control to or from another segment.

Table 5-6 *Segment program control transfer instructions*

Instruction	Operation
LCALL	Call subroutine
WDPOP	Wide return from page fault
WPOPB	Wide pop block
WRTN	Wide return
XCALL	Call subroutine
WRSTR	Wide restore from an I/O interrupt

The **LCALL** and **XCALL** instructions initiate the transfer to another segment. The **WRTN** instruction returns program control from the **LCALL** and **XCALL** instructions. The **WRSTR** instruction returns program control from a base level I/O interrupt. The **WPOPB** instruction returns program control from an intermediate-level I/O interrupt. Refer to the chapter, "Device Management," for a description of I/O interrupts.

The processor checks the direction of a transfer. A subroutine call must be inward (towards segment 0) and a return from a subroutine call or I/O interrupt must be outward (towards segment 7).

NOTE: *No segment crossing occurs with an interrupt request when the current segment equals 0 and the interrupt-servicing code resides in segment 0.*

If the processor detects an invalid segment crossing, it does not execute the instruction; instead, it initiates a protection fault in the originating segment. The processor sets AC1 to 7 for an illegal outward subroutine call, or sets AC1 to 8 for an illegal inward return.

NOTE: *The processor performs, without software assistance, all the functions necessary for a segment crossing.*

Subroutine Call

To transfer program control to another segment with the **XCALL** or **LCALL** instruction, the processor

1. Verifies that the instruction can access the destination segment.
2. Validates the entry point through a gate array in the destination segment.
3. Redefines the wide stack and transfers the call arguments to it.
4. Transfers program control to the destination segment.

Gate Array

A *gate array* is a series of locations that specify entry points (or gates) to the segment. The processor accesses a gate array through a pointer in page zero (locations 34_8 and 35_8 of reserved memory) of the destination segment. Figure 5-6 shows the format of a gate array; Table 5-7 describes the gate array format.

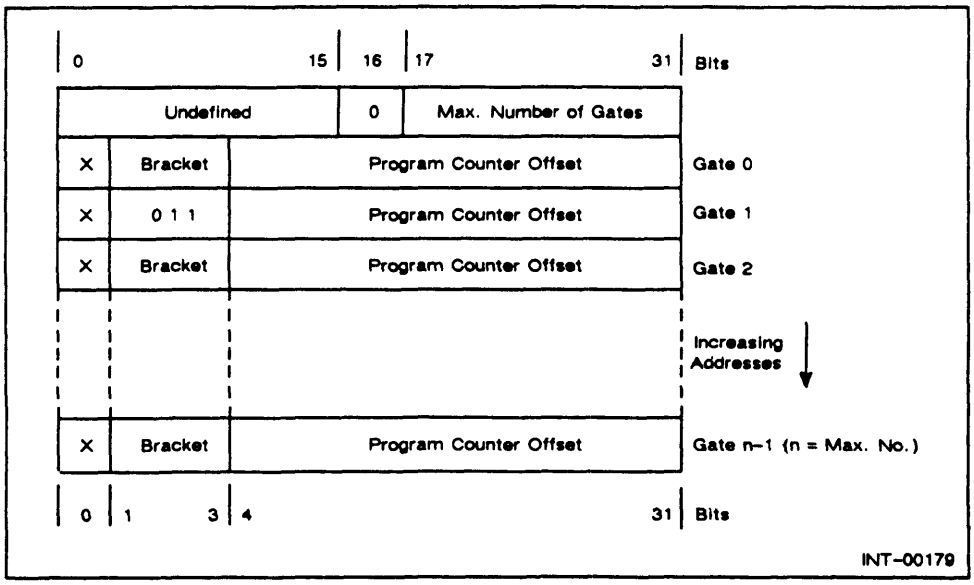


Figure 5-6 *Gate array format*

Table 5-7 *Gate array format description*

Name	Contents
Doubleword 1	
Undefined	The processor ignores these bits.
0	Bit 16 is always 0.
Maximum Number of Gates	The maximum number of gates (n) specifies the total number of gates. If the maximum number is zero, the destination segment cannot be the target of an inward segment crossing.
Doublewords 2 through n+1 (gates 0 through n-1)	
X	The processor ignores the contents of this bit.
Bracket	The bracket is the gate bracket, an unsigned integer in the range of zero to seven. The bracket identifies the highest numbered source segment that can use the gate. For instance, if the Gate 1 gate bracket contains 011_2, only segments 0 through 3 can access this segment.
Program Counter Offset	The program counter offset is the address of the first instruction of the subroutine in the destination segment (target address).

Transfer

The call subroutine instructions (**LCALL** and **XCALL**) are coded as:

LCALL [@]*displacement*[,*index*][,*argument_count*]
XCALL [@]*displacement*[,*index*][,*argument_count*]

The effective address formed from [@]*displacement*[,*index*] may specify an inner ring or the current ring. The 16-bit *argument_count* coded with the **LCALL** or **XCALL** instruction indicates the number of arguments pushed onto the wide stack.

Figure 5-7 shows how the processor interprets the final effective address; Table 5-8 describes the effective address format.

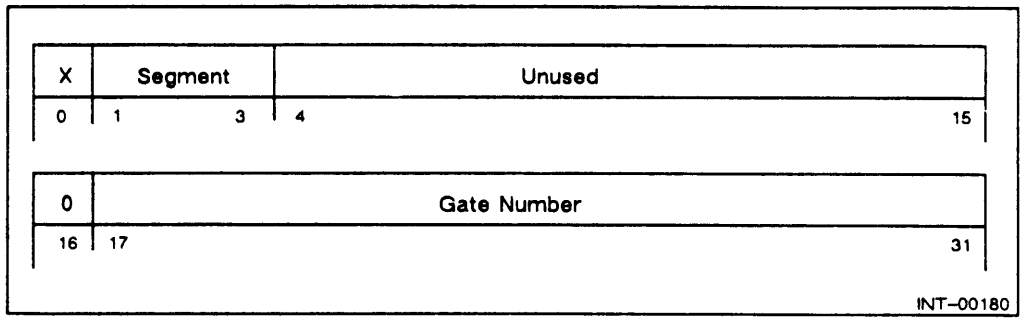


Figure 5-7 XCALL *or* LCALL *effective address*

Table 5-8 XCALL *or* LCALL *effective address format description*

Name	Bits	Contents
X	0	This bit is ignored by the processor.
Segment	1-3	The segment bits specify the segment number of the destination segment.
Unused	4-15	These bits are ignored by the processor.
0	16	Always 0.
Gate Number or Call Offset	17-31	Index to an element (a gate) in the gate array. If the segment bits (1-3) specify a destination segment less than the current segment of execution, then the gate number specified by bits 17-31 is a gate in the destination segment. If the destination segment specified by bits 1-3 equals the current segment of execution, then bits 17-31 do not indicate a gate number, but a target offset in the current ring to which control is transferred.

To perform a valid inward segment crossing, as depicted in Figure 5-8, the processor

1. Tests for a valid segment by checking the validity bit in the segment base register.

   If the segment is accessible, the processor continues to the next step.

   If the segment is not accessible, the processor aborts the call, sets AC1 to 3, and services the protection fault.

2. Checks for a valid gate by

   a. Comparing the gate number to the maximum number of gates.

      If the gate number is less than the maximum number of gates, the segment crossing continues.

   b. Comparing the segment number to the gate bracket number of the indexed gate.

      The segment number used for comparison is either the current segment of execution (if there was no indirection) or the segment of the last indirect address.

      If the segment number is equal to or less than the value in the gate bracket, the processor copies the segment number from the effective address to bits 1-3 of the program counter.

      Next, the processor copies the program counter offset (bits 4-31 from the indexed gate) to the program counter bits 4-31, and continues (to step 3).

   If a gate number or a gate bracket comparison fails, the processor aborts the call, sets AC1 to 6, and services the protection fault. The protection fault occurs in the source segment.

3. Stores the wide frame pointer and wide stack pointer registers into page zero locations of the source segment.

   The values of the wide stack limit and wide stack base registers should be identical to the values in reserved memory.

4. Redefines the wide stack for the destination segment by loading the wide stack pointer, wide stack limit, and wide stack base registers from page zero locations of the destination segment.

   NOTE: *Page zero must be memory resident. A page fault may not occur when referring to these locations because an infinite page fault will be signaled and the processor will halt.*

   The **WSAVS** *or* **WSAVR** *instruction subsequently initializes the wide frame pointer.*

5. Checks for a potential destination stack overflow.

   The 16-bit *argument_count* coded with the **LCALL** or **XCALL** instruction specifies the number of arguments pushed onto the wide stack. The processor uses this parameter to determine if the number of arguments to copy exceeds the size of the wide stack.

   If the processor detects a potential overflow, it does not copy the arguments. It sets AC1 to 2 and processes a stack fault in the destination segment. The program counter word in the return block contains the address of the first instruction to execute in the destination segment.

6. Creates a doubleword containing the current processor status register and the number of arguments pushed onto the source wide stack (*PSR/arg_count*). The actual contents of the *PSR/arg_count* doubleword depend on the value of the high bit in the *argument_count* coded with the call subroutine instruction. If the high bit equals

   - 0, the *PSR/arg_count* doubleword is in the following format:

     Bits 0–15 contain current PSR.
     Bits 16–31 contain *argument_count*.

   - 1, the processor assumes the top doubleword pushed on the source stack is in the following format:

     Bits 0–15 are undefined.
     Bits 16–31 contain *argument_count* with bit 16 equal to 0.

     In this case, the processor ignores the *argument_count* coded with the call subroutine instruction and places the current PSR into bits 0–15 of the stack doubleword.

   NOTE: *Before the* **LCALL** *or* **XCALL** *instruction completes execution, the processor removes the* PSR/arg_count *doubleword from the source segment's stack (only the destination segment's stack will contain the* PSR/arg_count *value).*

7. Copies the number of arguments from the source stack to the destination stack, if no potential overflow exists.

   The order of the arguments in the destination stack matches the order of the arguments in the source stack.

   NOTE: *The copying of arguments is interruptable.*

8. Pushes the *PSR/arg_count* doubleword onto the destination wide stack.

9. Executes the first instruction of the subroutine.

   A wide save instruction (**WSAVR** or **WSAVS**) should be the first instruction of the subroutine. Either instruction would push a return block onto the destination wide stack and load the wide frame pointer with the updated value of the wide stack pointer.

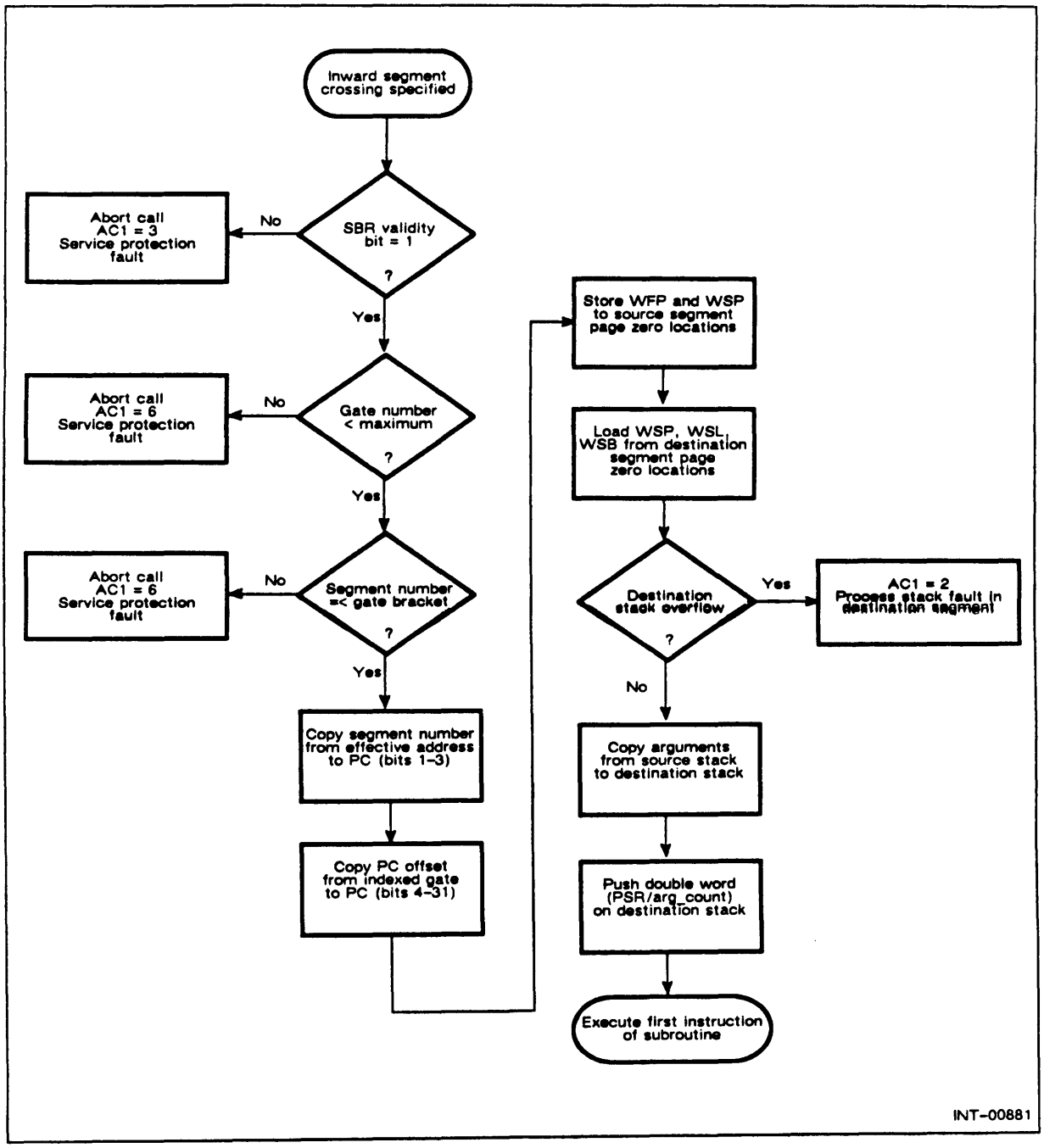


Figure 5-8 *Validating inward segment crossing sequence*

Trojan Horse Pointers

When executing a subroutine in another segment, the processor uses the access privileges of the destination segment to determine the validity of the reference. A *trojan horse pointer* exists if one of the arguments passed from the source segment points to a location in the destination segment (or a segment between the source and destination). A privileged access fault would occur if a program refers to a location in a lower numbered segment.

For example, a trojan horse pointer can exist when a program in segment 6 calls a subroutine in segment 2, and one of the arguments passed is a pointer to information in segment 2, 3, 4 or 5.

You can protect against a trojan horse pointer by using the Validate Word Pointer (**VWP**) or Validate Byte Pointer (**VBP**) instruction to ensure that the destination segment is greater than or equal to the source segment (before executing the subroutine).

The processor protects against a trojan horse pointer when it executes a character move instruction that moves data in descending order (such as **WCMT** and **WCMV**). The processor checks each data transfer and ensures that the source segment and destination segment remain the same.

Subroutine Return

As the last instruction of the subroutine, use the Wide Return instruction (**WRTN**) to return program control from the **LCALL** or **XCALL** instruction. Referring to Figure 5-9, the processor

1. Places the contents of the wide frame pointer into the wide stack pointer.
2. Pops the six doubleword return block.

   The processor pushed the first five doublewords of the return block when it executed the **WSAVR** or **WSAVS** instruction. The processor pushed the sixth doubleword (processor status register and the number of arguments) when it executed the **LCALL** or **XCALL** instruction.
3. Loads the program counter with the return address in the destination segment, and checks for inward return.
4. Stores the updated wide frame pointer and updated wide stack pointer registers into page zero locations of the source segment.

   The values of the wide stack limit and wide stack base registers should be identical to the values in reserved memory.
5. Redefines the wide stack for the destination segment by loading the wide stack limit, wide stack base, and wide frame pointer registers from page zero locations of the destination segment.

   NOTE: *Page zero must be memory resident. A page fault may not occur when referring to these locations because an infinite page fault will be signaled and the processor will halt.*
6. Calculates the address of the doubleword that precedes the arguments of the calling sequence and loads the wide stack pointer with this address. This address is equal to the destination (outer) segment's wide stack pointer value minus two times the argument count (from the return block).
7. Loads the program counter from the return block (which may be the instruction after the **LCALL** or **XCALL** instruction) and continues execution.

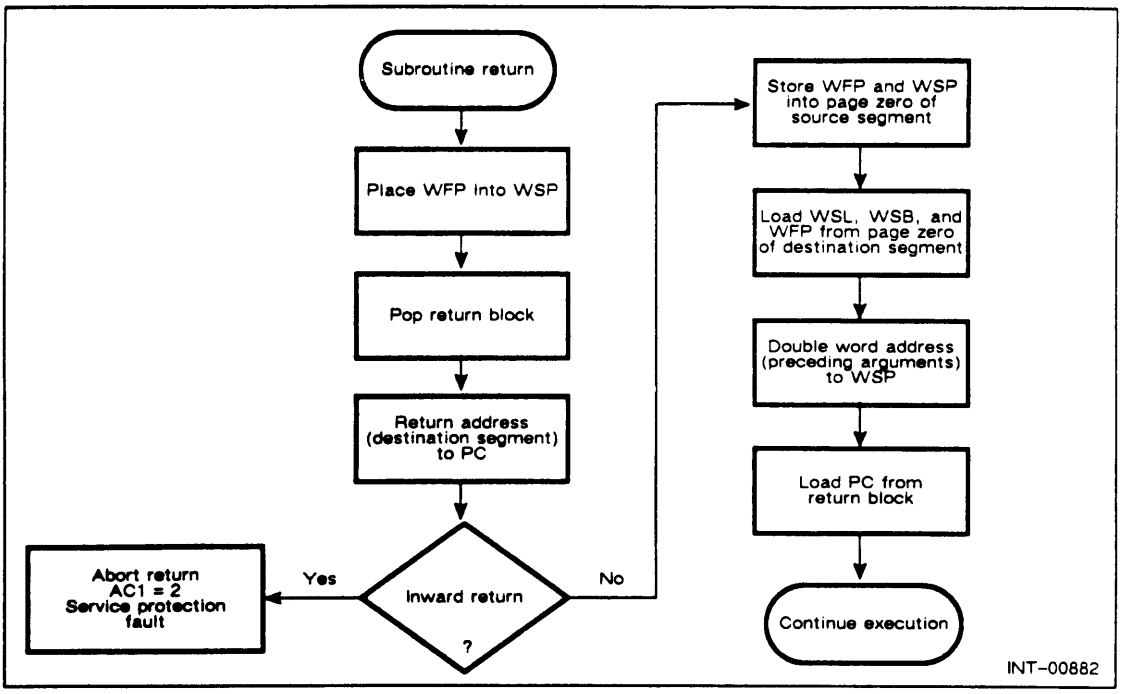


Figure 5-9 *Wide Return instruction sequence*

Fault Handling

While executing an instruction, the processor performs certain checks on the operation and the data. If the processor detects an error, a privileged or nonprivileged fault occurs before executing the next instruction. Table 5-9 lists the faults with their type.

Table 5-9 *Faults*

Fault	Type
Nonresident page	Privileged
Protection violation	Privileged
Stack operation	Nonprivileged
Fixed-point computation	Nonprivileged
Floating-point computation	Nonprivileged
Invalid decimal or ASCII data format	Nonprivileged

When the processor detects a fault, it pushes a return block onto the stack and jumps to the fault handler through the appropriate fault handler pointer in reserved memory. The initial and indirect pointers to a fault handler (except to a page fault handler) are 16 bits. Levels of indirection, if any, occur within the segment initially containing the pointer. A nonprivileged fault pointer is located in page zero of the current segment. A privileged fault pointer is located in page zero of segment 0.

If a privileged fault occurs while handling a nonprivileged fault, the processor suspends acting on the nonprivileged fault and processes the privileged fault. Refer to the chapter, "Memory and System Management," for privileged fault handling.

To service a nonprivileged fault, the processor

1. Sets AC1 to a value that identifies the fault when a stack fault, fixed-point fault or a decimal/ASCII fault occurs. Refer to the appendix, "Fault Codes," for a listing of fault codes.

2. Pushes a fault return block onto the stack.

   The fault return block contains the address of the instruction that the processor was executing at the time the fault occurred.

3. Checks for stack overflow.

   If a stack overflow occurs, the processor pushes a stack fault return block onto the stack and processes the stack fault. The stack fault return block contains the return address to the original fault.

   If no stack overflow occurs, the processor continues to service the original fault.

4. Jumps to the fault handler.

   The last instruction of a wide fault handler should be a **WPOPB** instruction so that the processor can continue to execute the interrupted program.

Execution of ECLIPSE 16-bit compatible instructions does not generate fixed-point faults. Certain ECLIPSE arithmetic instructions (**ADD**, **DIV**, etc.) set the state of the carry bit. If detection of the appropriate fault is desired, it is necessary to create a subroutine that checks the state of the carry bit upon completion of these instructions. A carry-out from accumulator bit 16 affects the system's carry bit upon execution of these ECLIPSE instructions. The *Instruction Dictionary* describes the ECLIPSE instruction set and the instructions which affect the carry bit.

Fixed-Point Overflow Fault

The processor detects a fixed-point overflow when attempting division by zero or when calculating a two's complement number that is too large to store in memory or in a fixed-point accumulator. The processor sets the processor status register (PSR) overflow flag (OVR) to one.

For the processor to service the fixed-point fault (or trap), you must set the PSR overflow fault mask (OVK) to one before the processor sets OVR. Use the **FXTE** instruction to set OVK to one, and the **FXTD** instruction to set OVK to zero.

- If OVK equals zero when the processor sets OVR to one, the processor ignores the overflow. OVR, however, remains set to one until explicitly changed. The processor continues normal program execution with the next sequential instruction.
- If OVK equals one, the processor initiates a fixed-point overflow fault at the end of the current fixed-point instruction.

To service a fixed-point fault, the processor

1. Pushes a wide return block (see Table 5-10 for the return block contents).

   The PSR value in the return block contains OVR set to 0, OVK set to 1, and IRES unchanged. The return address (doubleword number 6) is the address of the instruction the processor executes after servicing the fault.

   Table 5-10 *Fixed-point fault return block*

Doubleword in Block Pushed	Contents
1	PSR (Bits 0-15 contain the processor status register; bits 16-31 contain zeros)
2	AC0 (Bits 0-31)
3	AC1 (Bits 0-31)
4	AC2 (Bits 0-31)
5	AC3 (Bits 0-31)
6	CRY/PC (Bit 0 contains Carry; bits 1-31 contain address of the instruction following the fault-causing instruction)

2. Places the address of the instruction that caused the fault into AC0.
3. Sets the processor status register to zero.
4. Jumps to the fault handler through the 16-bit pointer in reserved memory (location 37_8 of the current segment).

Floating-Point Faults

The processor detects a floating-point error when

- Attempting division by zero
- Executing an IIS instruction with an invalid input argument
- Calculating a number too large to store in memory or in a floating-point accumulator.

The processor sets both the appropriate floating-point status register (FPSR) fault flag (OVF, UNF, INV, or MOF) and the ANY flag to one. If the error is an invalid input argument, the processor also returns a code to the FPSR INP bits.

For the processor to service a floating-point fault (or trap), you must set the floating-point fault mask (TE) to one before the processor sets a floating-point fault flag. Use the **FTE** instruction to set TE to one and the **FTD** instruction to set TE to zero.

- If TE equals zero when the processor sets a floating-point fault flag to one, the processor ignores the fault. The processor continues normal program execution with the next sequential instruction.
- If TE equals one when the processor sets a floating-point fault flag to one, the processor initiates a floating-point overflow fault at the end of the current floating-point instruction.

To service a floating-point fault, the processor

1. Pushes a return block onto the stack.

   The processor reads the 16-bit address of the floating-point fault handler from page zero (location 45_8) of reserved memory for the current segment. If the first word of the floating-point handler is an ECLIPSE MV/Family instruction (bit 0 equals 1 and bits 12 through 15 are 1000_2), the processor pushes a wide return block onto the wide stack. Otherwise, the processor pushes a narrow return block onto the narrow stack. Table 5-11 describes the wide return block and Table 5-12 describes the narrow return block.

   The return address in the return block (PC) is the address of the next instruction that the processor executes after servicing the fault. Use a store floating-point status instruction (**LFSST** or **FSST**) to determine the address of the floating-point instruction that caused the fault.

Table 5-11 *Wide floating-point fault return block*

Doubleword in Block Pushed	Contents
1	PSR (Bits 0-15 contain the processor status register; bits 16-31 contain zeros)
2	AC0 (Bits 0-31)
3	AC1 (Bits 0-31)
4	AC2 (Bits 0-31)
5	AC3 (Bits 0-31)
6	CRY/PC (Bit 0 contains Carry; bits 1-31 contain address of the instruction following the fault-causing instruction)

Table 5-12 *Narrow floating-point fault return block*

Word in Block Pushed	Contents
1	AC0 (Bits 16-31)
2	AC1 (Bits 16-31)
3	AC2 (Bits 16-31)
4	AC3 (Bits 16-31)
5	PC (Bit 0 equals CRY; bits 1-15 contain bits 17-31 of the address of the instruction following the fault-causing instruction)

2. Sets FPSR(TE) to zero to disable floating-point traps.

3. Sets the PSR to zero (for a wide floating-point fault).

4. Jumps to the fault handler through the 16-bit pointer in reserved memory location 45_8 of the current segment. (The processor services narrow and wide floating-point faults using the same pointer.)

Decimal and ASCII Data Faults

The processor checks for a valid decimal or ASCII data type and for valid data when executing any decimal instruction which requires a decimal string as input. If either the data type or the data is invalid, the fault occurs at the end of the current instruction.

To service a decimal/ASCII fault, the processor

1. Pushes a return block onto the stack.

   The processor pushes a wide return block onto the wide stack if executing a 32-bit instruction (such as **WEDIT** or **WSTIX**). The processor pushes a narrow return block onto the narrow stack if executing an ECLIPSE 16-bit instruction (such as **EDIT** or **STIX**).

   The length and width of the return block depends on the type of fault that occurs and the instruction that causes it. For example, the **WEDIT** instruction uses the wide stack for temporary storage. When a fault occurs, the processor pushes the return block in addition to the temporary words that the **WEDIT** instruction requires.

   Table 5-13 lists the decimal and ASCII fault codes. The first and second columns list the code that appears in AC1. The third column lists the type of return block pushed; the following two sections describe the wide and narrow fault return blocks. The fourth and fifth columns list the instructions and conditions that caused the fault.

2. Sets the processor status register to 0.

3. Places the fault code in AC1 (bits 16-31) and the value of the program counter for the instruction that caused the fault in AC0.

4. Jumps to the fault handler through the 16-bit pointer in reserved memory location 46_8 of the current segment. Both the wide and narrow faults use the same fault pointer and handler.

Table 5-13 *Decimal and ASCII fault codes*

Code Returned in AC1		Return Block Type	Faulting Instruction	Condition
Narrow	Wide			
000000	100000	2	EDIT, WEDIT	An invalid digit or alphabetic character encountered during execution of one of the following subopcodes: DMVA, DMVF, DMVN, DMVO, DMVS.
000001	100001	1	LDIX, STIX	Invalid data type (6 or 7).
		3	WEDIT, WLDIX, WSTIX, WDMOV, WDDEC, WDINC, WDCMP, EDIT	Invalid data type (6 or 7).
000002	100002	2	EDIT, WEDIT	DMVA or DMVC subopcode with source data type 5; AC2 contains the data size and precision.
000003	100003	2	EDIT, WEDIT	An invalid opcode; AC2 contains the data size and precision.
000004	100004	1	LDI, STI, STIX, WLDI, WSTI, WSTIX, WLDIX	Number too large to convert to specified data type. |number|> (10^{16}) - 1 Number too large to convert to specified data type. Number > (10^{32}) - 1
000005	—	3	EDIT, LDI, LDIX, STI, STIX	Invalid microinterrupt return block. (Applies only to ECLIPSE interrupt-resumable instructions.)
000006	100006	1	WLSN, WLDI, LSN, LDI, LDIX, WLDIX	Sign code is invalid for this data type for this data type.
		3	EDIT, WEDIT,WDINC, WDMOV, WDCMP, WDDEC	
000007	100007	1	WLSN, WLDI, WLDIX, LSN, LDI, LDIX	Invalid digit.
		3	WDMOV, WDCMP, WDINC, WDDEC	

Wide Fault Return Blocks

Tables 5-14 through 5-16 list the contents and types of wide return blocks. After the processor pushes a wide return block, the accumulators retain their original contents, except that AC1 contains the fault code.

Table 5-14 *Wide return block for decimal data fault (type 1)*

Doubleword In Block Pushed	Contents
1	PSR (Bits 0-15 contain the processor status register; bits 16-31 contain zeros)
2	AC0 (unchanged)
3	AC1 (contains original descriptor)
4	AC2 (contains original source indicator or destination indicator for WSTI or STIX instruction)
5	AC3 (undefined)
6	CRY/PC (Bit 0 contains Carry; bits 1-31 contain 31-bit address of the decimal instruction causing the fault)

Table 5-15 *Wide return block for ASCII data fault (type 2)*

Doubleword In Block Pushed	Contents
1	PSR (Bits 0-15 contain the processor status register; bits 16-31 contain zeros)
2	AC0 (contains current value of P — byte pointer to subopcode that caused the fault)
3	AC1 (contains original descriptor)
4	AC2 (undefined)
5	AC3 (undefined)
6	CRY/PC (Bit 0 contains Carry; bits 1-31 contain 31-bit address of the WEDIT instruction causing fault)

Table 5-16 *Wide return block for ASCII data fault (type 3)*

Doubleword In Block Pushed	Contents
1	PSR (Bits 0-15 contain the processor status register; bits 16-31 contain zeros)
2	AC0 (contains original source descriptor for WDMOV and WDCMP)
3	AC1 (contains original descriptor)
4	AC2 (contains original source pointer for WDMOV and WDCMP)
5	AC3 (contains original destination pointer for WDMOV, WDCMP, WDINC, and WDDEC)
6	CRY/PC (Bit 0 contains Carry; bits 1-31 contain 31-bit address of the instruction causing the fault)

Narrow Fault Return Blocks

Tables 5-17 through 5-19 list the contents and types of narrow return blocks. After the processor pushes a narrow return block, the accumulators retain their original contents, except that AC1 contains the fault code. The tables lists the contents of the accumulators immediately before the block is pushed. Note that the only bits 16 through 31 of the accumulators are affected.

Table 5-17 *Narrow return block for decimal data fault (type 1)*

Word Number In Block Pushed	Contents
1	ACO
2	AC1 (original descriptor)
3	AC2 (original source indicator — or destination indicator for WSTI or STIX)
4	AC3 (undefined)
5	CRY/PC (Bit 0 contains Carry; bits 1-15 contain address of the decimal instruction causing the fault)

Table 5-18 *Narrow return block for ASCII data fault (type 2)*

Word Number In Block Pushed	Contents
1-4	Reserved for ECLIPSE compatibility
5	AC0 (current value of P — byte pointer to subopcode that caused the fault)
6	AC1 (original descriptor)
7	AC2 (undefined)
8	AC3 (undefined)
9	CRY/PC (Bit 0 contains Carry; bits 1-15 contain address of the decimal instruction causing the fault)

Table 5-19 *Narrow return block for ASCII data fault (type 3)*

Word Number In Block Pushed	Contents
1	AC0
2	AC1 (original descriptor)
3	AC2 (undefined)
4	AC3 (undefined)
5	CRY/PC (Bit 0 contains Carry; bits 1-15 contain address of the decimal instruction causing the fault)

Stack Faults

The processor checks for a wide stack fault after a wide stack operation, and checks for a narrow stack fault after a narrow stack operation. When a stack overflow occurs, the program overwrites the data in the area beyond the stack. When a stack underflow occurs, the program accesses incorrect information. Once detected, the processor always services the narrow or wide stack fault.

Wide Stack Fault Operations

After a wide push operation, the processor compares the contents of the wide stack pointer to the contents of the wide stack limit. If the wide stack pointer value is greater than the wide stack limit value, the processor detects a wide stack overflow fault.

After a wide pop operation, the processor compares the contents of the wide stack pointer to the contents of the wide stack base. If the wide stack pointer value is less than the wide stack base value, the processor detects a wide stack underflow fault.

You can disable wide stack overflow fault detection by loading the value 37777777777_8 into the wide stack limit register, thus the wide stack limit is larger then the wide stack pointer. You can disable wide stack underflow fault detection by loading the value 20000000000_8 into the wide stack base register.

When a wide stack fault occurs, the processor

1. Sets the wide stack pointer equal to the wide stack limit (for a wide stack *underflow* only).
2. Pushes a wide return block onto the wide stack. (See Table 5-20.)

   The return address word in the wide return block points to the next instruction that the processor executes after servicing the fault.

Table 5-20 *Wide stack fault return block*

Doubleword in Block Pushed	Contents
1	PSR (Bits 0-15 contain the processor status register; bits 16-31 contain zeros).
2	AC0
3	AC1
4	AC2
5	AC3
6	CRY/PC (Bit 0 contains Carry; bits 1-31 contain 31-bit address of the instruction causing the fault if a type 1 fault, otherwise, the address of the next instruction in the instruction stream.)

3. Sets the PSR flags (OVK, OVR, and IRES) to 0.
4. Sets bit 0 of the wide stack pointer to 0.
5. Sets bit 0 of the wide stack limit to 1.
6. Updates the wide stack pointer and the reserved memory locations for the stack parameters in the current segment.
7. Loads AC0 with the address of the instruction that caused the fault.

8. Loads AC1 with the code that describes the fault. (See Table 5-21.)

Table 5-21 *Wide stack fault codes*

AC1 Code	Meaning
000000	Overflow on every stack operation other than SAVE, WMSP, or segment crossing.
000001	Underflow or overflow would occur if the instruction were executed (pertains to WMSP, WSSVR, WSSVS, WSAVR, and WSAVS instructions.) (PC in return block refers to the instruction that caused the stack fault.)
000002	Too many arguments on a cross segment call.
000003	Stack underflow.
000004	Overflow due to a return block pushed as a result of a microinterrupt or fault.

9. Jumps to the wide stack fault handler through the 16-bit pointer in page zero of the current segment (location 14_8).

If you write a wide stack fault handler, the handler routine should

1. Determine the nature of the fault (underflow or overflow).
2. Reset bit 0 of the wide stack pointer and the wide stack limit to the original values.
3. Take other appropriate action, such as allocating more stack space or terminating the program.
4. Use a **WPOPB** instruction as the last instruction of the fault handler to return to the faulting program.

Narrow Stack Fault Operations

The narrow stack is a series of single words managed by three reserved memory words. The narrow stack supports program development and upward program compatibility for ECLIPSE 16-bit programs.

After a narrow push operation, the processor compares the contents of the narrow stack pointer to the contents of the narrow stack limit. If the stack pointer value is greater than the stack limit value, the processor detects a narrow stack overflow fault.

After a narrow pop operation, the processor compares the contents of the narrow stack pointer to 401_8. If the stack pointer value is less than 400_8 and bit 0 of the narrow stack limit is zero, the processor detects a narrow stack underflow fault.

You can disable narrow stack overflow fault detection by setting bit 0 of the narrow stack pointer to zero and bit 0 of the stack limit to one. You can disable narrow stack underflow fault detection by

- Starting the narrow stack at a location greater than 401_8.

  If the narrow stack starts at location greater than 401_8, underflow still occurs when the value of the stack pointer becomes less than 400_8. The processor can detect underflow if a program pops enough words from the narrow stack to cause the narrow stack pointer to wraparound.

- Setting bit 0 of either the narrow stack pointer or the narrow stack limit to one.

  If bit 0 of the narrow stack pointer or narrow stack limit is set to one, either all or part of the stack may reside in reserved memory page zero ($0-377_8$), or the stack may underflow into reserved memory page zero without interference from the narrow stack fault handler.

When a narrow stack fault occurs, the processor

1. Sets the narrow stack pointer equal to the narrow stack limit (for stack *underflow* only).

2. Sets bit 0 of the narrow stack pointer to zero and bit 0 of the narrow stack limit to one.

   Thus, the narrow stack limit is (temporarily) larger than the narrow stack pointer, which disables overflow fault detection.

3. Pushes a narrow return block onto the narrow stack. (See Table 5-22.) Note that the only bits 16 through 31 of the accumulators are affected.

   The return address word in the narrow return block points to the next instruction the processor executes after servicing the fault.

4. Jumps to the narrow stack fault handler through the 16-bit pointer in page zero of the current segment (location 43_8).

Table 5-22 *Narrow stack fault return block*

Word Number In Block Pushed	Contents
1	AC0
2	AC1
3	AC2
4	AC3
5	CRY/PC (Bit 0 contains Carry; bits 1-15 contain the address of the instruction causing the fault.)

If you write a narrow stack fault handler, the handler should

1. Determine the nature of the fault (underflow or overflow).

2. Reset bit 0 of the narrow stack pointer and the narrow stack limit to their original values.

3. Take other appropriate action, such as allocating more stack space or terminating the program.

4. Use a **POPB** instruction as the last instruction of the fault handler to return to the faulting program.

End of Chapter

6

Queue Management

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

An entry in a queue is called a *data element*. The beginning and end of a queue are called the *head* and the *tail*, respectively. In a typical first in, first out (FIFO) queue, data elements are enqueued at the tail and dequeued at the head.

One advantage of using a queue rather than a single-threaded list is that queue data elements refer to the data elements that precede *and* follow them. Data elements can be added anywhere in the queue, not just at the tail. Conversely, data elements can be removed anywhere in the queue, not just at the head.

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 addresses, 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 refer to 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 -1, then that data element is at the tail of the queue. If a data element's backward link contains -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.

NOTES: *Removing a data element from the queue sets both forward and backward links to -1. If the same data element is removed twice, results are indeterminate.*

If a data element's forward or backward link contains self-directed pointers (a pointer to itself), a queue instruction will loop until interrupted.

A data element may contain user information (optional) as well as the forward and backward links. This user information can precede or follow the forward and backward links. Figure 6-1 shows the structure of a data element with user data either preceding or following the links. The user determines the structure and the meaning of the information.

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

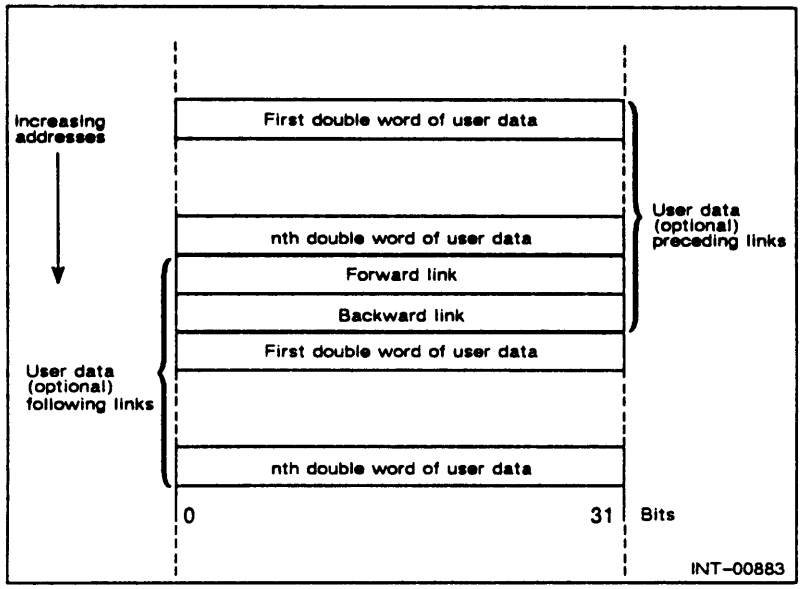


Figure 6-1 *Data elements with user data*

Queue Descriptor

Each queue uses a *queue descriptor* that indicates the current head and tail of the queue. A queue descriptor is two doublewords. The first doubleword 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. (See Figure 6-2.)

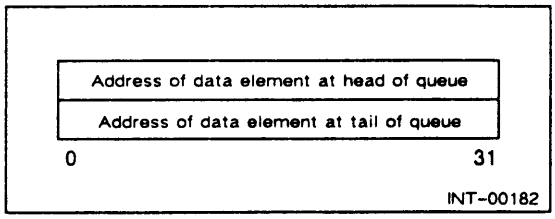


Figure 6-2 *Format of queue descriptor*

Setting Up and Modifying a Queue

To define an empty queue, create a queue descriptor that contains -1 in both of its pointers. To place a data element into the empty queue, load the address of the data element into both doublewords of the queue descriptor (indicating a one-element queue) and load -1 into the data element's forward and backward links. To add or remove 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 act as reference points that the processor uses to add the data element at the right point or to remove the appropriate data element.

Queue Examples

The examples following demonstrate how you can form queues, how you can add and remove data elements in the queue, and how the processor updates the various links and descriptors. Note that in each example, the user data follows the links.

Queue Descriptor of an Empty Queue

Figure 6-3 shows the queue descriptor for an empty queue.

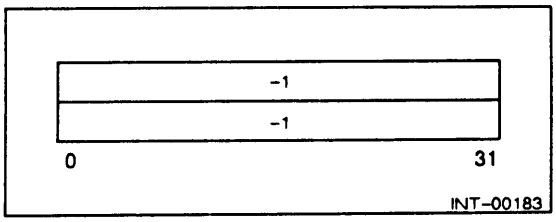


Figure 6-3 *Queue descriptor for an empty queue*

Adding a Data Element into an Empty Queue

Figure 6-4 illustrates how the processor adds a data element (at location A) into an empty queue. After adding the data element, the processor updates 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 -1. The last word contains the backward link of -1.

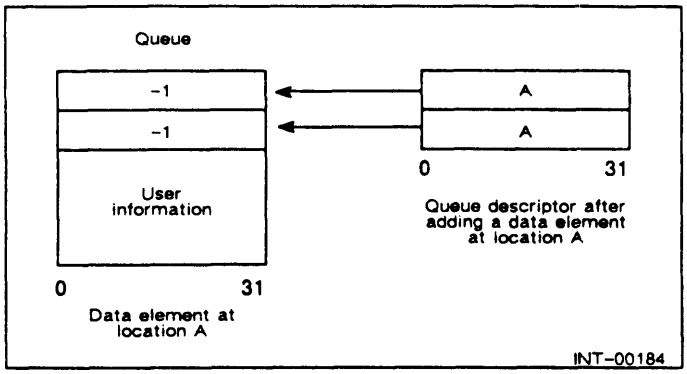


Figure 6-4 *Data element added to an empty queue*

Adding a Data Element at the Head of a Queue

Figure 6-5 illustrates how the processor adds a data element (at location B) at the head of the queue before data element A. After adding the data element, the processor updates the queue descriptor to refer to the new head. It also changes the backward link of data element A to refer to the preceding data element (B). The links of data element B show that it is the head of the queue and that element A follows it.

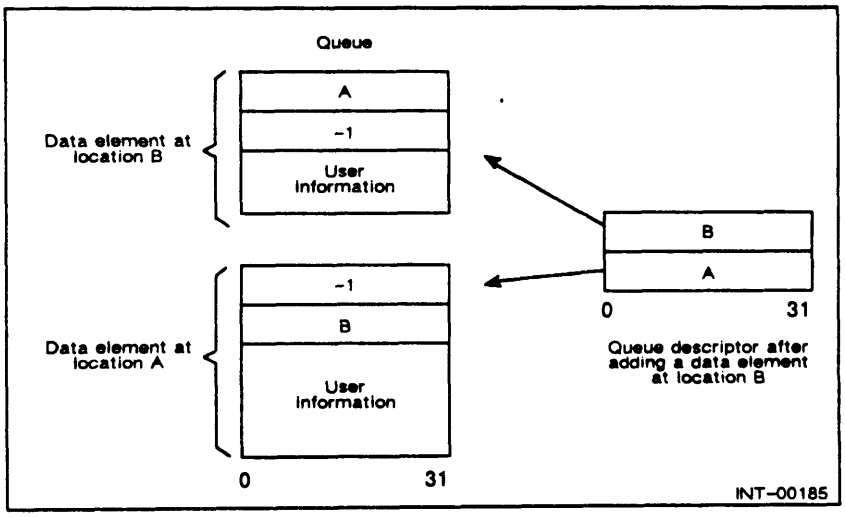


Figure 6-5 *Data element added at head of queue*

Adding a Data Element at the Tail of a Queue

Figure 6-6 illustrates how the processor adds a data element (at location C) at the tail of the queue, after data element A. 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 queue's new head.

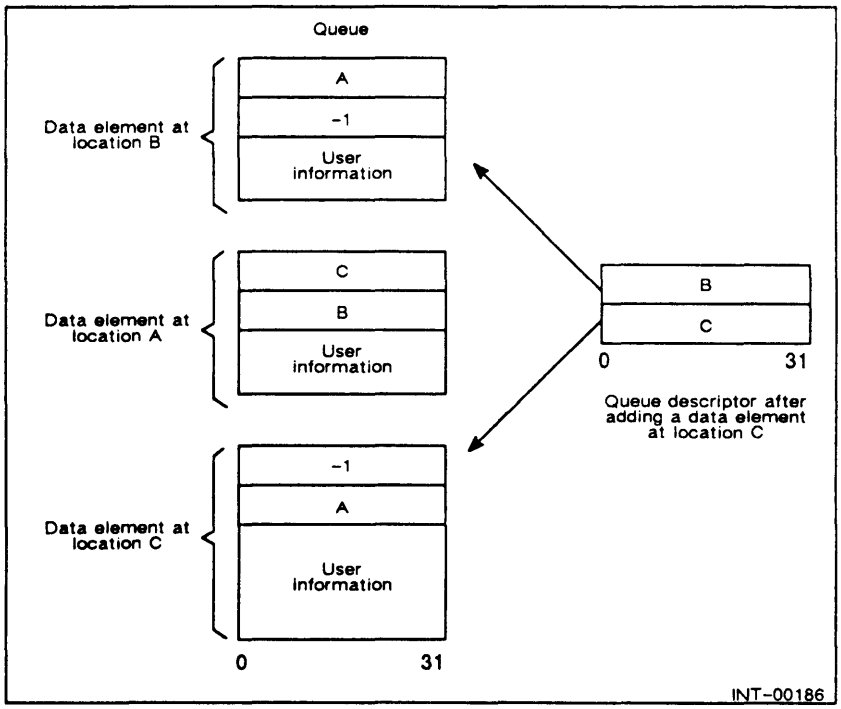


Figure 6-6 *Data element added at tail of queue*

The example below shows how an **ENQT** instruction may be used in a programming call.

```
;This subroutine moves an element from one queue to the end of
;another. It is the responsibility of the caller to set the
;transition bit, if necessary.
;
;Calling conventions: XJSR      QMOVE
;                               <return>
;      AC0 = Source queue descriptor address
;      AC1 = Address of element to be moved
;      AC2 = Destination queue descriptor address
      QMOVE:      WSSVS       0          ;
                  NLDAI       QLOCK,3    ;Queue descriptor lock offset.
;First, handle the source queue.
      QLP1:       WSZBO       0,3        ;Can we lock source?
                  WBR         QSPIN1     ;No, wait.
                  DEQUE                  ;Remove from source.
                  NOP                    ;No-op.
                  WBTZ        0,3        ;Unlock source lock.
;Now, handle the destination queue.
      QLP2:       WSZBO       2,3        ;Can we lock destination?
                  WBR         QSPIN2     ;No, wait.
                  WMOV        2,0        ;Destination
                                         ;descriptor address.
                  WMOV        1,2        ;Element to be added.
                  WADC        1,1        ;At the end.
                  ENQT                   ;
                  NOP                    ;No-op.
                  WBTZ        0,3        ;Unlock destination lock.
;All done -- lights on and return.
                  WRTN                   ;
```

```
;Spin lock for the source queue.
      QSPIN1:     WSZB        0,3       ;Source unlocked yet?
                  WBR         QSPIN1    ;No, wait.
                  WBR         QLP1      ;Try to get source lock.
;Spin lock for the destination queue.
      QSPIN2;     WSZB        2,3       ;Destination unlocked yet?
                  WBR         QSPIN2    ;No, wait.
                  WBR         QLP2      ;Try to get destination lock.
```

Removing a Data Element

Figure 6-7 illustrates how the processor removes data element B. After removing the data element, the processor 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.

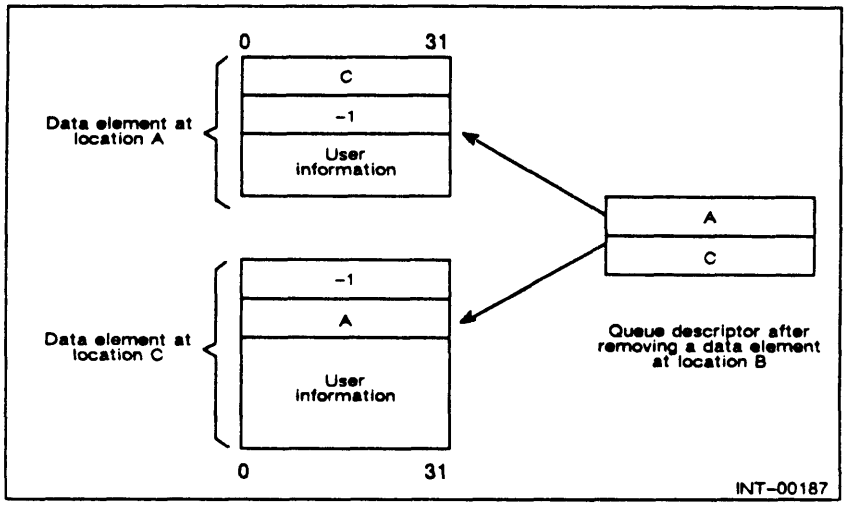


Figure 6-7 *Data element removed*

The following example shows how a **DEQUE** instruction may be used in a programming call.

```
;This subroutine removes an element from a linked list queue. It is
;the responsibility of the caller to set the transition bit, if
;necessary.
;
;Calling conventions: XJSR      PDEQ
;
;      AC1 = Queue descriptor address
;      AC2 = Element to be queued
      PDEQ:       WSSVR       0         ;Save return block on stack.
                  WMOV        1,0       ;Move queue address to AC0.
                  WMOV        2,1       ;Move element to be removed
                                        ;to AC1.
                  NLDAI       QLOCK,2   ;Queue descriptor lock offset.
      PDEQ1:      WSZBO       0,2       ;Can we lock it?
                  WBR         PSPIN     ;No, wait.
                  DEQUE                 ;
```

```
                NOP                     ;No-op.
                WBTZ        0,2         ;Unlock it.
                WRTN                    ;And return to calling
                                        ;program.
PSPIN:          WSZB        0,2         ;Unlocked yet?
                WBR         PSPIN       ;No, wait.
                WBR         PDEQ1       ;Yes, grab it!
```

Queue Instructions

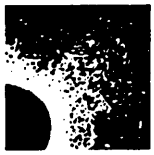
Table 6-1 lists the instructions for manipulating queues. **ENQH** and **ENQT** instructions add data elements to queues, and the **DEQUE** instruction removes data elements. The remaining instructions perform queue or queue-like searches.

NOTE: *The* **WMESS** *instruction is a powerful instruction for indivisibly updating a queue or automatically updating a database without software locking.*

Table 6-1 *Queue instructions*

Instruction	Operation
ENQH	Add a data element towards the queue head.
ENQT	Add a data element towards the queue tail.
DEQUE	Remove a queue data element.
NBStc	Narrow search queue backward (towards head); 16-bit test condition
NFStc	Narrow search queue forward (towards tail); 16-bit test condition
WBStc	Wide search queue backward (towards head); 32-bit test condition
WFStc	Wide search queue forward (towards tail); 32-bit test condition
WMESS	Wide mask, skip and store if equal

End of Chapter



7

Graphics Management

The Graphics Instruction Set (GIS) performs high-speed graphic functions, directly supporting windowing systems in which several programs share one bitmap. GIS allows each program to have both a physical bitmap (such as a display screen) and a virtual bitmap, which is used to store regions of the screen that have been taken over by another program. GIS instructions can switch from one bitmap to the other in midstream; the user's program need not know which bitmap is being used; and the operating system does not need to oversee every drawing operation to prevent one program from overwriting another's data.

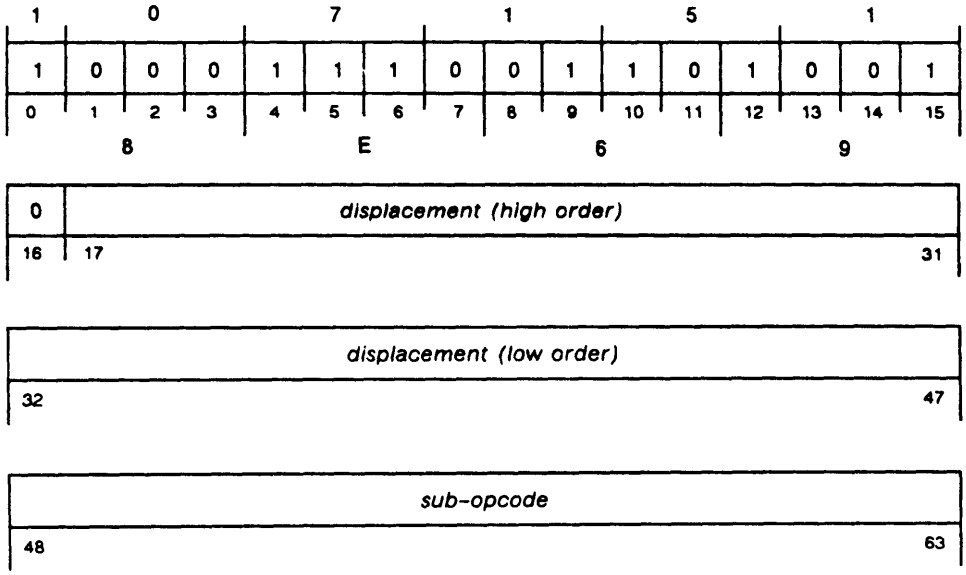
The Graphics Instruction Set, as presented in this manual, is Data General's GIS II for ECLIPSE MV/Family systems. Information on GIS I may be found in the *Data General 4000-Class Integrated Systems, Functional Characteristics* manual (DGC No. 014-001066).

Graphics Instruction Set

The Graphics Instruction Set includes both privileged and nonprivileged instructions. Privileged GIS instructions are used to maintain the various GIS databases discussed later in this chapter. Nonprivileged instructions do the following:

- Read or write a single pixel.
- Draw a line, or a series of connected lines.
- Fill a rectangular region of the bitmap with a solid color.
- Copy a rectangular region from one place to another.
- Write a text character or other symbol onto an image.
- Change a drawing attribute.

This section provides some general information about GIS instructions. The *Instruction Dictionary* describes each instruction in detail. All GIS instructions are four words (64 bits) long, and have the following format:



Bits 0–15 of the instruction identifies this instruction as a GIS instruction; these bits are always 107151_8 ($8E69_{16}$).

Bits 16–47 (*displacement*) are a routine's address that you must provide to handle instruction traps. These traps may be used to simulate additional instructions. If your system currently does not support GIS microcode, the displacement is the program-counter-relative address of a runtime routine that emulates the GIS instruction functions. Bit 16 is always 0.

Bits 48–63 (*sub-opcode*) identify the particular GIS operation to perform. The processor accepts only those values listed in Table 7–1. Other values will produce an instruction trap.

Table 7-1 *GIS instructions*

Instruction	Sub-opcode Octal [hex]		Type	Description
WGBITBLT	30	[18]	Nonprivileged	Copies pixels to a form or location
WGCHRBLT	31	[19]	Nonprivileged	Writes a character into a form
WGLDCURS	34	[1C]	Privileged	Writes cursor block to cursor descriptor memory
WGLFORM	20	[10]	Privileged	Loads a form into form cache memory
WGPFORMS	21	[11]	Privileged	Removes form(s) from form cache
WGPLINE	27	[17]	Nonprivileged	Draws line segment(s) in a form
WGRDATTR	32	[1A]	Nonprivileged	Reads attributes of a form
WGRDPAL	22	[12]	Privileged	Reads a palette register
WGRDPIXL	24	[14]	Nonprivileged	Reads a pixel from a form
WGRFLOOD	26	[16]	Nonprivileged	Sets rectangle color
WGWRATTR	33	[1B]	Nonprivileged	Writes attribute of a form
WGWRPAL	23	[13]	Privileged	Writes a palette register
WGWRPIXL	25	[15]	Nonprivileged	Writes a pixel into a form

Certain guidelines are consistent for each GIS instruction:

- AC1 contains a form ID. The form ID, created by the operating system, refers to the *form descriptor* of the form in which the instruction draws.
- AC2 contains a pointer to an instruction packet (if such a packet is defined for that instruction).
- AC3 is not used by GIS instructions. This allows efficient invocation of GIS subroutines from high-level languages that use AC3 to hold the frame pointer.
- Attributes (such as line style, foreground color, etc.) are associated with the form being operated on.
- The microcode for GIS instructions is designed to optimize its speed of execution if certain conditions are met. General guidelines for faster execution of all GIS instructions include:
  - Using combination rules that do not require *both* source and destination pixels. (In the case of a virtual bitmap, the destination pixel will not be read from memory to perform the combination if it is not needed.)
  - Working with a rectangle list containing a single rectangle.
- Any parameter, which requires an unsigned number, should have bit 0 of the 32-bit value equal to 0. Entering a negative value (bit 0 equal to 1) where an unsigned number is called for, will give undefined results.

GIS instructions operate on forms. A *form descriptor* describes the form itself and points to related databases, such as attributes and rectangle and cursor descriptors. The following sections describe forms, data structures, the form cache, interrupts, and GIS fault handling.

Forms

The *form* is the basic unit of pixel space, the "paper" on which a picture is drawn. The form is the object upon which all GIS operations are performed. Although the data structures that define a form are complex, they combine to create the appearance of a simple rectangular set of pixels.

Forms are managed by the operating system. Screens and bitmaps are considered to be a system resource, like disk space, that must be shared by many programs and users. The privileged GIS instructions create and delete forms, change their sizes, and move them around on the screen.

Every GIS drawing instruction refers to a specific form with a *form ID* in AC1. When a user program creates a form, the operating system must assign a form ID and pass it to the program.

The properties of a form are specified in a block of data called a *form descriptor*. Figure 7-1 shows how the descriptor ties together all the data structures that define the form. The section, "GIS Data Structures," describes the various properties of the form in detail.

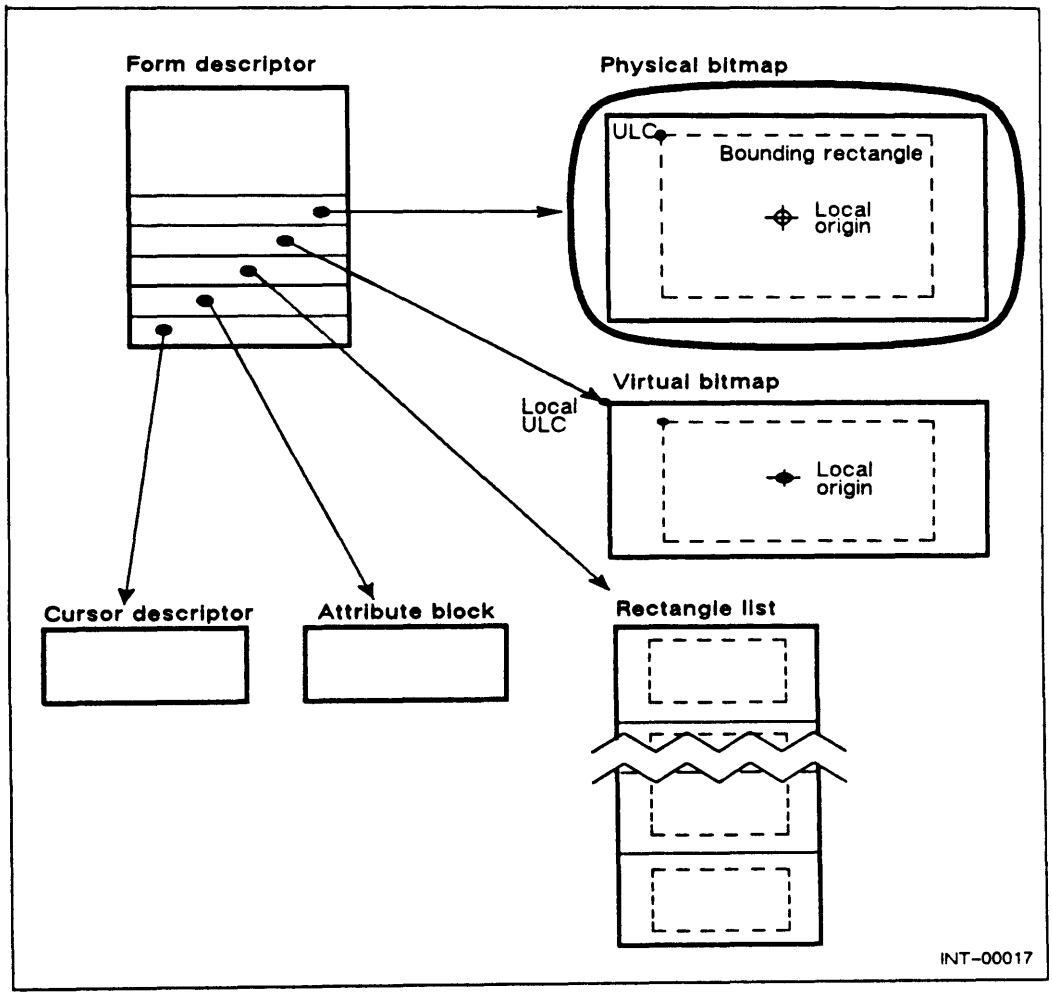


Figure 7-1 *Form data structures*

Forms and Bitmaps

Every form is associated with one or two bitmaps — a physical bitmap and/or a logical bitmap. A *physical bitmap* is part of a display device, and changes to this bitmap are immediately visible on the screen. A *virtual bitmap* is placed in main memory; and changes to a virtual bitmap are not visible until the pixels are copied to a physical bitmap. The origin of a bitmap is always the upper-left corner of the bitmap (coordinates 0, 0).

Virtual bitmaps have two main purposes: to allow pictures to be created in main memory, and moved later to the screen for display; to provide hardware support of windowing systems, in which several forms may overlap on the screen. The size of a virtual bitmap must be equal to or greater than the size of its associated form. The depth of a virtual bitmap is limited only by the virtual address space available within the segment (to a maximum of $2^{28} - 1$).

Figure 7-2 illustrates the following example.

> A program creates a form, called F1, that has both a physical and a virtual bitmap. Initially the form is fully visible, and all drawing takes place on the physical bitmap. Then another program creates a form, F2, whose physical bitmap overlaps part of the physical bitmap of F1. The operating system must then handle this conflict by copying the obscured part of F1 to the F1 virtual bitmap, setting up its data structures to indicate the change.
>
> Since the GIS instructions know that F1 is now divided into several pieces, all drawing operations automatically switch from one bitmap to the other. The first program is never aware that part of its form is now on the virtual bitmap.

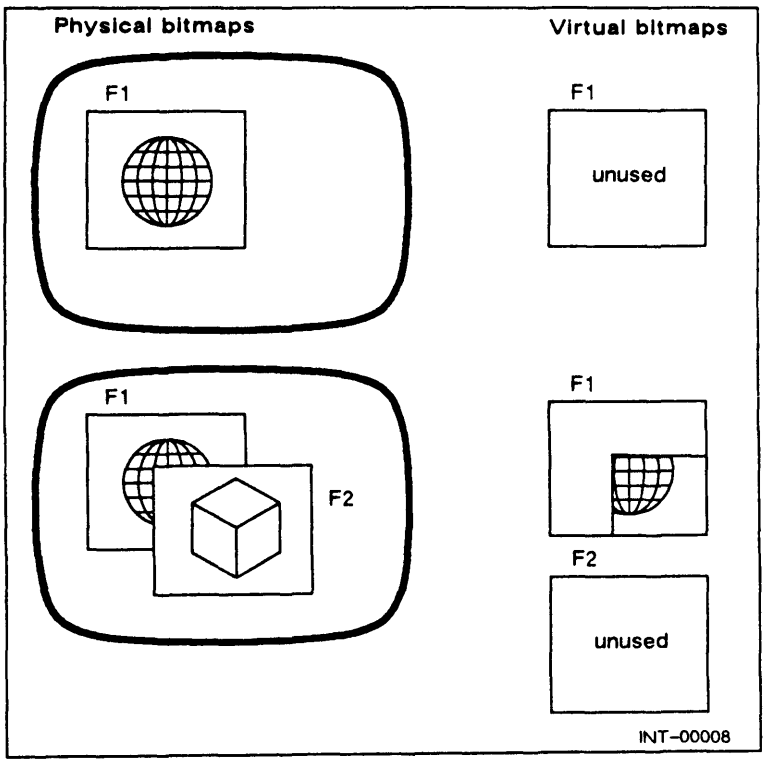


Figure 7-2 *Windowing with virtual bitmaps*

The data structure that keeps track of this forms division is the *rectangle* . which consists of one or more *rectangle descriptors*. Each descriptor gives the si d position of a rectangle, and tells which bitmap is on.

The use of the rectangle list is illustrated in Figure 7-3. F2 is completely visible, so its rectangle list consists of a single descriptor. The descriptor specifies that the rectangle is on the physical bitmap. F1 is partially obscured by F2. The visible portion of F1 is an irregular L shape. In order to describe this arrangement, the operating system must handle the L as two rectangular pieces. So the rectangle list for F1 contains three descriptors: two pointing to the physical bitmap; one pointing to the virtual bitmap.

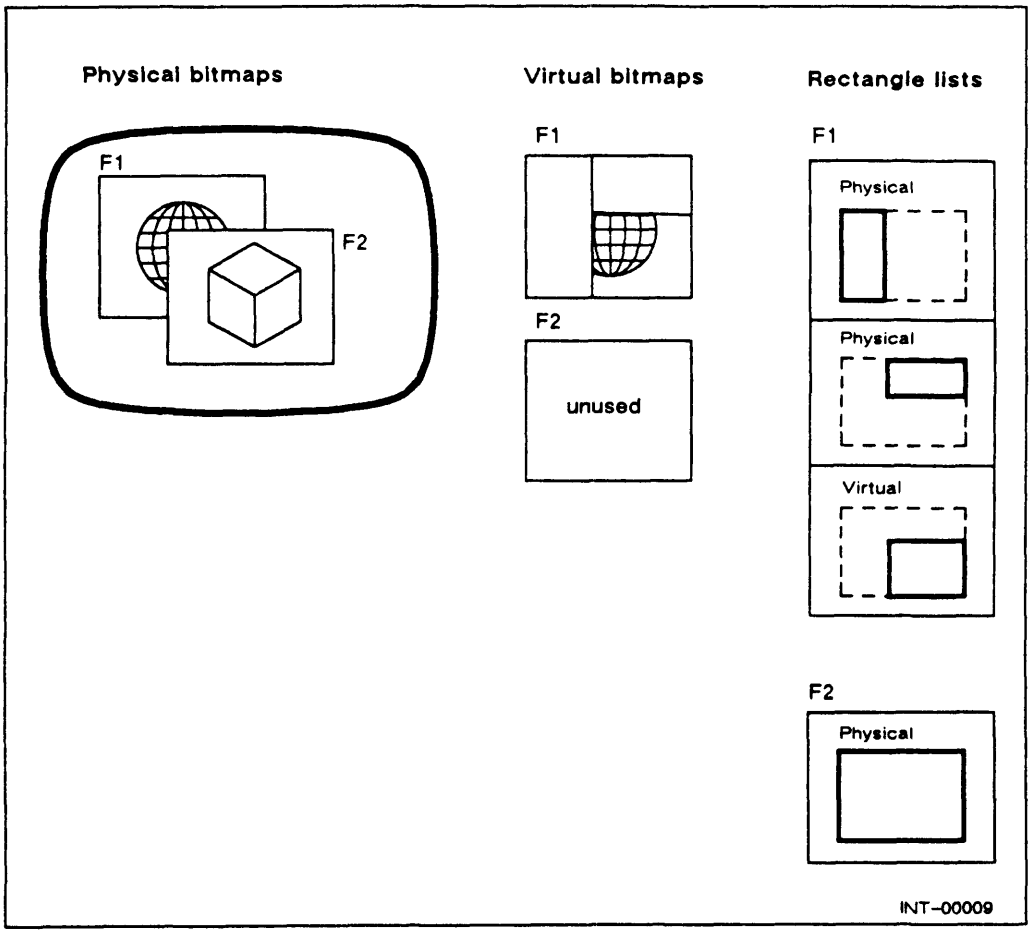


Figure 7-3 *Use of rectangle list*

Local Origin

Every form has a *local origin* used by all GIS instructions that refer to the form. This allows each form to have its origin at the upper-left corner (ULC), center, or any convenient location (the local origin does not necessarily have to lie within the form).

Bounding Rectangle

All rectangles are defined by a ULC and a size (X-extent and Y-extent). Every form has a *bounding rectangle* that specifies the usable range of values for X and Y coordinates in the form. The bounding rectangle is not a region of memory; it is an abstract entity that

is defined by the coordinates of its ULC and its X-extent and Y-extent. All GIS instructions apply *clipping*, so that no instruction may read or modify any pixels outside the bounding rectangle.

The bounding rectangle encompasses all rectangles on the rectangle list for the form. The ULC of the bounding rectangle is equal to the ULC of the uppermost, lefthand rectangle in the rectangle list. Pixels with coordinates outside the bounding rectangle are never drawn.

A rectangle that is write inhibited may be accessed by read operations, but not write operations. This rectangle is used to implement clip rectangles on a form.

Some instructions execute faster if the ULC starts at the leftmost pixel of an even doubleword. Virtual bitmap doubleword addresses are calculated as follows:

- The LOC_VIRT_Y value in the form descriptor is subtracted from the Local_Y coordinate (in the instruction packet). The LOC_VIRT_X value in the form descriptor is subtracted from the Local_X coordinate. The recomputed X and Y coordinates now address a virtual global coordinate.
- The global Y coordinate, X pitch, and Y pitch are multiplied together.
- This value is added to the product of the global X coordinate and the X pitch. If the modulus 32 of this value equals zero, then the coordinates point to the leftmost pixel of the doubleword.
- The integer portion of this intermediate value is divided by 32 and then added to the bitmap's base address.

This procedure, expressed as an equation, is:

```
Doubleword address =
   integer portion[((local_Y - LOC_VIRT_Y) * X_pitch * Y_pitch
                  + (local_X - LOC_VIRT_X) * Y_pitch) / 32]
   + bitmap base address
```

Coordinate System

GIS uses three types of coordinates: user, virtual, and physical. The user coordinates are local; the virtual and physical coordinates are global. The calculation of the final coordinates is dependent on whether the bitmap accessed is virtual or physical.

- The user coordinates are X and Y values contained in the drawing packets of all GIS instructions. The acceptable user coordinates are 32-bit signed integers. Within this chapter these coordinates are specified as Local_X and Local_Y.
- The virtual coordinates are the user X and Y values transformed onto the virtual bitmap. These coordinates are positive 31-bit integers relative to the ULC of the virtual bitmap. The values LOC_VIRT_X and LOC_VIRT_Y, contained in the form descriptor, are used in the conversion of the local coordinates to virtual global coordinates.
- The physical coordinates are the user X and Y values transformed onto the physical bitmap. These coordinates are positive 31-bit integers relative to the ULC of the physical bitmap. The values VIRT_PHY_X and VIRT_PHY_Y, contained in the form descriptor, are used in the conversion of the virtual global coordinates to physical global coordinates.

Figure 7-4 illustrates the conversion of coordinates from one type to another using form F1 from the previous example. The bounding rectangle upper-left corner (BR_ULC) is mapped to the global virtual bitmap using the local-to-virtual bitmap values in the form descriptor (LOC_VIRT_X and LOC_VIRT_Y). LOC_VIRT_X and LOC_VIRT_Y are the X and Y coordinates, respectively, of the virtual bitmap's ULC relative to the origin of the form. The coordinates for the two rectangles which are on the physical bitmap are then calculated using the virtual-to-physical bitmap coordinates in the form descriptor (VIRT_PHY_X and VIRT_PHY_Y). VIRT_PHY_X and VIRT_PHY_Y are the X and Y coordinates, respectively, of the virtual bitmap's ULC relative to the origin of the physical bitmap.

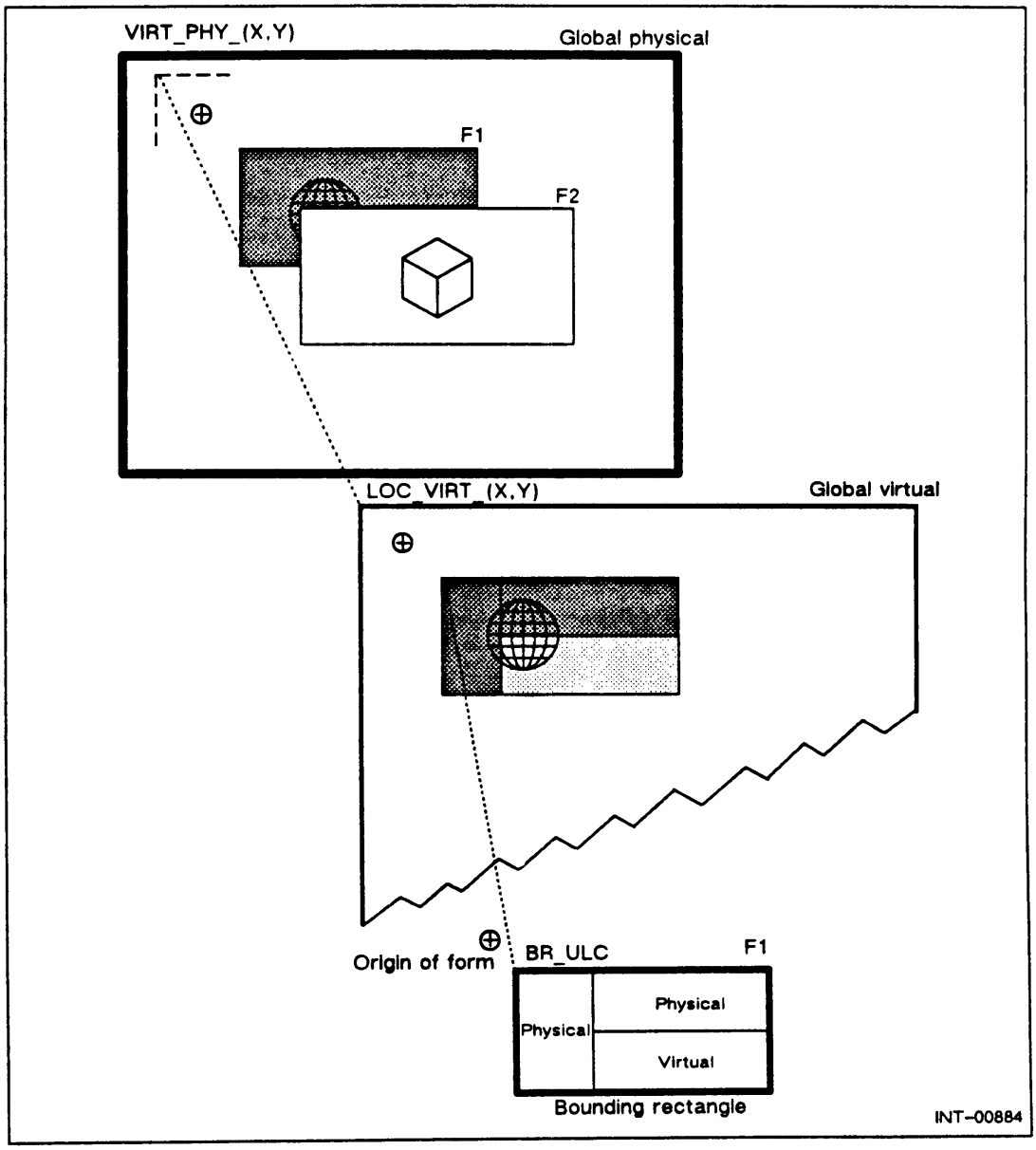


Figure 7-4 *Coordinate conversions*

GIS Data Structures

The data structures used by GIS are both powerful and very generalized. These structures include form descriptors and their related databases: rectangle descriptors, form blocks, cursor descriptors, and color descriptors.

The following properties must be true for all data structures (except virtual bitmaps) used by GIS instructions:

- They must always be resident in physical main memory.
- Except for bitmaps, they must not cross page boundaries.
- They must be doubleword aligned; i.e., the first word of every data structure must have an even address.
- The maximum size of any data structure is one page.

Most of the addresses stored in these data structures are physical addresses.

Form Descriptor

The *form descriptor* describes the form itself and points to related databases, such as attributes and cursor descriptor. The operating system creates a form descriptor in response to a user request.

The form descriptor block is a doubleword table of signed and unsigned 32-bit integers. All unsigned numbers in the form descriptor should be 31-bit integers or signed 32-bit positive integers (bit 0 is always 0). Entering a negative value (bit 0 equals 1), where an unsigned number is called for, will cause undefined results. The form descriptor's contents are listed in Table 7-2 (the "Mnemonic" column is for reference within this text only). The form descriptor's properties and related databases are detailed in the sections that follow.

If the form has only a physical bitmap associated with it, the virtual bitmap portion of the form descriptor will be filled with zeros. These "empty" bitmaps will never be used as long as no rectangles on the form's rectangle list refer to the nonexistent bitmap. If a rectangle does refer to a nonexistent bitmap, the results are undefined.

Drawing will not occur if any of the following conditions (relating to the destination form) exist:

- Both the form mask (in the form descriptor) and the operation mask (in the attribute block) equal zero.
- No writable rectangle exists.
- Combination rule number five is used (destination moved to destination).
- Both foreground and background pixel drawing is suppressed (Bits 0 and 1 of the LINE_CTRL doubleword in the attribute block equal zero). This applies only to the **WGPLINE** and **WGCHRBLT** instructions.
- Any other combination of items in the form descriptor that would cause the instruction to perform no operation.

If a GIS instruction is operating on a particular form descriptor, changing any information, other than the following, will produce undefined results:

- Global ULC (VIRT_PHY_X, VIRT_PHY_Y)
- Local ULC (LOC_VIRT_X, LOC_VIRT_Y)
- Rectangle bits (FLAGS)
- Rectangle list (RECT_LIST)

If any of these values change, the operating system should increment the form-generation number in the form descriptor.

Table 7-2 *Form descriptor contents*

Double Word #	Mnemonic	Contents
1	LENGTH	Number of 16-bit words in this form descriptor (unsigned).
2	ATTR_BLK	Physical address of the attribute block (unsigned).
3	FORM_GEN	Form generation number (unsigned). This value should be zero when the form is first created and incremented by the operating system every time the form is modified. This value is used by the processor for comparison with a value in the long context block.
4	BR_ULC_X	X coordinate (signed) of the bounding rectangle's ULC (with respect to the local origin of the form).
5	BR_ULC_Y	Y coordinate (signed) of the bounding rectangle's ULC (with respect to the local origin of the form).
6	BR_EXT_X	Width of the bounding rectangle in pixels (unsigned).
7	BR_EXT_Y	Height of the bounding rectangle in pixels (unsigned).
8	VIRT_PHY_X	X coordinate (signed) of the virtual bitmap's ULC, relative to the origin of the physical bitmap. This value and VIRT_PHY_Y are used to convert the virtual global coordinates to physical global coordinates.
9	VIRT_PHY_Y	Y coordinate (signed) of the virtual bitmap's ULC, relative to the origin of the physical bitmap.
10	LOC_VIRT_X	X coordinate (signed) of the virtual bitmap's ULC, relative to the origin of the form. This value and LOC_VIRT_Y are used to convert the local X and Y coordinates to virtual global coordinates.
11	LOC_VIRT_Y	Y coordinate (signed) of the virtual bitmap's ULC relative to the origin of the form.
12	FLAGS	Flag bits. The bits and their interpretations are

Bit	Setting	Description
0	1	The rectangle list contains at least one rectangle on the physical bitmap.
	0	The rectangle list contains no rectangles on the physical bitmap.
1	1	The rectangle list contains at least one rectangle on the virtual bitmap.
	0	The rectangle list contains no rectangles on the virtual bitmap.
2	1	The rectangle list contains at least one rectangle not on either the virtual bitmap or the physical bitmap.
	0	The rectangle list contains all rectangles that are on either the virtual bitmap or the physical bitmap.
3	1	The rectangle list contains at least one write-inhibited rectangle.
	0	The rectangle list contains no write-inhibited rectangles.
4	1	Only one rectangle is on the rectangle list.
	0	Multiple rectangles are on the rectangle list.
5	1	When writing pixels, replicate the low-order byte (palette index) three times to fill a 3-byte pixel. (This allows programs written for 8-bit pixels to run on devices having 24-bit pixels.)
6-31		Reserved for future use; should be set to 0.

(continued)

Table 7-2 *Form descriptor contents (concluded)*

Double Word #	Mnemonic	Contents
13	FORM_MASK	Form mask bits (unsigned). Defines the pixel depth of the bitmaps associated with a form. For a bitmap with n bits per pixel, the low-order n bits of the FORM_MASK are set to either 1 or 0. The remaining bits in the FORM_MASK must be set to 0. When a GIS instruction executes, the FORM_MASK is logically ANDed with the Operation Mask (in the Attribute Block) to produce a mask that determines which bits within a pixel should be operated upon.
14	RECT_LIST	Physical address of the start of the form's rectangle list (unsigned).
15	CURSOR_DESC	Physical address of a cursor descriptor (unsigned).
16	DEV_TYPE	Device type of the physical bitmap (unsigned). The individual bits and their interpretations are **Bit Setting** **Description** 0 Indicates presence of a video board. 0 Video board exists. The physical bitmap portion of the form descriptor contains valid information. 1 No video board exists. The physical bitmap portion of the form descriptor is invalid. The remaining bits in the Device Type doubleword are ignored. 1-12 Reserved and should be set to 0. 13-15 Number of bits per pixel allowed: **Value (octal)** **Number of bits per pixel (decimal)** 0 1 1 2 2 4 3 2, 4, or 8 5 32 16-25 Reserved and should be set to 0. 26-31 Internal pixel transfer value. Indicates the maximum number of pixels to be transferred. For example, 10_8 indicates 8 pixels per transfer; 20_8 indicates 16 pixels; 40_8 indicates 32 pixels.
17	P_BMAP_ADDR	Microcode ID (unsigned) for video board (base address).
18	P_X_PITCH	Number of bits per pixel in the physical bitmap; also known as "X pitch." This unsigned number is a power of 2 in the range 1-32.
19	P_Y_PITCH	Number of pixels per line for the physical bitmap, also known as "Y pitch." This unsigned number must be a power of 2.
20	P_LOG2_XPITCH	Base 2 logarithm of the X pitch of the physical bitmap (unsigned).
21	P_LOG2_YPITCH	Base 2 logarithm of the Y pitch of the physical bitmap (unsigned).
22	V_BMAP_ADDR	Logical address of the start of the virtual bitmap memory (this unsigned address must be doubleword aligned).
23	V_X_PITCH	Number of bits per pixel in the virtual bitmap, also known as "X pitch." This unsigned number must be a power of 2 in the range 1-32.
24	V_Y_PITCH	Number of pixels per line for the virtual bitmap, also known as "Y pitch." This unsigned number must be a power of 2.
25	V_LOG2_XPITCH	Base 2 logarithm of the X pitch for the virtual bitmap (unsigned).
26	V_LOG2_YPITCH	Base 2 logarithm of the Y pitch for the virtual bitmap (unsigned).

Form Mask

The *form mask* (FORM_MASK) in the form descriptor is a value that specifies which bits in a pixel can be accessed by drawing operations. For example, if you set the form mask to 1, only the low order bit of any pixel can be modified. Also, when a pixel is read by a **WGRDPIXL**, **WGCHRBLT**, or **WGBITBLT** instruction, only the bits enabled by the mask will be read; the instruction will read zeros for all other bits.

The form mask is used to implement *palette sharing*, a technique that helps programs to share the display without destroying each other's data. The following example explains the use of palette sharing.

> A graphics workstation has 256 colors available. You want to run two concurrent programs, called P1 and P2, each with its own form. Each program needs 16 colors, so the operating system must provide a separate set of 16 palette registers to each program.
>
> The operating system sets up both form descriptors with a value of 15 (00001111_2) in the form mask. This permits each program to use only the low-order four bits of its pixels. In addition, the operating system uses the **WGRFLOOD** instruction to set all pixels in the P1 form to 0, and to set all pixels in the P2 form to 16 (00010000_2).
>
> The setting of the form mask ensures that neither program can modify the upper four bits of its pixels. Thus, both programs can use pixel values from 0 to 15, but only P1 actually has these values in the bitmap. P2 "thinks" that it is using values from 0 to 15, but it is really using values from 16 to 31, because the operating system has preset the high-order bits of all pixels in the form. The form mask prevents both programs from modifying the high-order bits, so each can use only its assigned range of values.

Rectangle Descriptor

Although the user perceives a form as a unit, in reality it may be divided into many pieces. For convenience, each piece is a rectangle. A *rectangle descriptor* describes one of the set of rectangles that make up a form. A rectangle descriptor indicates whether that area of the bounding rectangle is on a physical bitmap, a virtual bitmap, or on neither bitmap. Note that the rectangle descriptor allows you to declare that a rectangle is on neither bitmap. This allows a program to save some execution time by updating only the parts of the form that are visible to the user.

Each form has a structure called the *rectangle list* (RECT_LIST in the form descriptor) that is used to keep track of which bitmap is used for various parts of the form. The list consists of one or more rectangle descriptors. Each descriptor gives the size of a rectangle, and tells which bitmap it is on. The rectangle descriptor, as shown in Table 7-3, consists of six unsigned 32-bit integers.

The collection of rectangles that makes up an entire form is called a *tiling* of that form. Certain constraints are placed on a tiling. Rectangles:

- may not overlap;
- must completely tile the bounding rectangle for the form; and
- may not lie outside the form.

If any of these conditions are violated, undefined results will occur.

Table 7-3 *Rectangle descriptor contents*

Double Word #	Mnemonic	Contents
1	NEXT	Physical address of the next rectangle descriptor in the list, or -1 if it is the last one.
2	FLAGS	Flags bits. The individual bits and their interpretations are Bit — Meaning when 1 0 — The rectangle is on the physical bitmap. 1 — The rectangle is on the virtual bitmap. 2 — The rectangle is not on any bitmap. 3 — The rectangle is write inhibited. 4-31 — Reserved for future use; should be set to zero. NOTE: *Bits 0, 1, and 2 are mutually exclusive.*
3	ULC_X	X coordinate of the rectangle's ULC with respect to the local origin of the form.
4	ULC_Y	Y coordinate of the rectangle's ULC with respect to the local origin of the form.
5	EXT_X	Width of the rectangle in pixels.
6	EXT_Y	Height of the rectangle in pixels.

Form Attributes

Values such as foreground color and line style are stored in the *attribute block*, pointed to by the ATTR_BLK doubleword in the form descriptor. Initially, the attribute block is filled with a set of default values which is loaded into the forms cache by the Load Forms instruction (**WGLFORM**). All of the attribute block values can then be examined with the Read Attribute (**WGRDATTR**) instruction and modified with the Write Attribute (**WGWRATTR**) instruction; these instructions specify an index number for an attribute in an accumulator. Additionally, the packet for the Draw Polyline (**WGPLINE**) instruction may contain values which supersede those in the attribute block. If an attribute is changed while a GIS instruction is operating on that form, the results are undefined.

An attribute block consists of unsigned 32-bit integers created when a form descriptor is created. The contents of the attribute block are summarized in Table 7-4 and further explained in the following sections.

Operation Mask and Combination Rule

Two attributes — the *operation mask* and the *combination rule* — apply to all instructions that draw in the form. These attributes specify how pixels are to be combined for any instruction that writes to a form.

The *operation mask* specifies which bits in a pixel can be modified by drawing operations. A 0 means "do nothing with this bit"; a 1 means "operate on this bit using the combination rule." For example, if you set the operation mask to 1, only the low-order bit of any pixel will be modified.

The operation mask is functionally equivalent to the form mask. The difference is that the form mask, located in the form descriptor, will generally be used by the operating system to restrict a user's access to the bitmap. The effect of the operation mask can never be more than what the form mask allows; the operation mask and form mask are always logically ANDed together. The operation mask is part of the attribute block, and user programs may freely use it.

Table 7-4 *Form attributes*

Double-word # (octal)	Index # (decimal)	Mnemonic	Description
1	—	LENGTH	Length of the attribute block in 16-bit words.
2	0	OP_MASK	Operation mask — determines which bits within a pixel will be affected by a GIS operation.
3	1	COMBO_RULE	Combination rule — specifies one of 16 boolean functions to be applied during a GIS operation (see Table 7-5).
4	2	LINE_CTRL	Line control — set of flag bits that govern the drawing of lines. The individual bits and their interpretations are **Bit # Setting Description** 0 0 Draw the foreground pixels. 1 Suppress the foreground pixels. 1 0 Draw the background pixels. 1 Suppress the background pixels. 2 0 Draw the initial point(s). 1 Suppress the initial point(s). 3 0 Draw the final point(s). 1 Suppress the final point(s). 4 0 WGPLINE packet contains no attributes. 1 WGPLINE packet contains three new attributes for each line segment. 5 0 WGPLINE packet contains a contiguous line. 1 WGPLINE packet contains a noncontiguous line. 6-31 Reserved for future use; should be set to zeros.
5	3	LINE_F_COLOR	Line foreground color — Pixel value of the foreground color used when drawing lines.
6	4	LINE_B_COLOR	Line background color — Pixel value of the background color used when drawing lines.
7	5	LINE_STYLE	Line style — Set of bits that determine the texture of any lines that are drawn. Bits set to 1 indicate that a foreground color pixel should be planted. Bits cleared to 0 indicate that a background pixel should be planted.
10	6	CHAR_CTRL	Character control — Set of bits that govern the drawing of characters. The individual bits and their interpretations are **Bit # Setting Description** 0 0 Draw the foreground pixels. 1 Suppress the foreground pixels. 1 0 Draw the background pixels. 1 Suppress the background pixels. 2-31 Reserved for future use; should be set to zero.
11	7	CHAR_F_COLOR	Character foreground color — Pixel value of the foreground color used when drawing characters.
12	8	CHAR_B_COLOR	Character background color — Pixel value of the background color used when drawing characters.

You can use the operation mask to change the color of pixels being written to a form. This is useful in situations where you have a library of shapes or other small images that you will combine into a large picture. Each shape can be drawn in a library form, using a pixel value for which all bits are 1. When you transfer a shape to the main image area, the operation mask can selectively clear some bits so that the resulting pixels contain any desired color.

GIS instructions also apply a *combination rule* that controls how pixels are modified. Rather than simply overwriting the destination with the source data, GIS instructions can perform a number of different logical functions (on a bit-by-bit basis) to each bit in the source pixel and destination pixel.

Table 7-5 summarizes the sixteen different combination rules. Rules without a description in the table are not considered useful. Your program will probably do most drawing operations with rule 3, which performs a simple copy of source pixels to destination pixels. Note that any combination rule that does not involve *both* source and destination pixels will execute faster, such as combination rule numbers 0, 3, 5, 12, and 15.

Table 7-5 *Combination rules*

Rule	COMBO_RULE Bits 28-31	Logical Function	Description
0	0 0 0 0	dest := 0	Set destination bits to 0.
1	0 0 0 1	dest := src AND dest	—
2	0 0 1 0	dest := src AND (NOT dest)	—
3	0 0 1 1	dest := src	Move source bits to destination.
4	0 1 0 0	dest := (NOT src) AND dest	Mask out: set to 0 all destination bits for which the corresponding source bit is 1.
5	0 1 0 1	dest := dest	NO-OP: no change to destination.
6	0 1 1 0	dest := src XOR dest	Logical XOR.
7	0 1 1 1	dest := src OR dest	Merge: set to 1 all destination bits for which the corresponding source bit is 1.
8	1 0 0 0	dest := (NOT src) AND (NOT dest)	—
9	1 0 0 1	dest := src XNOR dest	Logical XNOR (equivalence).
10	1 0 1 0	dest := (NOT dest)	Complement destination bits.
11	1 0 1 1	dest := src OR (NOT dest)	—
12	1 1 0 0	dest := (NOT src)	Move complement of source bits to destination.
13	1 1 0 1	dest := (NOT src) OR dest	—
14	1 1 1 0	dest := NOT (src AND dest)	—
15	1 1 1 1	dest := 1	Set destination bits to 1.

Rules 6 and 9 implement the logical XOR and XNOR functions. With rule 6, the new value of the destination pixel is the exclusive OR of the source and the previous contents of the destination. For example, if you use this rule to write color 3 (0011_2) into a pixel containing color 5 (0101_2), the pixel will be set to (3 XOR 5), or 6 (0110_2).

This rule may produce some odd effects on the screen, but it has the unique advantage that you can erase or "undraw" any object by drawing it twice. Continuing the example above, if we use the same rule to write color 3 into the pixel that now contains 6, we find it will be restored to its original value of 5.

This effect is useful for programs that display temporary items, such as a menu or cursor, on top of an image. The temporary items can be quickly erased by re-executing the program statements that created them; there is no need to laboriously redraw the original image. Rule 9 has the same reversible property as rule 6, although it produces different colors.

Another useful property of rule 6 (or 9) is that it can be used to swap two pictures in memory without using any temporary storage.

When any pixel is operated on, the following calculation determines the resulting destination pixel (OP_AND_FORM_MASK refers to the logical AND of the operation mask and the form mask):

1. The combination rule is applied to the source pixel and the destination pixel.
2. This intermediate result is logically ANDed with the OP_AND_FORM_MASK.
3. The destination pixel is logically ANDed with the complement of the OP_AND_FORM_MASK.
4. Steps 2 and 3 are inclusively ORed together to produce the resulting destination pixel.

This procedure, expressed as an equation, is:

```
DEST_PIXEL := [OP_AND_FORM_MASK AND (SOURCE_PIXEL RULE DEST_PIXEL)]
              OR [~OP_AND_FORM_MASK AND DEST_PIXEL]
```

The visible effect of applying a combination rule during an instruction depends on the assignment of colors to the values of the source and destination pixels.

Line Drawing Attributes

There are four values in the attribute block that affect the operation of the **WGPLINE** instruction: the *line style* (LINE_STYLE), *line control word* (LINE_CTRL), *line foreground color* (LINE_F_COLOR), and *line background color* (LINE_B_COLOR). The colors planted are affected by the operation mask, form mask, and combination rule.

The values for line style, line foreground color, and line background color may also be specified by the **WGPLINE** packet. In this case, these values in the attribute block are undefined during and immediately after execution of the **WGPLINE** instruction.

LINE_STYLE together with LINE_CTRL are used to texture a line. LINE_STYLE is a string of 32 bits that define a pattern (solid, dotted, dashed, etc.). As **WGPLINE** draws each pixel, it looks at each bit in LINE_STYLE, going from the most significant bit (MSB) to the least significant bit (LSB). If the selected LINE_STYLE bit is 1 for a given pixel, the pixel is planted with the line foreground color. If the selected LINE_STYLE bit is 0, the pixel is planted with the line background color. When the LSB of LINE_STYLE is reached, the processor returns to the MSB of LINE_STYLE. This procedure starts at the first pixel of the first line segment, continuing to the last pixel of the last line segment.

LINE_CTRL is used to suppress the line foreground and/or line background colors when drawing a polyline. It is also used to suppress the initial and/or final endpoint of the polyline. Suppression of a pixel means that it will be left unaffected by the instruction.

If the line being drawn is a single point (all the endpoints of the polyline are the same coordinate), the point is drawn only if bits 2 and 3 of LINE_CTRL are set to 0. The attributes for the vertex of contiguous lines segments are derived from the attributes for the trailing line segment (see Figure 7-5).

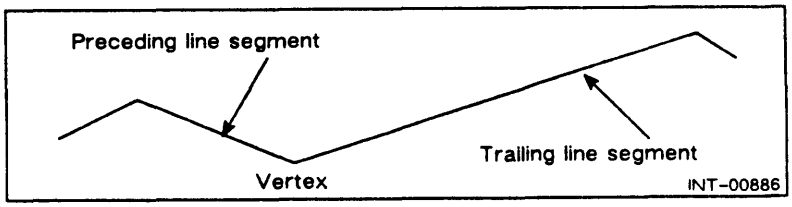


Figure 7-5 *Vertex of contiguous line segments*

For solid lines, use a line style of -1 (all bits equal 1). Some other values are given in Figure 7-6.

Binary value	Resulting line
11111111111111111111111111111111	Solid ————————
11111111111100001111111111110000	Dashed — — — — — — — —
10001000100010001000100010001000	Dotted . . . . .
11111111100010001111111110001000	Centerline — · — · — · — · — · —

INT-00010

Figure 7-6 *Effect of line style*

Character Drawing Attributes

Three attributes affect the action of the **WGCHRBLT** instruction: the *character control* word (CHAR_CTRL), *character foreground color* (CHAR_F_COLOR), and *character background color* (CHAR_B_COLOR). The colors planted are affected by the operation mask, form mask, and combination rule.

When **WGCHRBLT** draws a character, it normally writes the foreground color into all pixels corresponding to ones in the character cell, and it writes the background color into all pixels corresponding to zeroes. This action can be modified by CHAR_CTRL.

Bits 0 and 1 allow the foreground and/or background pixels to be suppressed. When they are suppressed, **WGCHRBLT** skips over them, instead of writing the specified color. When both bits are zero, each character drawn will be enclosed in a rectangle of the background color. Suppressing the background lets previously drawn material be seen around the characters. Suppressing the foreground causes each character to be drawn as a rectangle of background, with the character shape "cut out" so that previously drawn material can show through.

Character Fonts

A *font* is a set of shapes for letters, numbers, and punctuation marks, also known as a *character set*. The GIS Character Block Transfer (**WGCHRBLT**) instruction writes characters onto a bitmap. Under GIS, a character is defined by a rectangular set of pixels on a form. To use **WGCHRBLT**, the character source form must have only one bit per pixel (X pitch = 1) and be one virtual rectangle; otherwise, there are no limits on the size or shape of the character. **WGCHRBLT** can be used to draw graphic icons, logic circuits, and other special-purpose objects. A complete font can consist of many small forms or a single large form that contains the shapes for all the characters.

Cursor Descriptor

A cursor is a pattern drawn on the bitmap screen to represent the position of a pointing device.

Every form descriptor includes the address of a *cursor descriptor* (CURSOR_DESC) that can define the location of a graphic cursor for use with an input device such as a mouse

or light pen. This permits the cursor to be managed by the operating system, even though it is drawn over the user's picture in the form. GIS instructions use the descriptor to determine if a drawing operation may overwrite the cursor. If so, the instruction is interrupted. Then the operating system can remove the cursor, complete the user's operation, and later restore the cursor.

There are two different types of cursors: *image* and *cross hair*. The image cursor may take any shape — for instance, an arrow. This cursor is defined by the rectangle that contains the visible portion of the image.

The cross-hair cursor consists of one horzontal line and one vertical line. This cursor is defined by the endpoints of the horizontal and vertical lines that form the cross hair. In the case of a full-screen cross hair, the horizontal and vertical lines always span the entire width and height, respectively, of the bitmap screen. The endpoints of a full-screen cross hair are always on the edge of the bitmap screen. In the case of a cross hair that is very large, the cross hair may be clipped to the edge of the bitmap screen. The endpoints of a large cross hair define the visible portion of the cross hair.

For each of the two types of cursors, there is a different cursor descriptor. The cursor descriptor consists of signed and unsigned 32-bit integers. The first doubleword of the descriptor remains the same in both cases. This is the FLAGS doubleword, which contains bits that determine the format of the descriptor that follows. The format of the cursor descriptors is shown in Figure 7-7, and in Tables 7-6 (cross-hair descriptor) and 7-7 (image descriptor). The Load Forms instruction (**WGLFORM**) loads the initial cursor descriptor block into the forms cache; the Load Cursor Descriptor instruction (**WGLDCURS**) may then be used by the operating system to change the cursor by modifying the cursor descriptor block and updating the internal cache entries.

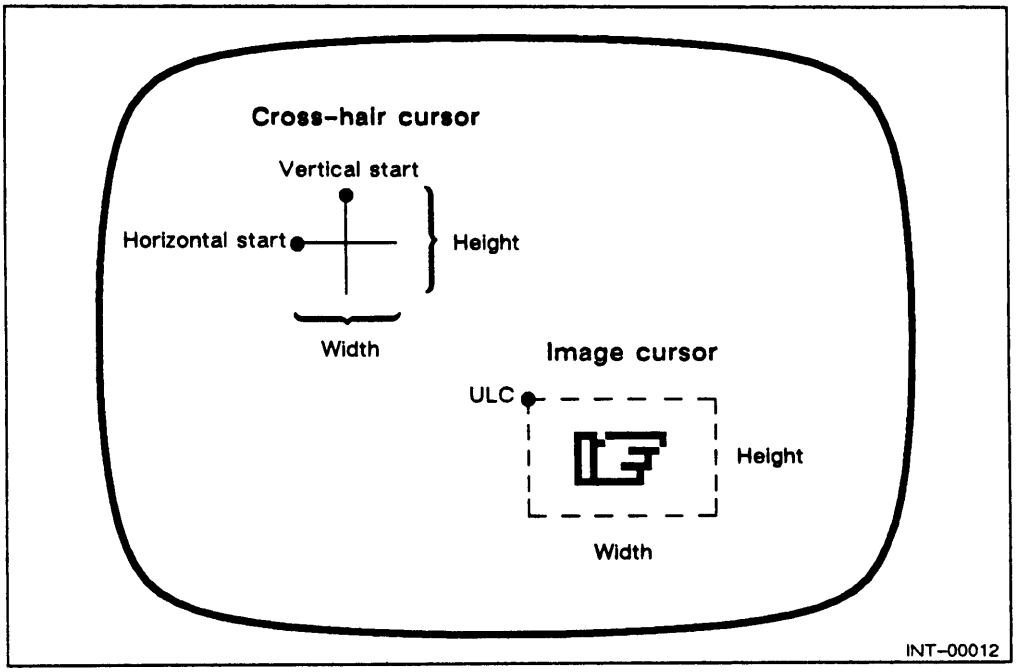


Figure 7-7 *Types of cursors*

Table 7-6 *Cross-hair cursor descriptor*

Double Word #	Mnemonic	Contents
1	FLAGS	Flag bits. The individual bits and their interpretations are Bit # — Meaning 0 — If 1, cursor is visible. If 0, cursor invisible. 1, 2 — Must be set to 01_2 for cross-hair cursor. (Only one of bits 1 and 2 may be set at any one time. Setting both of these bits will produce undefined results.) 3-31 — Reserved for future use; should be set to 0.
2	H_START_X	X coordinate (signed) of left endpoint of horizontal line with respect to physical bitmap origin.
3	H_START_Y	Y coordinate (signed) of left endpoint of horizontal line with respect to physical bitmap origin.
4	H_END_X	X coordinate (signed) of right endpoint of horizontal line with respect to physical bitmap origin.
5	H_END_Y	Y coordinate (signed) of right endpoint of horizontal line with respect to physical bitmap origin.
6	V_START_X	X coordinate (signed) of top endpoint of vertical line with respect to physical bitmap origin.
7	V_START_Y	Y coordinate (signed) of top endpoint of vertical line with respect to physical bitmap origin.
8	V_END_X	X coordinate (signed) of bottom endpoint of vertical line with respect to physical bitmap origin.
9	V_END_Y	Y coordinate (signed) of bottom endpoint of vertical line with respect to physical bitmap origin.

Table 7-7 *Image cursor descriptor*

Double Word #	Mnemonic	Contents
1	FLAGS	Flag bits. The individual bits and their interpretations are Bit # — Meaning 0 — If 1, cursor is visible. If 0, cursor invisible. 1, 2 — Must be set to 10_2 for image cursor. (Only one of bits 1 and 2 may be set at any one time. Setting both of these bits will produce undefined results.) 3-31 — Reserved for future use; should be set to 0.
2	ULC_X	X coordinate (signed) of ULC of cursor rectangle with respect to physical bitmap origin.
3	ULC_Y	Y coordinate (signed) of ULC of cursor rectangle with respect to physical bitmap origin.
4	EXTENT_X	Width of cursor rectangle in pixels (unsigned).
5	EXTENT_Y	Height of cursor rectangle in pixels (unsigned).

Color Descriptors

Different display devices in the Data General product line have different palette organizations. The Write Palette (**WGWRPAL**) and Read Palette (**WGRDPAL**) instructions, however, use a generalized data format that is compatible with all displays. Furthermore, the data format supports future hardware designs that may use palette registers up to 96 bits long.

In this data structure, each palette register's contents is translated to or from a *color descriptor* consisting of four 32-bit numbers. The first three numbers represent red, green, and blue components, respectively, and are used only by color displays. The fourth number represents the gray-scale intensity, and is used only by monochromatic displays. If you are writing a program to run on both color and monochromatic displays, you should specify values for all four numbers.

Unlike most data used by GIS, palette values are left justified within the descriptor words. That is, when the hardware reads or writes a value that is less than 32 bits long, it reads or writes the leftmost bits of the word.

Left justification increases the compatibility of systems with different display hardware. When you write a palette register with **WGWRPAL**, the hardware takes as many bits as it can use, starting from the leftmost bit. You can specify the color values to any desired precision, and the hardware will match your intentions as closely as possible, given the number of bits in the actual palette register. When you read a palette register with **WGRDPAL**, the hardware places the bits from the palette in the leftmost bits of the descriptor word(s), and sets all unused bits to 0.

Form Cache

In order for a GIS instruction to operate on a form, the target form must be loaded into the form cache (using the Load Form instruction, **WGLFORM**). If the target form is not in the form cache, a *form cache miss* fault occurs (refer to the section, "Fault Handling"). The operating system controls access to forms by loading or not loading a particular form into the form cache in response to this fault.

A cache tag uniquely identifies a form in the form cache. The cache tag is three doublewords consisting of the address of the form descriptor and two keys. The first key is the user's form ID, a number that is supplied on a GIS instruction in AC1. The form ID must be a value other than 0. The second key is some process-specific number chosen by the operating system. For ECLIPSE MV/Family machines, this number is the contents of the segment base register for the ring on whose behalf the form was loaded into the form cache. This key is then specific to each ring and to each process.

To determine if the form required by a particular GIS instruction is in the form cache, the processor obtains the user's form ID and the operating system's key and searches the form cache for a cache tag containing these values.

- If a cache tag is found whose contents match the user's form ID, the remaining actions are dependent upon the segment number:

  If the GIS instruction was issued from segments 1 through 7, the processor then checks the operating system's key. If the key matches, the GIS instruction continues, using the form pointed to by the form descriptor address. If the key does not match, the processor returns a form cache miss fault to the operating system.

  If the nonprivileged GIS instruction was issued from segment 0, the processor ignores the operating system's key and continues execution of the GIS instruction.

- If no match of the user's form ID is found, the processor returns a form cache miss fault to the operating system.

While processing a GIS instruction, any forms in the form cache that are unused by this instruction may be purged.

Interrupts

All GIS instructions are interruptible and either resumable or restartable. When an interrupt occurs, a GIS instruction saves the current state on the wide stack and sets bit 2 (IRES) of the Processor Status Register to 1. When the interrupt service is complete, control passes back to the GIS instruction, which pops the saved state off the wide stack, sets bit 2 of the Processor Status Register to 0, and continues with its execution. For further information, refer to the section, "Interrupt Servicing," in the chapter, "Device Management."

Fault Handling

When a GIS instruction executes, the processor checks various parameters before continuing execution of the instruction. Conditions that could cause a GIS fault are:

- Form ID not found in form cache,
- Instruction could corrupt the cursor,
- Unknown attribute index, or
- Invalid **WGCHRBLT** source form.

If one of these conditions occurs, the processor generates a GIS fault (using the same mechanism for page faults, as described in the chapter, "Memory and System Management"). When a GIS fault occurs, the following results (refer also to Figure 7-8):

1. If the current segment is not 0, the processor stores the wide frame pointer and wide stack pointer in their respective locations in page zero of the current segment and performs a segment crossing to segment 0.
2. The processor uses the contents of locations 32_8 and 33_8 of page zero in segment 0 as a base address to store a context block (the internal state of the machine) in memory. The structure of the 10-doubleword GIS fault context block is:

Double Word #	Contents
1	Program status register (bits 0-15 contain the PSR; bits 16-31 contain zeros)
2	User's AC0
3	User's AC1
4	User's AC2
5	User's AC3
6	Carry (bit 0) and PC (bits 1-31) on fault
7	Next segment of execution (bits 0-2)
8	Microcode block state
9	Value causing fault (see step 4).
10	Restart value

   When a GIS fault occurs, the restart value field (doubleword 10) of the context block contains a zero. While handling the fault, a new value may be placed in this location. Upon return from the fault handler, if this value is 1, the CPU will execute the instruction following the faulting GIS instruction.
3. The processor initializes the segment 0 stack from page zero of segment 0.

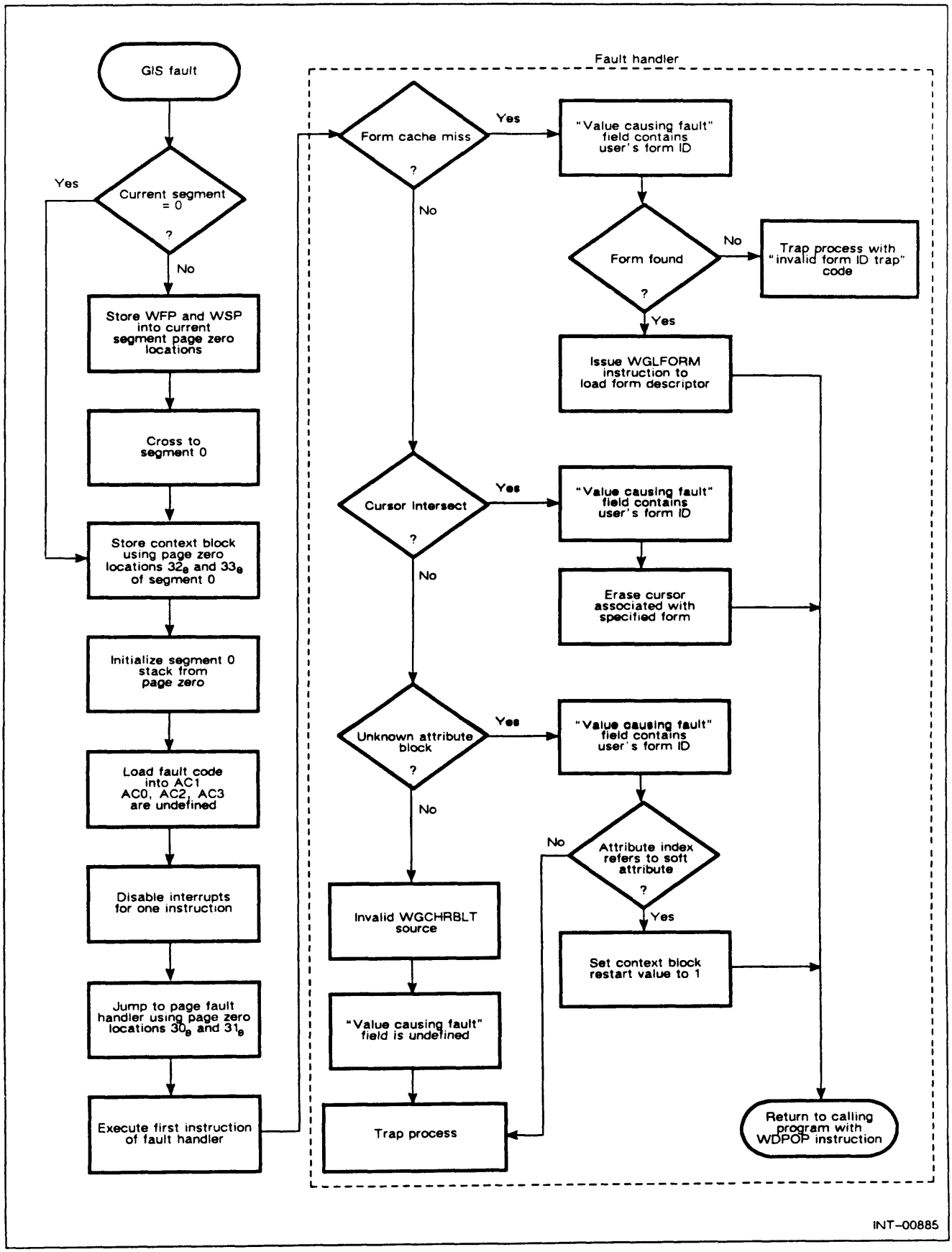


Figure 7-8 *GIS fault sequence*

4. The processor loads a fault code into AC1 as follows:

Fault Code	Fault Type	Value in Context Block Doubleword 9 (Value Causing Fault)
0-4	Reserved for page faults	Refer to the chapter, "Memory and System Management."
5	Form cache miss	User form ID of form not found in form cache.
6	Cursor intersect	User form ID supplied on GIS instruction.
7	Unknown attribute index	User form ID supplied on WGRDATTR or WGWRATTR instruction.
8	Invalid WGCHRBLT source	Undefined.
9-12	Reserved	Reserved for future use.

The unknown attribute index and invalid **WGCHRBLT** source faults can only be taken at the start of a GIS instruction. Once a GIS instruction starts to execute, these faults are invalid. The form cache miss and cursor intersect faults may be taken at any time.

The contents of AC0, AC2, and AC3 are undefined.

5. The processor disables interrupts for one instruction and jumps to the page fault handler using the address in locations 30_8 and 31_8 of segment 0.

6. The processor executes the first instruction of the fault handler. The fault handler may then take action dependent upon the type of fault:

   - Form cache miss (form cache does not contain specified form ID).

     a. The value causing fault field in the context block contains the user's form ID.

     b. The operating system searches for specified form in its internal databases.

     c. If no form is found, the operating system can trap the process using the same mechanism as an inward address violation. The trap code will be "invalid form ID trap." (Refer to the chapter, "Memory and System Management.")

     d. If a form is found, the operating system issues a **WGLFORM** instruction to load the form descriptor.

     e. The operating system returns control with a **WDPOP** instruction.

   - Cursor intersect (a particular GIS instruction could corrupt the cursor).

     a. The value causing fault field in the context block contains the user's form ID.

     b. The operating system erases the cursor associated with the specified form.

     c. The operating system returns control with a **WDPOP** instruction.

- Unknown attribute block (the attribute index provided on a **WGRDATTR** **WGWRATTR** instruction falls outside of the form descriptor's attribute block).

  a. The value causing fault field in the context block contains the user's form ID.

  b. If the attribute index, specified by the user, does not refer to a valid "soft" attribute, the operating system traps the process.

  c. If the attribute index, specified by the user, refers to a valid "soft" attribute, the operating system sets the restart value field of the context block to 1.

  d. The operating system returns control with a **WDPOP** instruction.

- Invalid **WGCHRBLT** source (the source form given on this instruction does not meet the following restrictions: the rectangle list consists of a single rectangle on the virtual bitmap; the virtual bitmap is one bit per pixel deep.)

  a. The value causing fault field in the context block is undefined.

  b. The operating system traps the process.

Fixed-Point Overflow

Certain GIS instructions perform arithmetic operations during execution. If these operations produce a fixed-point overflow, the following may occur:

- The instruction continues, but the results are undefined.
- The processor status register overflow bit (OVR) is set to one.

GIS instructions, such as **WGPLINE** or **WGRFLOOD**, may produce a fixed-point overflow during an *overdraw* condition.

An overdraw condition occurs when a GIS instruction attempts to either write a location that is beyond the *clippable area* or draw a line with endpoints that are further apart than the allowable maximum.

Each bounding rectangle has a corresponding clippable area associated with it. The clippable area is defined as a rectangle whose sides are parallel to the bounding rectangle with each side no farther than $2^{31} - 1$ points from the opposite side of the bounding rectangle. Any instruction's packets may contain coordinates that encompass points within the clippable area (valid) and beyond the clippable area (invalid). GIS will apply correct clipping to valid coordinates, but will not draw any points outside the bounding rectangle. Coordinates outside the clippable area produce a fixed-point overflow with undefined results. Figure 7-9 is a representation of the clippable area.

In the figure, the parameters of the bounding rectangle are defined by its ULC and points X + X_EXT, Y + Y_EXT. A line drawn through point A (with the **WGPLINE** instruction) is valid if any point on that line does not extend beyond the clippable area and produce an overdraw condition. Therefore, line BC (which extends beyond the bounding rectangle) is valid and the portion within the bounding rectangle will be drawn. However, line DE, which contains points beyond the acceptable clipping area, is invalid and causes a fixed-point overflow with undefined results.

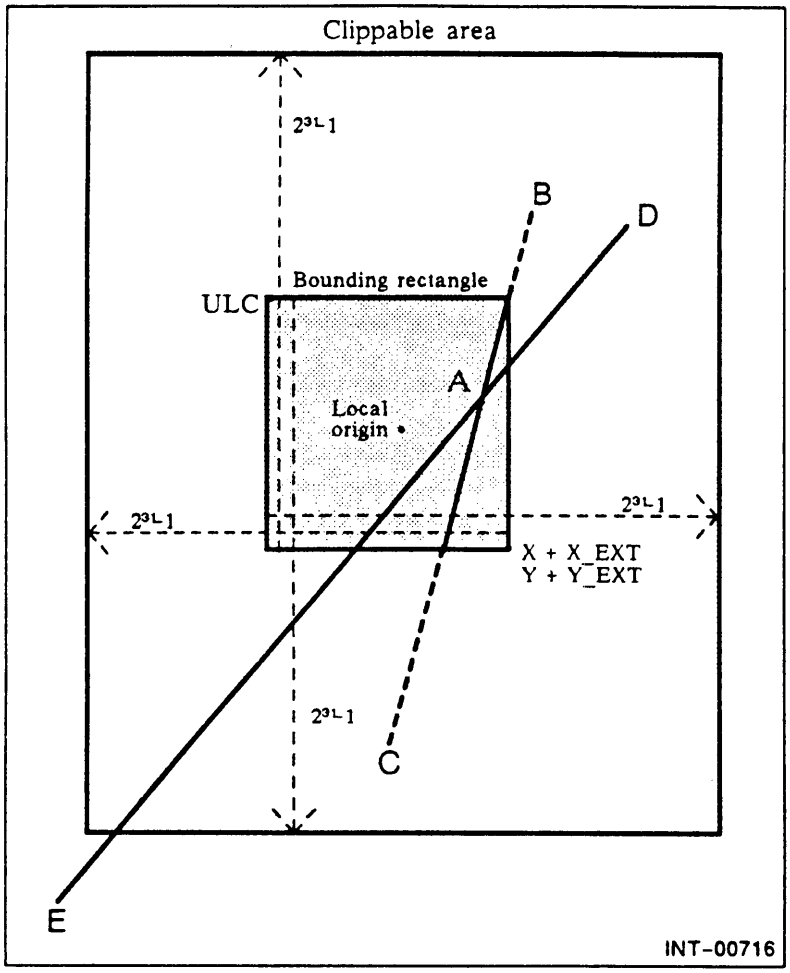


Figure 7-9 *Overdraw condition parameters*

An overdraw condition also occurs if a GIS instruction, such as **WGPLINE**, attempts to draw a line with endpoints that are further apart than the allowable maximum. This limit is defined as 2^{29} points from one endpoint to the perpendicular of the second point. Both endpoints must also be within the clippable area. If the maximum allowable distance is exceeded, a fixed-point overflow may occur even though the endpoints are within the clippable area. As shown in Figure 7-10, line FG is valid if each endpoint is less than or equal to 2^{29} points to the perpendicular of the other endpoint (FZ and GZ). If the endpoints are valid, the portion of the line within the bounding rectangle will be drawn.

For further information on fixed-point overflow, refer to the section, "Fixed-Point Overflow Fault," in the chapter, "Program Flow Management."

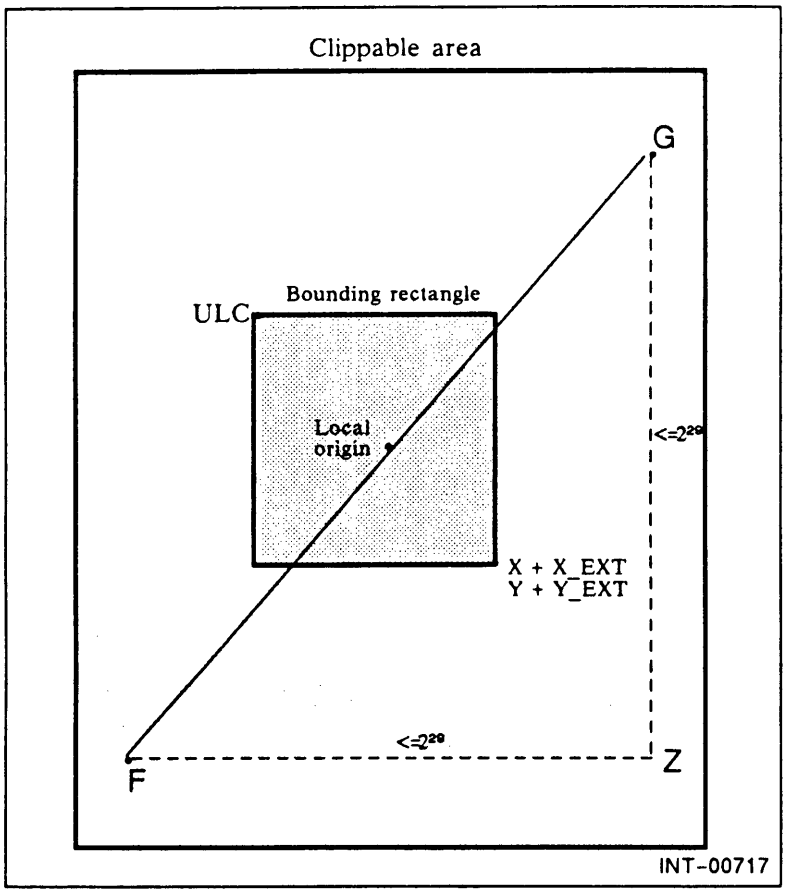
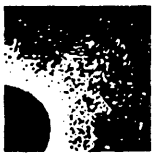


Figure 7-10 *Overdraw condition parameters for endpoints*

End of Chapter



8

Device Management

The processor supports devices that transfer data using a slow-, medium-, or high-speed transfer rate. With a programmed I/O facility, the processor transfers 1 or 2 bytes of data between a device and an accumulator. With a data channel I/O facility, the processor transfers words or blocks of words between a medium-speed device and memory. With a burst multiplexor channel I/O facility, the processor transfers bursts of data between a high-speed device and memory.

For instance, a slow-speed asynchronous line controller transfers data with the programmed I/O facility. Medium-speed devices, such as line printers and magnetic tapes, transfer data with the data channel (DCH) I/O facility. The high-speed disk drives and high-speed magnetic tape drives transfer data with the burst multiplexor channel (BMC) I/O facility.

Devices are either external or internal to the computer.

- External devices are those peripherals residing on either the I/O bus or BMC with communications generally handled through a device controller (such as disk drive and magnetic tape units, printers, and terminals).
- Internal devices are those which are integral to the computer and accessible directly by the processor without the necessity of communicating through a device controller (such as the CPU, real-time clock, programmable interval timer, system control processor).

Depending upon the operating system, a device is usually accessed through a system call to an operating system. This chapter presents basic information to assist in reading and writing an interrupt or device handler routine. The chapter first provides general information pertinent to all devices: device access, I/O instructions, and interrupts. We then discuss the data channel and burst multiplexor channel, device controllers in general, and writing device handler routines to support external peripherals. The chapter concludes with information on integral (internal) devices. For descriptions of the structure, functions, signals, and timing characteristics of the I/O buses, refer to the machine-specific interface designer's guide.

I/O Communication

A peripheral generally consists of one or more devices and a device controller. Communication between the processor and a device is through the device controller using one or more of the I/O facilities (programmed, data channel, burst multiplexor channel). An intermediary for some ECLIPSE MV/Family systems is an I/O channel controller (IOC). An IOC manages access to an I/O channel with each IOC maintaining its own set of buses (generally both a data channel and a burst multiplexor channel). Figure 8-1 shows an ECLIPSE MV/Family system with dual I/O channel controllers.

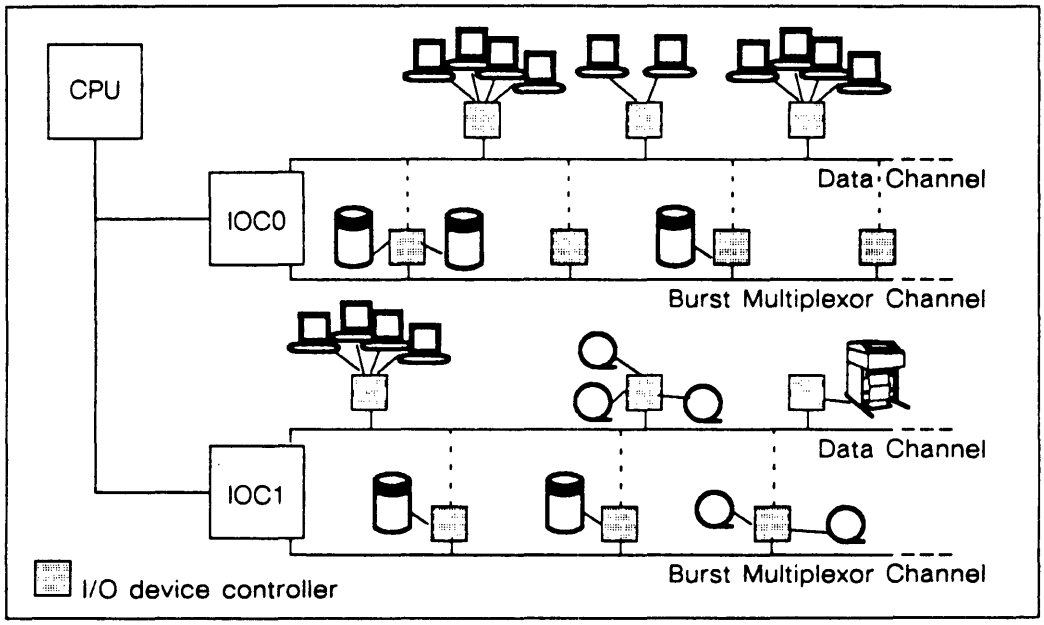


Figure 8-1 *An ECLIPSE MV/Family system with dual IOCs*

The I/O bus is shared by all the device controllers as well as by the computer, while the BMC bus is shared only by the computer and device controllers which incorporate the BMC facility. Since these buses are shared they are, by necessity, half-duplex buses with only one operation occurring at any one time. However, an operation can be occurring simultaneously on each bus.

The I/O bus connects in parallel to each DCH and BMC device controller in the system; the BMC connects in parallel to each device controller that uses the BMC facility.

The direction of all I/O transfers is relative to the computer. *Output* refers to information moving from the computer to a device controller; *input* refers to information moving from a device controller to the computer.

I/O Access

The processor accesses a device with address translation disabled or enabled. When address translation is

- disabled — the processor is running in physical mode (it ignores the contents of the segment base register) and executes the I/O instruction.
- enabled — the processor checks the segment base register for the current segment before it executes an I/O instruction.

Bits 2 and 3 of the segment base registers affect I/O access for each segment as follows:

- Bit 2 is the LEF or I/O mode bit (specifying how the processor interprets the LEF and I/O instruction opcodes). If bit 2 is set to

  1 It indicates *LEF mode*. The processor interprets and executes both I/O and LEF instructions as LEF (load effective address) instructions.

  0 It indicates *I/O mode*. The processor interprets I/O instructions and LEF instructions as I/O instructions. (Executing an I/O instruction requires an additional interpretation of bit 3.)

  NOTE: *Bit 2 affects the* **LEF** *instruction but not the* **ELEF**, **XLEF**, *and* **LLEF** *instructions.*

- Bit 3 is the I/O validity flag (enabling or disabling the execution of an I/O instruction). If bit 3 is set to

  1 The processor allows execution of I/O instructions.

  0 The processor detects a protection violation when attempting to execute an I/O instruction.

Refer to the chapter, "Memory and System Management," for information on the segment base registers and servicing a protection fault.

I/O Registers

Several registers provide status and control information for the I/O channel controllers, the DCH controller, and the BMC controller:

- BMC and DCH map slot registers — contain logical-to-physical address translation information for BMC and DCH transfers to memory.
- I/O channel definition register — contains error and status information for the BMC and DCH.
- I/O channel status register — contains status information for the I/O channel.
- I/O channel mask register — contains mask bits for each I/O channel.
- CPU dedication control register (multiple-processor systems only) — specifies which processor in a multiple-processor system will receive I/O interrupts.

The section, "Data Channel/Burst Multiplexor Channel," discusses the registers, their effects on I/O transfers, and the I/O instructions for manipulating these registers.

Types of Information Transfers

The computer and a device controller transfer information in one of three ways:

- Programmed I/O control — Moves a word, or part of a word, between an accumulator and a register in the device controller. This type of transfer occurs over the I/O bus when a program executes the appropriate I/O instruction.

  This method provides communication with all internal devices and sets up all DCH and BMC operations (with the actual data transfers performed over the respective channels). Since at least one instruction — and most likely several — must execute for each character or word transferred, programmed I/O is slower and generally used only for devices that do not have to transfer large quantities of information quickly.

- Data channel control — Generally moves a block of data, one word at a time, between physical memory and the device through a register in the device controller. The block of data is transferred automatically over the data channel once the program sets up the transfer for a particular device.

  Transferring large blocks of data under DCH control reduces the amount of program overhead required. The information to set up the DCH transfer is assembled in the accumulators and then transferred to the device controller with programmed I/O instructions. The block of data is then automatically transferred between memory and the device controller over the data channel.

  Each time the device controller is ready to transfer a word from the block, it requests access to the DCH bus controller. When access is granted, the device controller transfers a memory address to the bus controller and then the word is either transferred and written to memory or read from memory and transferred to the device controller. Because multiple I/O instructions do not execute for each word transferred, block transfers can occur at high rates.

- Burst multiplexor channel control — Moves a block of data. The block of data is transferred automatically over the BMC once the program sets up the transfer for a particular peripheral. The block is transferred in a burst of several words between physical memory and the device through registers in the device controller. The words of the burst are transferred in serial succession (one word at a time).

  The information to set up the BMC transfer can be assembled in the accumulators and then transferred to the device controller with programmed I/O instructions over the I/O bus (identical to data channel control). Once the device controller is set up, the block of data is then transferred directly between memory and the device controller over the BMC.

  After the program sets up and initiates the BMC transfer for a block of data, it does not have to take further action. Each time the device controller is ready to transfer a specified number of words from the block, it requests access to the BMC. When access is granted, the device controller provides a starting memory address and the number of words to be transferred during the data transfer. Then the number of words specified for the data burst are transferred.

  Since the channel only has to be activated once and a memory address only has to be supplied once for each burst of words transferred, block transfers over the BMC can occur at much higher rates than those under data channel control.

With the majority of ECLIPSE MV/Family systems, the actual transfer of data using the DCH or BMC facilities does not disturb the state of the processor. The data is transferred directly between the registers in the device controller and physical memory. This greatly reduces the amount of program overhead in the form of executing I/O instructions and loading or storing data. A program sets up the device for the transfer; the controller for the respective bus then performs the transfer.

All DCH and BMC device controllers receive their commands using the standard I/O bus. Data transfers for DCH devices are performed on the I/O bus; data transfers for BMC devices are performed on the BMC bus.

In addition, where intelligent device controllers are used, the commands to set up the transfer can be assembled in a control block in host memory. After the program supplies a starting memory address for the control block and initiates the device controller with programmed I/O instructions using the I/O bus, the control block is transferred to the device controller over the respective bus (DCH or BMC). Multiple device controller operations can be performed without additional program intervention when a group of control blocks are linked together.

General I/O Instructions

The I/O instructions provide communications between the processor and a device controller. A general set of I/O instructions provides device-independent operations. A special set of I/O instructions communicates with the device controller, loads a device map, or services a vector interrupt.

The general I/O instructions receive or send data, and initialize or test a device flag. The Programmed I/O (**PIO**) instruction issues a general I/O instruction (contained in an accumulator) to an I/O device on a specified I/O channel. Table 8-1 lists the general I/O instructions.

Table 8-1 *General I/O instructions*

Instruction	Opcode	Operation
DIA[*f*] *	001	Data in A (from A buffer of device to an accumulator)
DIB[*f*] *	011	Data in B (from B buffer of device to an accumulator)
DIC[*f*] *	101	Data in C (from C buffer of device to an accumulator)
DOA[*f*] *	010	Data out A (from an accumulator to A buffer of device)
DOB[*f*] *	100	Data out B (from an accumulator to B buffer of device)
DOC[*f*] *	110	Data out C (from an accumulator to C buffer of device)
IORST *	101	I/O reset
NIO[*f*] *	000	No I/O transfer (initialize a Busy/Done flag)
SKP*t* *	111	I/O skip (test a Busy/Done flag and skip on condition)

* *ECLIPSE compatible instruction*

Figure 8-2 illustrates the format for a general I/O instruction; Table 8-2 describes the format.

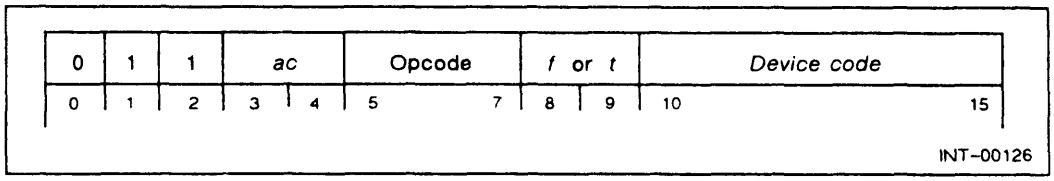


Figure 8-2 *General I/O instruction format*

Table 8-2 *General I/O instruction format description*

Mnemonic	Bits	Meaning
011	0-2	This binary code indicates an I/O instruction.
ac	3, 4	The *ac* field specifies a fixed-point accumulator in the range 0-3. Bits 16-31 of the accumulator contain the data to send to or receive from a device.
Opcode	5-7	The opcode field identifies the I/O instruction operation. Table 8-1 lists the I/O instructions and their opcodes.
f or *t*	8, 9	This field indicates a device flag and a function to perform on that flag. The *f* bit identifies a device flag to change; the *t* bit identifies a device flag to test. Depending on the I/O instruction and the device, the instruction initializes or tests the device flag of the device. (See Tables 8-3 and 8-4.)
Device code	10-15	The device code field identifies a unique device controller to send data to or to receive data from. With a 6-bit device code, the processor can communicate with up to 64_{10} device controllers per I/O channel. The assembler translates a standard three, four, or five letter device mnemonic to a device code. Refer to the machine-specific "Standard I/O Device Codes" appendix for a list of standard device mnemonics and their corresponding device codes.

Device Flags

The Busy and Done flags indicate the device state to a device handler routine. For all external and internal devices (except the CPU), the flags are Busy and Done. For the CPU, the flags are interrupt on and powerfail. Note that the CPU is considered a device for programming purposes.

When both the Busy flag and Done flag equal zero, the device is idle. To start a device, issue an I/O instruction with the proper device flag control that sets the Busy flag to one and the Done flag to zero. When the device finishes the operation and becomes ready to start another operation, the device controller sets its Busy flag to zero and its Done flag to one.

The interrupt on flag (ION) controls the device interrupt system. When ION equals

- 0, the processor ignores interrupt requests.
- 1, the processor services interrupt requests. (Refer to the note in the section, "Instruction Interruption.")

The read-only powerfail flag indicates the power state to the CPU device driver. When the powerfail flag equals

- 0, the processor detects the proper power voltage ranges.
- 1, the processor detects a powerfail condition.

Table 8-3 *Device flag controls for general devices*

Assembler Code for *f*	Bits 8 9	I/O Busy	I/O Done	CPU ION
(option omitted)	0 0	No effect	No effect	No effect
S	0 1	Set to a 1	Set to a 0	Set to a 1
C	1 0	Set to a 0	Set to a 0	Set to a 0
P	1 1	Pulses a special I/O bus control line		No effect

Table 8-4 *Device flag tests for skip instruction*

Assembler Code for *t*	Bits 8 9	I/O	CPU
BN	0 0	Test for Busy = 1	Test for ION = 1
BZ	0 1	Test for Busy = 0	Test for ION = 0
DN	1 0	Test for Done = 1	Test for powerfail = 1
DZ	1 1	Test for Done = 0	Test for powerfail = 0

Interrupts

The processor and an operating system maintain the I/O facilities through a hierarchical interrupt system. Any program can initiate an I/O operation by requesting a data transfer to or from a device. With most operating systems, the program transmits the request through I/O system calls, which initialize the device and transfer data by invoking the interrupt system.

When a device completes an operation, its controller sets its Done flag to 1 and the Busy flag to 0 to indicate that it requires service. The program can then test the state of the Done flag with the I/O Skip (**SKP**) instruction to determine when this occurs. Checking a device's Done flag needs to be done frequently to ensure that service is not delayed. These status checks are time consuming, and, to avoid the necessity of repeating them, all ECLIPSE MV/Family computers have a program interrupt facility.

All device controllers that use the program interrupt facility have access to an interrupt request line — a single, direct signal to the processor along which requests for service are communicated. An interrupt request can be generated by a device controller when the controller's Done flag is set to 1. The processor can respond to an interrupt request by halting the normal flow of program execution and transferring control to an interrupt-handling routine. A program can control which device controllers may request interrupts and when the processor may start an interrupt by manipulating a number of interrupt flags.

Interrupt Flags

The operating system maintains control of the interrupt system by manipulating the interrupt on flag, an interrupt mask, and device flags. The interrupt on flag and interrupt mask reside in the processor. The interrupt on flag enables or disables all interrupt recognition, while the interrupt mask enables or disables selective device interrupt recognition.

The Busy and Done device flags reside in the device controller and provide the interrupt communication link between the processor and the device. By manipulating the flags and the interrupt mask, the interrupt system can ignore all interrupt requests or selectively service certain interrupt requests.

If the interrupt on flag and interrupt mask enable processor recognition of the interrupt request, the processor services the interrupt. When the interrupt on flag (ION) equals

- 1, the processor responds to an interrupt request.
- 0, the processor cannot respond to an interrupt request.

The CPU instructions, Interrupt Disable (**INTDS**) and Interrupt Enable (**INTEN**), control the state of the interrupt on flag. Information on CPU instructions is presented later in this chapter in the section, "Integral Devices."

Each of the 16 bits in the interrupt mask may be associated with one or more devices. When a bit in the interrupt mask equals

- 1, an interrupt request from that device(s) to the processor is blocked.
- 0, the processor services an interrupt request from the device.

To change the state of a bit in the interrupt mask, use the Mask Out instruction (**MSKO**), a CPU instruction. Each device controller using the interrupt facility contains an interrupt disable flag. All device controller interrupt disable flags are manipulated at

once with a MSKO instruction using a mask contained in an accumulator. (Each device controller is assigned by its hardware to a bit position in the mask. Mask bit assignments for standard device controllers are given in the machine-specific supplement.) When a MSKO instruction executes, each device controllers' interrupt disable flag is set to the value of the assigned bit of the mask (a value of 1 disables the device from posting an interrupt to the processor; a value of 0 enables the posting of an interrupt). Following powerup, or when a reset occurs, all interrupt disable flags are set to 0.

To service an interrupt, the processor first determines the action to take on the currently executing instruction, next redefines the interrupt mask, and finally services the interrupt request. The section "Processor Interrupt Servicing" explains the processor's actions to transfer program control to the interrupt handler, and then to the interrupt service routine.

Instruction Interruption

Most instructions that require only a minimum of processor execution time are noninterruptible. For instructions that require more execution time, the processor (if required) interrupts the executing instruction, updates the accumulators, and services the interrupt. If the instruction must continue where it left off (resumable instruction), the processor also sets the processor status register interrupt resume flag (IRES) to 1. After servicing the interrupt, the processor either restarts or resumes the interrupted instruction.

NOTE: *Some processors use nonmaskable interrupts (NMIs) to control internal processor events. NMIs are not masked by the state of ION. Thus, any instruction that is interruptable (such as* **WBLM***) produces undefined results if the instruction overwrites the executing opcode, regardless of the state of ION. An NMI causes the execution of the instruction to be stopped and then resumed with updated accumulator values. If the instruction (***WBLM***) overwrites itself, then the opcode may no longer be available when the instruction is resumed.*

ECLIPSE MV/Family processors set the processor status register bit 2 (IRES) to 1 when an interrupt occurs during execution of a resumable instruction. The processors also set PSR bit 3 (IXCT) to 1 if the interrupted instruction were executed by a Pop Block and Execute (**PBX**) instruction.

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

Processor Interrupt Servicing

To service an interrupt request (Figure 8-3), the processor

1. Sets ION to 0.
2. Determines if the address translation facilities are enabled.

   If address translation is enabled, the processor continues with step 3.

   If address translation is disabled, the processor gets the interrupt handler pointer from physical location 1_8 and continues with step 7.

   Refer to the chapter "Memory and System Management" for information on enabling and disabling the address translation facilities.
3. Stores the current stack register values in the respective page zero locations of the current segment.
4. Changes the current segment of execution to segment 0.
5. Initializes the wide stack using the values from page zero locations of segment 0.
6. Fetches the interrupt handler pointer from word 1_8 of reserved page zero memory for segment 0.
7. Resolves the effective address of the interrupt handler.
8. Examines the first word of the interrupt handler, which may be one of the following types:

   Type 1 — An ECLIPSE MV/Family 32-bit processor instruction.

   A 32-bit processor instruction contains bit 0 equal to 1 and bits 12-15 equal to 1001_2.

   Type 2 — An ECLIPSE 16-bit instruction.

   Instructions other than **XVCT** or type 1 are identified as ECLIPSE 16-bit instructions.

   Type 3 — A vector interrupt (**XVCT**) instruction.
9. Stores the return address in the following locations (according to instruction type):

   Type 1 — Logical locations 2_8 and 3_8 of segment 0.

   Type 2 — Location 0 of segment 0.

   Type 3 — The vector stack (as part of the return block) for the **XVCT** instruction, during the vector interrupt processing.
10. Jumps indirectly (according to instruction type) as follows:

    Type 1 — To the immediate interrupt handler and executes the type 1 instruction as the first instruction of the handler.

    A jump instruction (**LJMP** or **XJMP**) can be used to jump indirectly through the return address in order to return from the interrupt handler.

    Type 2 — To the ECLIPSE interrupt handler and executes the type 2 instruction as the first instruction of the ECLIPSE interrupt handler.

    Type 3 — To the vectored interrupt handler (through word 7_8 of page zero for segment 0) and executes the **XVCT** instruction. The next section describes vectored interrupt processing.

    The last instruction of the vectored interrupt handler should be a wide restore from vector interrupt instruction (**WRSTR**), which pops the wide return block from the vector stack.

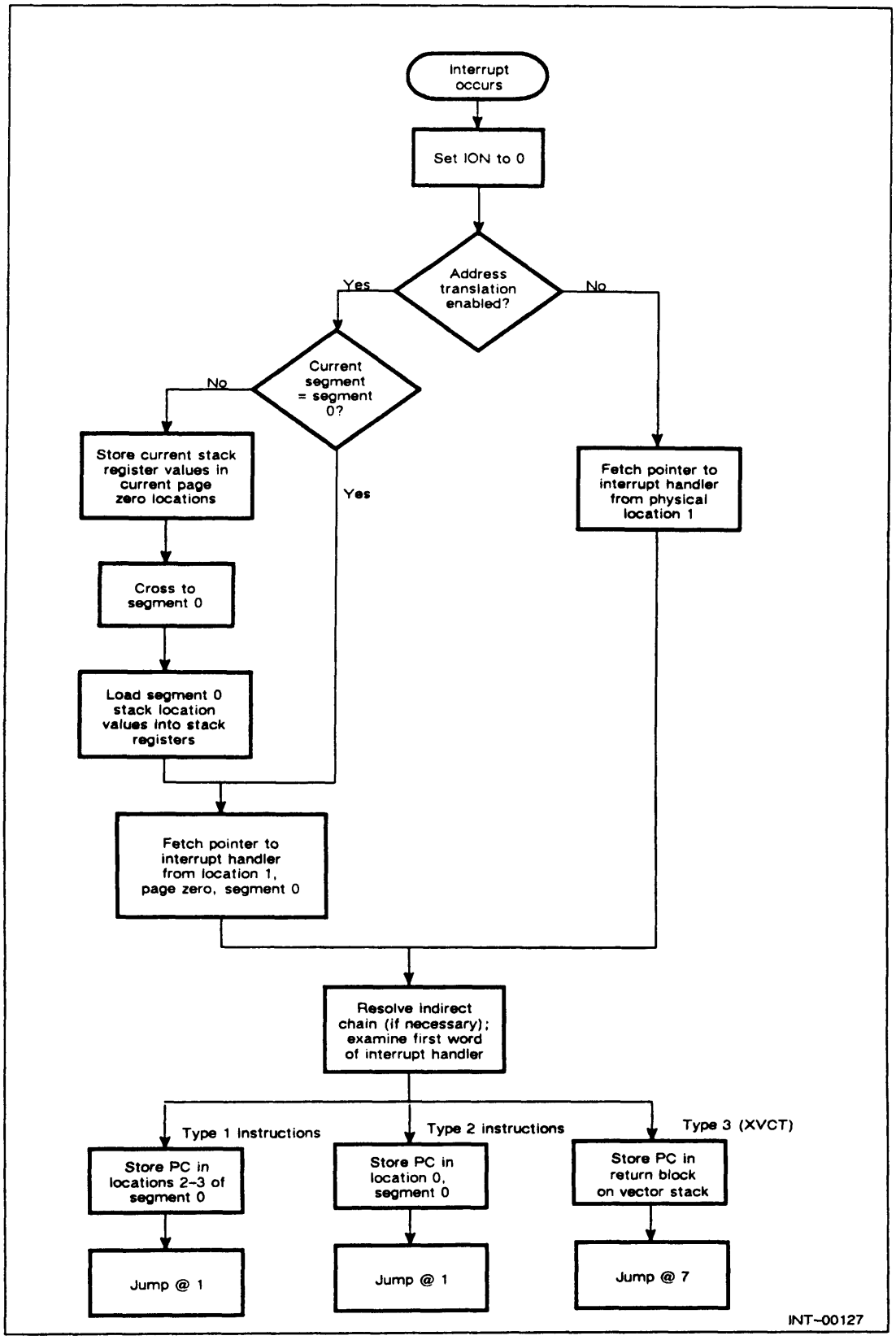


Figure 8-3 *Interrupt sequence*

Vectored Interrupt Processing

When the processor executes a vector interrupt (XVCT) instruction, the processor (Figure 8-4) tests the contents of the interrupt-level word in location 0 of page zero reserved memory for segment 0. If this location equals

- zero, then the processor begins base-level interrupt processing.
- nonzero, then the processor begins intermediate-level interrupt processing.

The processor, in either case, increments the interrupt-level word by one.

NOTE: *Software, as part of the interrupt return, must decrement the interrupt-level word by one.*

Base-Level Interrupt Processing

To service a base-level vector interrupt, the processor must be executing in segment 0. If the current segment of execution is segment 1 through 7, the processor

- Saves the wide stack pointer and the wide frame pointer in the reserved memory locations of the current segment. (The wide stack base and wide stack limit contents are the same as the reserved memory contents.)
- Crosses to segment 0.

To service a base-level vector interrupt, the processor

1. Saves the wide stack parameters (the wide stack registers and the pointer to the wide stack fault handler) from the reserved memory locations of segment 0 in an internal processor state.
2. Uses the three vector stack parameters in reserved memory (locations 4_8, 6_8, and 7_8) to initialize the four wide stack registers and wide stack fault pointer for the vector stack. Loading the vector stack information enables vector stack underflow and overflow detection.

   - Vector stack pointer parameter (location 4_8)

     The processor, interpreting the parameter as a 16-bit word, zero-extends the vector stack pointer before loading it into the wide stack base, wide stack pointer, and wide frame pointer registers.

   - Vector stack limit parameter (location 6_8)

     The processor, interpreting the parameter as a 16-bit word, zero-extends the vector stack limit before loading it into the wide stack limit register.

     NOTE: *The 16-bit vector stack base and limit parameters initially restrict the vector stack to the lower 128 Kbytes of segment 0.*

   - Vector stack fault address parameter (location 7_8)

3. Pushes the previously saved wide stack parameters from the internal processor state onto the vector stack.
4. Pushes a wide return block onto the vector stack.
5. Continues execution as described in the section, "Final Interrupt Processing."

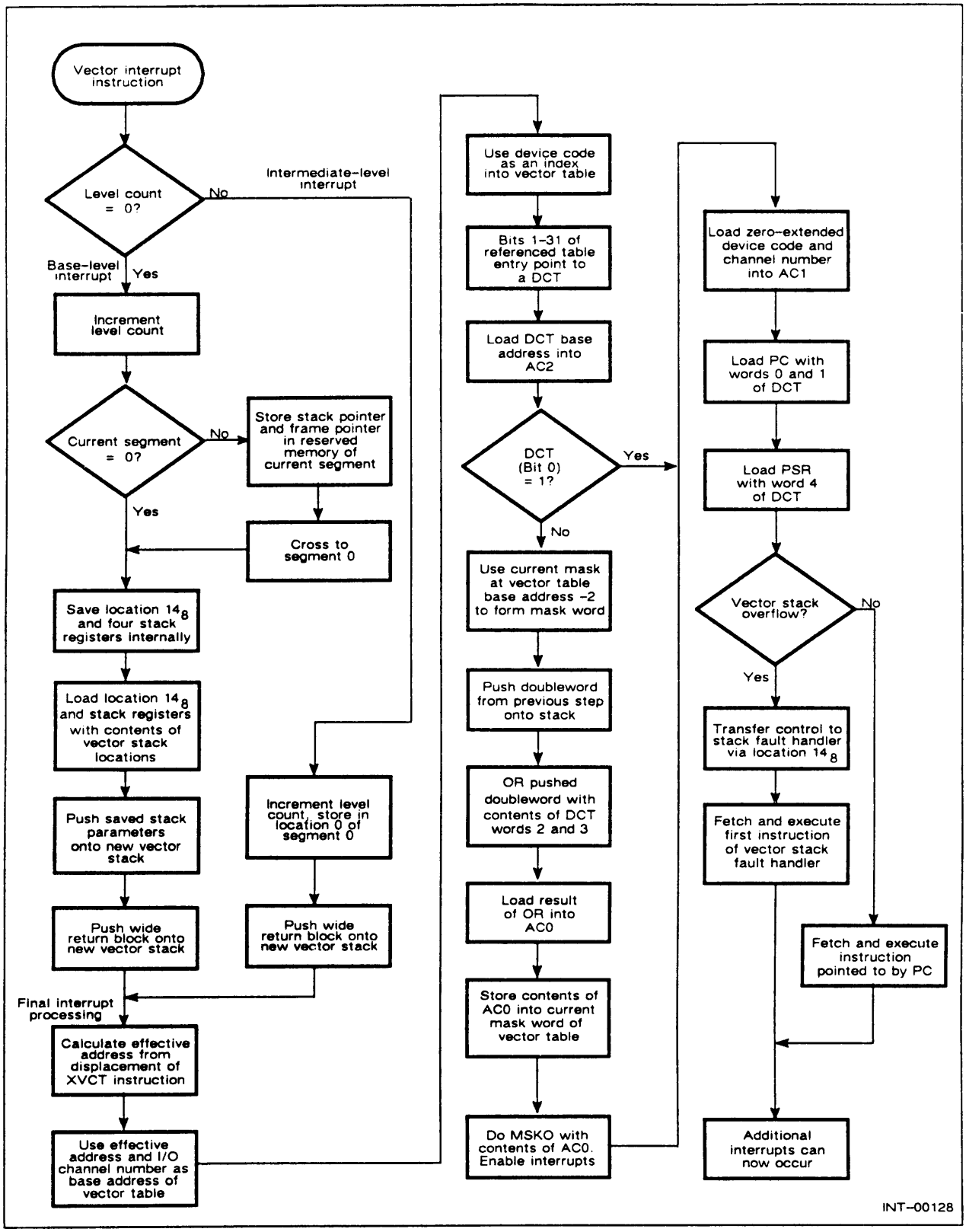


Figure 8-4 *Vectored interrupt processing sequence*

Intermediate-Level Interrupt Processing

The processor begins intermediate-level interrupt processing with the current segment equal to segment 0. To service an intermediate-level vector interrupt, the processor

1. Pushes a wide return block onto the vector stack.
2. Continues execution as described in the section, "Final Interrupt Processing."

Final Interrupt Processing

To complete the vector interrupt servicing (Figure 8-5), the processor

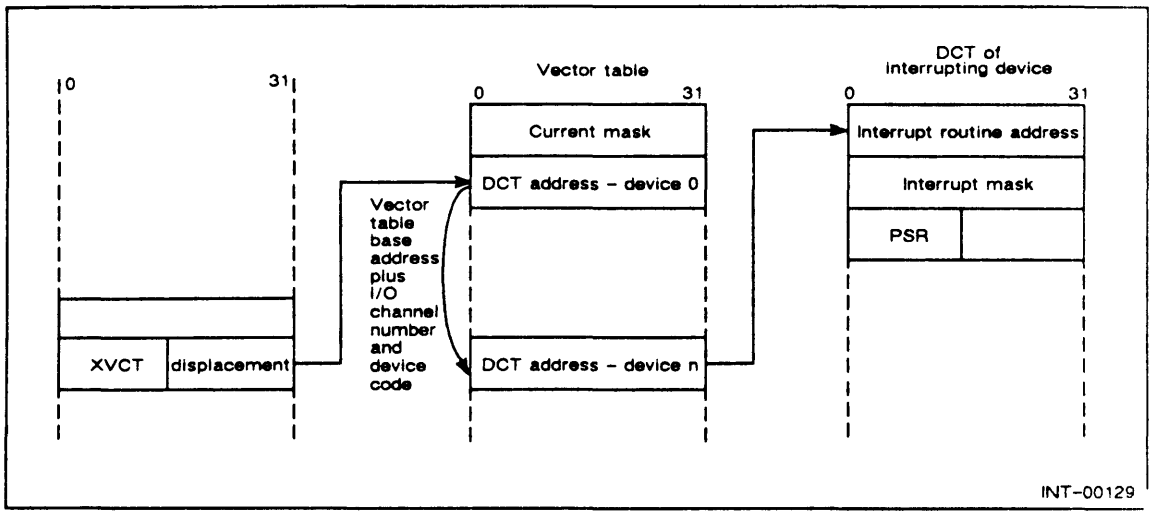


Figure 8-5 *Sequence of actions to conclude interrupt service*

1. Calculates the effective address from the displacement of the **XVCT** instruction. The indirection chain, if any, is narrow.

   There is one vector table for each I/O channel (0-6). The effective address identifies word 0 of the vector table for the currently interrupting I/O channel. Each table contains 64 doubleword entries (one entry for each device on the I/O channel). Figure 8-6 illustrates the vector table; Table 8-5 describes the contents of each table.

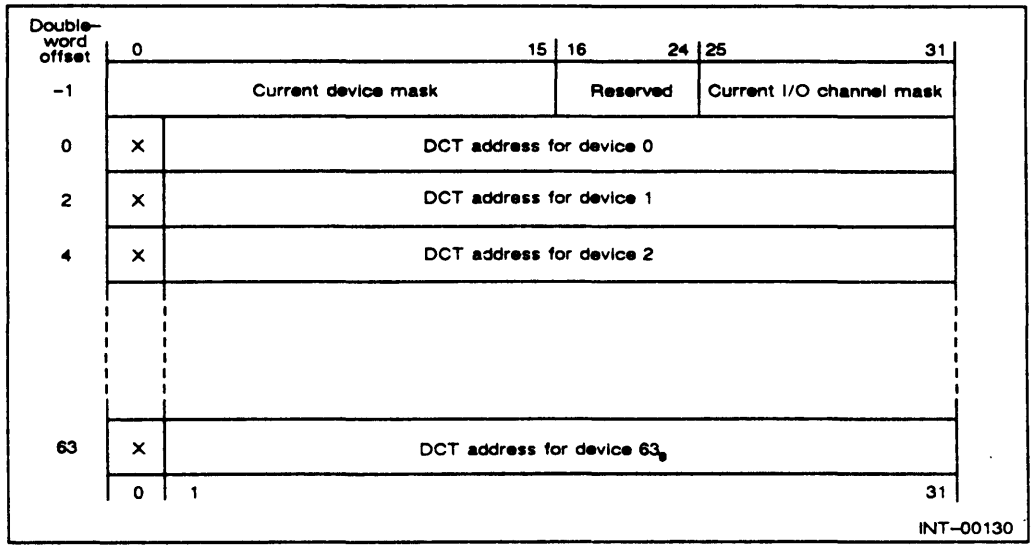


Figure 8-6 *Vector table*

Table 8-5 *Vector table contents*

Contents	Bits	Meaning
Doubleword -1		
Current device mask	0-15	Contains the current device mask. The processor uses this value (and the current I/O channel mask) to form a new device mask.
Reserved	16-24	Reserved for future use and must be set to 0.
Current I/O channel mask	25-31	Contains the current I/O channel mask. Each bit corresponds to an I/O channel number. The processor uses this value (and the current device mask) to form a new device mask.
Doublewords 0 through 63_8		
X	0	Unused and may be set to 0 or 1.
DCT address for device *n*	1-31	Contains 31-bit address to a device control table. Each address points to the base of the table for the interrupting device code (in the range, 0 to 63_8).

2. Uses the interrupting device code number as a doubleword offset from the base of the vector table to address an entry.

   Bits 1-31 of the vectored entry contain the base address of a device control table (DCT). The first five single words of the device control table are defined by the **XVCT** instruction. These words must be set up as shown in Figure 8-7 (Table 8-6 describes the contents of these words). In addition, you can build the device control table with more words to store device-dependent variables and constants for use by the device interrupt routine.

3. Loads AC2 with the base address of the device control table.

   NOTE: *On machines capable of supporting multiple processors (CPUs), if DCT doubleword offset 0 bit 0 equals 1, proceed to step 8. This allows a program to optionally not use the built-in masking functionality.*

4. Constructs a doubleword and pushes it onto the vector stack.

   Bits 0-7 of this doubleword contain all zeros. Bits 8-31 contain the current mask values (doubleword -1) from the vector table (bits 8-15 contain the current device mask and bits 16-31 contain the current I/O channel mask).

5. Loads AC0 with the inclusive OR of the pushed doubleword and the corresponding values from doubleword 1 of the device control table (I/O channel mask and device mask).

6. Stores AC0 into the appropriate locations of the current mask (doubleword -1 of the vector table). Bits 8-15 are stored in the current I/O channel mask; bits 16-31 are stored in the current device mask.

7. Performs the function of a mask out (**MSKO**) instruction with AC0 and enables interrupts. (Bits 8-15 are placed into the I/O channel mask register; bits 16-31 are placed into the device priority mask register.)

   When a mask bit equals one, the processor disables interrupt recognition of devices on the I/O channel that use that mask bit.

8. Loads the least significant bits of AC1 with the interrupting I/O channel number and device number, placing 0s in bits 0-23.

9. Loads the program counter with the address of the device interrupt routine (bits 1-31 of doubleword 0 of the device control table).

10. Initializes the processor status register using the contents of the PSR value (bits 0-15 of doubleword 2) in the device control table.

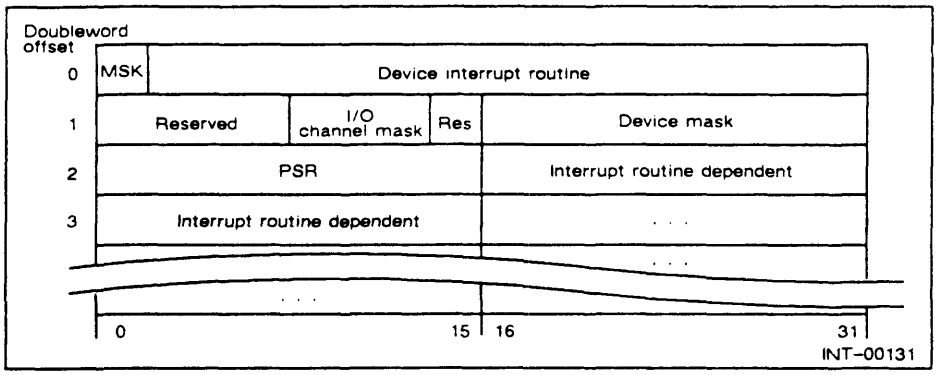


Figure 8-7 *Device control table (DCT)*

Table 8-6 *Device control table contents*

Contents	Bits	Meaning
Doubleword 0		
MSK	0	Enables masking function (applies only to multiple-processor systems). If 0, use built-in masking function.
Device interrupt routine	1-31	Contains 31-bit address of the interrupt routine for this device. The processor places this value in the program counter.
Doubleword 1		
Reserved	0-6	Reserved for future use and must be set to 0.
I/O channel mask	7-14	Contains the I/O channel mask. Each bit corresponds to an I/O channel number, for instance bit 14 equals I/O channel 7 (see the I/O channel mask register description in this chapter). A bit set to 1 masks interrupts on that I/O channel; a bit set to 0 enables interrupts from that channel.
Res	15	Reserved for future use and must be set to 0.
Device mask	16-31	Contains 16-bit device mask specific to this device. The processor uses this value when creating a new device mask.
Doubleword 2		
PSR	0-15	Contains new processor status register value.
Interrupt routine dependent	16-31	May be used for additional information.
Doublewords 3 through n		
Interrupt routine dependent	0-31	May be used for additional information.

11. Checks for a vector stack overflow.

    If the processor does not detect a vector stack overflow, it continues with step 12.

    If the processor detects a vector stack overflow, it transfers program control to the vector stack fault handler. The processor executes the first instruction of the vector stack fault handler before honoring further interrupts.

12. Executes the instruction addressed by the program counter.

    The processor executes the first instruction of the interrupt handler before honoring further interrupts.

NOTE: *On machines capable of supporting multiple processors, the processor executes the first instruction of the interrupt or vector stack fault handler. Then, if bit 0 of the value in AC2 equals 0, the processor honors further interrupts.*

The processor requires that the pointer chain — from the interrupt handler, to the vector table, to the device control table, and finally to the interrupt routine — remain in segment 0.

Interrupt Service Routines

This section is provided for programmers who write their own interrupt service routines. Interrupt service routines for device controllers supported by Data General Corporation's operating systems are included in the operating system software. In addition, DGC operating systems provide a system call that allows users to define non-DGC device controllers or applications-specific peripheral devices for which the user has written special device-driver routines. Refer to the respective operating system programmer's manual for information on the system call.

The interrupt service routine (or handler) must save the state of the processor, identify which device controller requires service, and service the device controller.

Saving the state of the processor involves saving

- the contents of any accumulators that will be used in the interrupt service routine,
- Carry, if necessary,
- the stack and frame pointers.

When the first instruction of the interrupt handler is a vector interrupt (**XVCT**) instruction, the instruction saves the state of the processor. The previous section, "Vectored Interrupt Processing," describes the **XVCT** instruction procedure.

There are various ways in which the interrupt handler can identify which device controller requires service. The interrupt handler can

- perform a polling routine. This routine is generally a sequence of I/O Skip (**SKP**) instructions that tests the states of the Done flags of all device controllers in use. With this method, device controller priorities are determined by the order in which the tests are performed. Note that the polling technique disregards the state of the device controllers' interrupt disable flags. Device controllers that are masked out will be recognized if their Done flags are 1, even though these device controllers could not have caused the interrupt.
- issue an Interrupt Acknowledge (**INTA**) instruction. This instruction reads the device code of the highest priority device controller requesting an interrupt into a specified accumulator. Note that with this method the device controller interrupt disable flags are significant. Device controllers that are masked out cannot respond to the **INTA** instruction.
- issue a Programmed I/O (**PIO**) instruction with an accumulator containing an **INTA** instruction. This procedure is identical to issuing an **INTA** instruction and is used in systems with multiple I/O channels.
- issue a vector interrupt (**XVCT**) instruction. This instruction determines which device controller requires service in the same way as the **PIO** instruction with an **INTA** instruction. In addition, the device code is used to vector automatically to the correct device controller service routine. The **XVCT** instruction also saves the state of the processor and performs other operations necessary to the handling of priority interrupts.

After determining which device controller requires service, the interrupt handler generally transfers control to a device controller service routine. This routine performs the information transfer to or from that device controller (if required) and either starts the device controller on a new operation or idles the device controller if it has no more operations pending.

When all service for the device controller has been completed, either the device controller service routine or the main interrupt handler must perform the following sequence to dismiss the interrupt:

- Signal the device controller to set its Done flag to 0 to dismiss the interrupt request that was just honored. If this is not done, the undismissed interrupt request will cause another interrupt — this time incorrectly — as soon as the interrupt handler finishes and attempts to return control to the interrupted program.
- Restore the pre-interrupt states of the accumulators, Carry, the stack pointer, and address translation unit.
- Set ION to 1 to enable interrupts again.
- Return to the interrupted program.

The instruction that enables interrupts (usually **INTEN**) sets ION to 1, but the processor does not allow the state of ION to change to 1 until the next instruction begins. Thus, after the instruction that turns interrupts back on, the processor starts executing one more instruction before another interrupt can be recognized. The interrupt (or device) handler should issue a return instruction (to the interrupted program) immediately after enabling interrupts. This prevents a waiting interrupt from overwriting the previously saved contents of the interrupted program's program counter before it is used to return control to the interrupted program.

Priority Interrupt

If ION remains 0 throughout the interrupt service routine, the routine cannot be interrupted and there is only one level of device priority. All device controllers that have not been disabled by the program are, for the most part, equally able to request interrupts and receive interrupt service. Only when two or more device controllers are requesting an interrupt at exactly the same time is a priority distinction made. When this happens priority is determined either by the order in which the I/O Skip instructions are given or, by an **INTA** or **XVCT** instruction, or by the order of the device controllers along the I/O bus. The program interrupt facility hardware and instructions allow the program to implement up to 16 interrupt priority levels.

In a system with device controllers of widely differing speeds and/or service requirements, a more extensive priority structure may be necessary. In order to avoid losing data, a program interrupt scheme must allow a slower device to interrupt a faster device. This involves creating a multiple-level priority structure and assigning a slower device to a higher priority level.

In general, a multiple-level priority interrupt scheme is used to allow higher-priority device controllers to interrupt the service routines of lower-priority device controllers. A hierarchy of priority levels can be established through program manipulation of the interrupt disable flags of all device controllers in the system. When the interrupt request from a device controller of a certain priority is honored, the interrupt handler sets up the new priority level. The handler does this by establishing new values for the interrupt disable flags of all device controllers according to an appropriate interrupt priority mask used with the **MSKO** instruction.

Device controllers whose interrupt disable flags are set to 1 by the corresponding bit of this priority mask are masked out (disabled) and are thereby regarded as being of lower priority than the device controller being serviced. Before proceeding with the device service routine, set ION to 1 so that the higher-priority device controllers may interrupt the current service routine.

Interrupt Priority Mask

The bit of the priority mask that governs the interrupt disable flag for a given device controller is assigned to that device controller by the hardware and cannot by changed by the program. Although lower-speed devices are generally assigned to higher-numbered mask bits, no implicit priority ordering is intended.

The manner in which these priority levels are ordered is completely up to the program. By means of the priority mask, the program can establish any desired priority structure, with one limitation: in the cases in which two or more device controllers are assigned to the same bit of the priority mask, these device controllers are constrained to be at the same priority level. When a device controller causes an interrupt, a decision must then be made to place all other device controllers that share the same mask bit with the interrupting device controller at a higher or lower priority level. If you mask out all device controllers that share that priority mask bit, the interrupting device controller is also masked out.

Priority Interrupt Handler

A priority interrupt handler differs from a single-level interrupt handler in several ways. The handler must be re-entrant (no information is lost that the handler will need to restore the state of the machine if a device controller service routine is interrupted by another, higher-priority device controller). Two additional items of information that should be saved (besides those saved by a single-level interrupt handler) are the return address to the previously interrupted program and the current priority mask. A standard method of storing return information for a re-entrant interrupt handler is through the use of the stack.

The interrupt handler (including the device controller service routines) for a multi-level priority scheme should perform the following tasks:

- Save the state of the processor (the contents of the accumulators, Carry, the return address to the previously interrupted program, the stack parameters, and the current priority mask).
- Identify the device controller that requested the interrupt.
- Transfer control to the service routine for that device controller.
- Establish the new priority mask with a **MSKO** instruction for that device controller's service routine and store it in memory at the location reserved for the current priority mask at that level of interrupt.
- Clear the current device interrupt.
- Enable interrupts (any device controller not masked out can now interrupt this service routine).
- Service the device controller that requested the interrupt.
- Disable interrupts in preparation for dismissal of this interrupt level, so that no interrupts will occur during the transition to the next lower level.
- Restore the previously saved state of the processor, and reinstitute the pre-interrupt priority mask with a **MSKO** instruction.
- Enable interrupts.
- Transfer control to the return address of the previously interrupted program that was saved.

The **XVCT** instruction performs the necessary initial tasks for a priority interrupt handler.

Data Channel/Burst Multiplexor Channel

The data channel (DCH) provides I/O communication for medium-speed devices and synchronous communications. The burst multiplexor channel (BMC) is a high-speed communications pathway that transfers data directly between main memory and high-speed peripherals. I/O-to-memory transfers for both DCH and BMC always bypass the address translator. The DCH and BMC use a map to control the transfer of data, with the BMC capable of operating in either mapped or unmapped mode. The ECLIPSE MV/Family instructions load map slots and return DCH and BMC status information.

The DCH and BMC provide access to memory for individual device controllers on demand. Devices that use either channel operate under a priority structure imposed on them by their respective channel (generally set by either their physical slot location, jumpers, or hardware switches on the device controller board). When one or more device controller requests access, priority is given to the device controller on the

- DCH that is closest to the data channel bus controller on the I/O bus.
- BMC that is assigned highest bus priority by its hardware.

A data channel or burst multiplexor channel transfer is set up with a program that specifies the

- I/O channel to be used for the transfer.

  In a multiple-channel environment, the system uses the default I/O channel for ECLIPSE 16-bit I/O instructions. The I/O Channel Select (**PRTSEL**) instruction can be used to change the default I/O channel. In a single-channel environment, the **PRTSEL** instruction performs no operation.

  NOTE: *On powerup or after a system reset, channel 0 is the default I/O channel. An I/O reset does not change the default.*

- Direction of the transfer (read or write).

  A read is a transfer of data from the device to the processor or physical memory (data in); a write is a transfer from the processor or memory to the device (data out).

- Address of the first word to transfer.

  The device transmits a word address to a *device map*. A device map is a set of map registers that control the addressing of memory for the data transfer.

- Total number of words to transfer.

Transfer Sequence

The entire I/O transfer sequence is synchronized by clock signals generated by the respective bus controller (BMC or DCH). For timing information, refer to the machine-specific interface designer's guide. The actual transfer sequence is a two-way communication between the bus controller and the device controller that proceeds as follows.

When a device controller has a word or block of data ready for transfer to memory or wants to receive data from memory, it issues a request to its respective bus controller. If the device controller has bus priority and no other controllers are active on that channel, the bus controller begins the cycle by acknowledging the device controller's request. The acknowledgment signal causes the device controller to send back to the bus controller the

- starting memory address.
- direction of the transfer.
- intelligent device controllers may also send the type of transfer (data or map load).
- BMC device controllers may also specify the mode of addressing (physical or logical) for the transfer.

Following the receipt of the address, the data itself is transferred in the specified direction on the appropriate bus.

Each time a data transfer is completed, the interaction between the bus controller and the device controller is over. The device controller carries out any tasks necessary to complete the data transfer, such as transferring the data to the device itself for an output operation.

As each word (or word of a burst) is transferred, the device controller increments its memory address register to point to the next memory location. If the word/block counter is used as a

- word counter, the device controller also increments the counter as each word (or word of a burst) is transferred.
- block counter, the device controller increments the counter only when each block has been transferred.

When the word/block counter becomes 0, the device controller typically terminates further transfers, sets its Busy flag to 0 and its Done flag to 1, and initiates a program interrupt request.

If the counter has not yet overflowed, the device controller continues the operation, issuing another request when it is ready for the next transfer.

Device Maps and Data Transfers

A memory allocation and protection (MAP) feature provides allocation of memory for I/O access. The MAP allows physical memory to be allocated in 2-kilobyte blocks (pages). During I/O operations, these pages are selected with a logical address that the MAP translates into a physical address for accessing memory.

The DCH or BMC facility uses a device map in either unmapped or mapped mode.

- In unmapped mode, the processor passes the word address directly to memory, as a physical address. You can use the load physical address (**LPHY**) instruction to translate a logical address to a physical address and store it in an accumulator. (The logical address must point to a current or higher number segment.) Then, send the physical address to the device, using an I/O instruction.
- In mapped mode, the processor uses the device map and the word address to translate the most significant bits of the logical address to a physical page number. The processor then concatenates the physical page number to the 10 least significant bits of the logical address to form the physical address.

Some I/O device controllers can load their own mapping information. Both the DCH and BMC allow device controllers to load their own map locations (slots) — upstream map loading. (Map slots for device controllers that do not include this provision are loaded by program control.)

Upstream map loading is performed by the device controller requesting access to its respective bus controller. When access is granted, the device controller specifies a map load operation along with a map slot address and then proceeds with the transfer as previously described. However, the data transferred to the respective channel bus controller is placed into the I/O map that is contained within the respective bus controller rather than being transferred into memory.

Table 8-7 lists the I/O instructions that affect a device map (a DCH map or BMC map).

NOTE: *Loading a data channel map from a device while DCH mapping is disabled (bit 14 of the I/O channel definition register is set to 0) will produce undefined results.*

Table 8-7 *I/O instructions for DCH/BMC maps*

Instruction	Operation
CIO, CIOI	Returns BMC/DCH status or loads map registers from accumulators ($^1/_2$ slot at a time).
IORST *	Sends a reset signal to all devices on all I/O channels to clear their states and turns off DCH and BMC mapping (clears bits 0, 3, 4, 7, 8, 9, and 14 of the I/O channel definition register).
WLMP	Loads BMC/DCH map slots from memory (as a block of doublewords).
LPHY	Translates a logical address to a physical address, loading the result into an accumulator (for use in the unmapped mode).

* *ECLIPSE compatible instruction*

The **CIO, CIOI,** and **WLMP** instructions initiate DCH/BMC map loads and return status information when in mapped mode. Use the **LPHY** instruction for map loads in unmapped mode. The DCH or BMC sets its Busy flag to 1 when a map load or read is in progress. Neither channel has a Done flag, and the channels never cause program interrupts.

Once you initialize the device, the transfer takes place in two phases.

1. The device driver initializes a device map with the starting word address of the block or subblock to transfer, with the number of words to transfer, and with the direction of the transfer.

2. The data channel or burst multiplexor channel facility transfers the data between the device and memory.

For large transfers, repeat the two phases until the processor transfers the total number of words.

DCH/BMC Maps

The map controlling a DCH or BMC transfer is a series of contiguous map slots (see Figure 8-8). Each map slot contains a pair of map registers: an even-numbered register and its corresponding odd-numbered register. The I/O map table can define up to 512 map slots for DCH operations and up to 1,024 map slots for BMC operations. Each map slot provides the physical page starting memory address for a 2-Kbyte block of memory as well as page access protection.

Note that either BMC or DCH mapping must be enabled before it can be performed.

- DCH mapping is enabled by setting the DCH map enable (DME) bit of the I/O Channel Definition register to 1. When this bit is 0, all DCH addresses are unmapped.

- BMC mapping is enabled by a control signal supplied by the BMC device controller. This signal is sent for each transfer sequence. Enabling or disabling BMC mapping is a function of the device controller and is not governed by the program.

DCH Maps

ECLIPSE MV/Family computer systems support 16 data channel maps, each of which contains 32 map slots. With every data transfer the DCH sends a logical address to the processor. The processor translates the logical address into a physical address using the appropriate map slot for that address.

NOTE: *Loading a data channel map from a device with DCH mapping disabled (bit 14 of the I/O channel definition register) will produce undefined results.*

BMC Maps and Address Modes

The device controller performing the data transfer controls the BMC. No program control or processor interaction is required, except when setting up the BMC's map table. The BMC contains its own map and has two modes of addressing. Each device controller sends a 20-bit address which the BMC map uses as a logical or physical address, depending on the addressing mode.

The BMC uses its map to translate logical page numbers into physical ones. (On some machines that implement it, the Store State Pointer instruction defines the memory locations of the BMC map.) The map table contains 1024 map registers, with each odd-numbered register containing a physical page number and each even-numbered register containing access data (see the following sections for the register contents). The BMC uses the logical page number as an index into the map table, and the contents of the selected map register become the high-order bits of the physical address.

The device controller specifies whether the BMC will operate in unmapped mode (physical) or mapped mode (logical).

- In *unmapped* mode, the BMC receives a 20-bit address from the device controller and uses this as the base physical memory address for the data transfer. As the BMC transfers each data word between the device and memory, it increments the base address, moving successive words to or from consecutive memory locations.

- In *mapped* mode, the BMC receives the 20-bit address from the device controller and uses the high-order 10 bits of this logical address to form a logical page number. The BMC map translates this logical page number into a 10-bit physical page number. This page number combines with the 10 low-order bits from the controller's logical address, to form a 20-bit base physical address, which the BMC uses to access memory.

  Note that when the BMC performs a mapped transfer, it increments the base address after it moves each data word. If the increment causes the 10 low-order bits to overflow, a new map register is selected for subsequent address translation. Depending on the contents of the map table, the BMC may not be able to transfer successive words to or from consecutive pages in memory.

Loading Maps From an I/O Device

In addition to the normal configuring of the map by the processor, the map can be upstream loaded from a device controller. In this operation, the device controller performs two map operations that write

- for the DCH, two 16-bit words into a map slot: one to the high-order register and one to the low-order register. These registers may be written to in any order or, if an existing map is being modified, only one register need be loaded. Note that bit 15 of the address word defines which register (high-order or low-order) receives the data word.

- for the BMC, any number of 16-bit words into map slot registers, beginning with a specified register. The device controller specifies the number of map slot registers to write to and the beginning register address. Note that two words are required to load each map slot; one for the high-order register and one for the low-order register. To load a number of complete map slots, the beginning register address must be even and an even number of 16-bit words must be specified.

DCH/BMC Registers

ECLIPSE MV/Family systems contain 512 DCH slots and 1024 BMC slots. Each 32-bit slot consists of two 16-bit map registers. These map registers and the I/O channel registers are numbered from 0 through 7777_8, as depicted in Figure 8-8 and explained in Table 8-8. The DCH and BMC map registers contain page number and access information. The I/O channel registers contain status and control information which affect DCH and BMC maps and data transfers. The figures and tables that follow describe the formats for each of the registers.

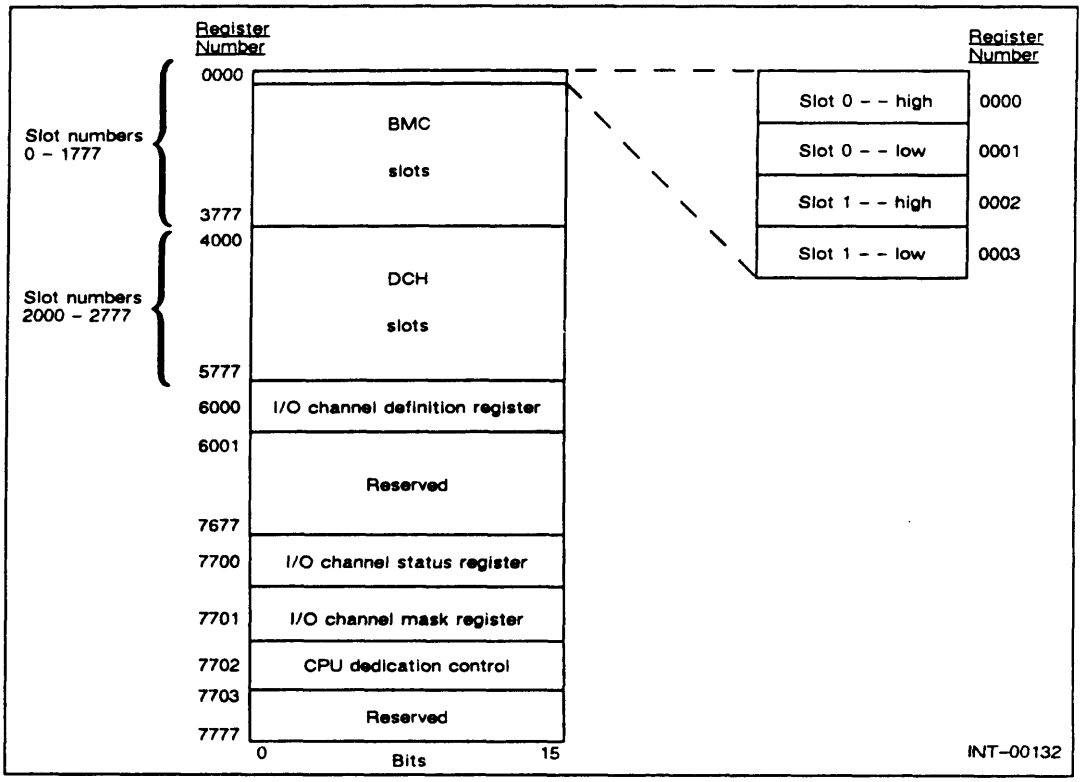


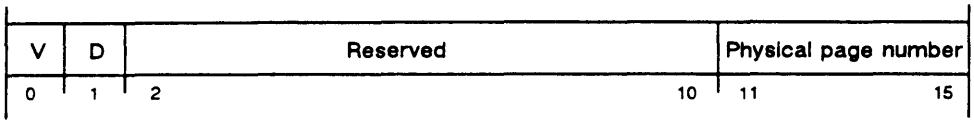
Figure 8-8 *DCH/BMC registers*

Table 8-8 *I/O registers*

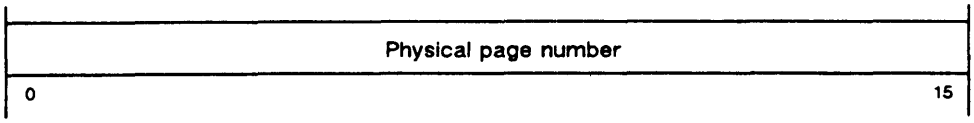
Registers (Octal)	Description
0000-3776	Even-numbered registers are the most significant half of BMC map slots 0-1777.
0001-3777	Odd-numbered registers are the least significant half of BMC map slots 0-1777.
4000-5776	Even-numbered registers are the most significant half of DCH map slots 0-777.
4001-5777	Odd-numbered registers are the least significant half of DCH map slots 0-777.
6000	I/O channel definition register.
6001-7677	Reserved.
7700	I/O channel status register.
7701	I/O channel mask register.
7702	CPU dedication control.
7703-7777	Reserved.

BMC/DCH Slot Register Formats

The processor translates the contents of the BMC and DCH even address registers ($0000-3776_8$ and $4000-5776_8$, respectively) as diagrammed below.



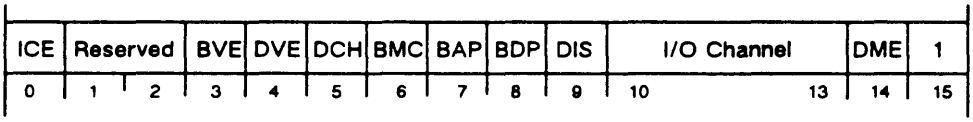
The processor translates the contents of the BMC and DCH odd address registers ($0000-3777_8$ and $4001-5777_8$, respectively) as diagrammed below.



Bits	Name	Contents or Function
0	V	Map validity bit If 0, processor allows access to page. If 1, processor denies access to page.
1	D	Data bit If 0, the channel transfers data to device or memory. If 1, the channel transfers zeros to device or memory.
2-10	Reserved	Reserved for use by the hardware. Write to with zeros; reading these bits returns an undefined state.
11-15 (odd) 0-15 (even)	Physical Page Number	The physical page number which the map uses when translating from a logical to a physical address.

I/O Channel Definition Register Format

The I/O channel definition register (6000_8) provides error and status information.

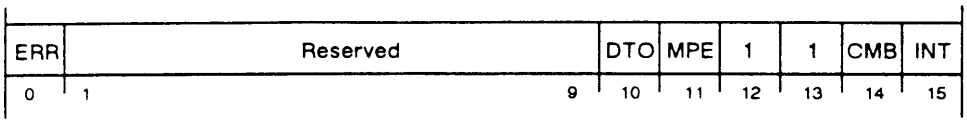


NOTE: *Writing a 1 into bits 3, 4, 7, or 8 complements these bits. The* **IORST** *and* **PRTRST** *instructions clear bits 3, 4, 7, 8, 9, and 14.*

Bits	Name	Contents or Function
0	ICE	I/O channel error flag; if 1, an error has occurred on the I/O channel (0 only when all other error bits are 0 — read-only bit).
1, 2	Reserved	Reserved for future use and returned as zero.
3	BVE	BMC validity error flag; if 1, BMC address validity protect error has occurred.
4	DVE	DCH validity error flag; if 1, DCH address validity protect error has occurred.
5	DCH	DCH transfer flag; if 1, a DCH transaction is in progress (read-only bit).
6	BMC	BMC transfer flag; if 1, a BMC transfer is in progress (read-only bit).
7	BAP	BMC address error; if 1, the channel has detected an address parity error.
8	BDP	BMC data error; if 1, the channel has detected a data parity error.
9	DIS	Disable block transfer; if 1, disables BMC block transfers to and from I/O memory port.
10-13	I/O channel	I/O channel number.
14	DME	DCH mode; if 1, DCH mapping is enabled.
15	1	Always set to 1.

I/O Channel Status Register Format

The read-only I/O channel status register (7700_8) provides I/O channel status information.



Bits	Name	Contents or Function
0	ERR	Error. If 1, the I/O channel has detected an error. This bit is set to 1 if any error-indicating bit in the IOC status register is set to 1.
1-9	Reserved	Bits 1 through 9 are reserved for future use.
10	DTO	DCH time-out error. If 1, a DCH read-modify-write operation time-out error has occurred.
11	MPE	Map parity error. If 1, a map parity error has occurred.
12	1	Always set to 1, indicating extended DCH map slots and operations are supported.
13	1	Always set to 1.
14	CMB	Current state of the mask bit for this I/O channel (refer to the I/O channel mask register format description).
15	INT	Interrupt pending; If 1, the channel is attempting to interrupt the CPU.

I/O Channel Mask Register Format

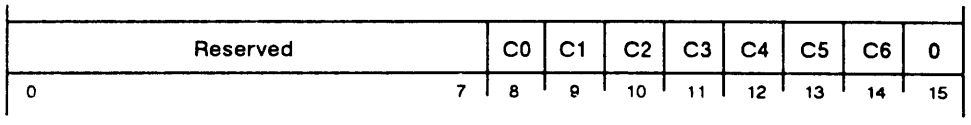
The write-only I/O channel mask register (7701_8) specifies a mask flag for each channel. When an I/O channel mask flag is set to 1, the processor ignores all interrupt requests from devices on that channel.

The Interrupt Acknowledge instruction (**INTA**) using an I/O channel number from 0 through 6 returns the device code of the highest priority interrupting device on that channel, which has its Done flag set. With channel 7, the **INTA** instruction returns the device code of the highest priority interrupting device on the highest priority channel, regardless of the state of the I/O channel mask register flags.

An I/O Channel Reset instruction (**PRTRST**) zeroes the mask bit for a single channel (0 through 6) or for all channels (7).

NOTE: *A* **CIO** *read to the I/O channel mask register produces undefined results.*

The format of the I/O channel mask register is as diagrammed below.



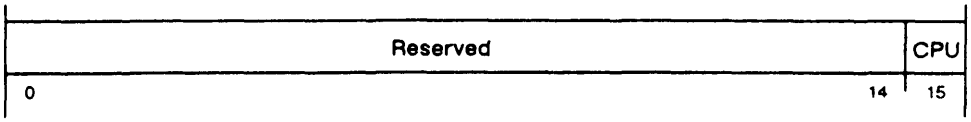
Bits	Name	Contents or Function
0-7	Reserved	Reserved for future use; should be set to 0.
8	C0	I/O channel 0 mask *
9	C1	I/O channel 1 mask *
10	C2	I/O channel 2 mask *
11	C3	I/O channel 3 mask *
12	C4	I/O channel 4 mask *
13	C5	I/O channel 5 mask *
14	C6	I/O channel 6 mask *
15	0	Reserved and set to 0.

* *If 1, prevents all devices connected to the indicated I/O channel from interrupting the processor. A system reset sets C0 to zero and C1 through C6 to ones.*

CPU Dedication Control Register Format

Each I/O channel contains a 16-bit register (7702_8), that controls which processor is to receive I/O interrupts. This register is applicable only to multiple-processor configurations; systems which support only a single processor ignore the contents of this register. This read/write register is available only while a multiple-processor system is in dedicated mode; an attempt to read or write this register while the system is in any mode other than dedicated may cause unpredictable results (refer to the section, "Multiple Central Processing Units," for more information on operating modes).

The format of the CPU dedication control register is as diagrammed below.



Bits	Name	Contents or Function
0-14	Reserved	Reserved for future use; must be set to 0.
15	CPU	Processor number to which all NOVA type interrupts (except cross interrupts) will be directed. On a system reset, this number is set to the value of the initial processor. Upon execution of an IORST instruction, this number is set to the value of the processor that issued the IORST instruction.

NOTE: *The actual number of "Reserved" and "CPU" bits implemented is dependent on the number of processors in the system.*

Device Controllers

Device controllers for either the data channel or burst multiplexor channel are similar in structure. From a programming point of view, the typical device controller operates as a collection of data registers, control registers, and status flags through which the program communicates. With these registers and flags, the program can route data between the computer and a device and monitor the operation of the device.

The distinction between registers and flags is generally one of information content:

- A flag contains a single bit of information.
- A register is made up of a number of bits (generally 16 or 32 bits).
- Fields are groups of bits in a register that convey a single piece of information.

The following describes the basic components of a typical device controller. The information is meant only to typify the workings of a device controller since each device controller is tailored to the specific device it controls. Refer to the programmer's reference for each device for a complete description of the device controller.

The information that the computer and a device controller transfer is generally one of three types: status, control, or data. Status information (from the device controller to the computer) informs the computer about the state of the peripheral. Control information (from the computer to the device controller) tells the peripheral what to do. Data originates from, or is sent to, the device during read or write operations.

Device Controller Registers

Registers in a device controller are classified according to the kind of information they store: data, control, and status. This classification is only a general one, since a register may contain more than one type of information. For instance, a register that serves as a control register when loaded by a program may serve as a status register when read by a program. Programmed I/O instructions designate the direction of an I/O transfer (input or output) and which device controller register (A, B, or C) is to receive the data. For example, the **DOB** instruction outputs data from an accumulator to the B register of a device controller.

Device controllers generally include one or more of the following:

- status register and/or flags (normally the Busy and Done flags),
- control register,
- data registers (or buffers),
- word (or block) counter,
- memory address register,
- burst counter (BMC devices only).

Each device controller contains the program interrupt components and the interface logic for their respective bus (BMC device controllers also contain the interface logic to the data channel). Other controller components, generally available to the program, are in the form of additional control and status registers.

Status Flags and Registers

The two fundamental status flags in a device controller are the Busy and Done flags (see the "Device Flags" section). Status registers (indicating the state of the device) consist primarily of status flags but can also contain control parameters. The control parameters contained in status registers are commonly those that change during the device operation.

Briefly, to place a device in operation, the program signals the device controller to set its Busy flag to 1 and its Done flag to 0 (both flags remain in these states for the duration of the operation, indicating that the device is in use). When the device completes its operation, the device controller sets the Busy flag to 0 and the Done flag to 1. The setting of the Done flag to 1 can be used to trigger a program interrupt. Whether or not a program interrupt occurs depends on the state of the interrupt facility. Regardless of the current state of the interrupt facility, no interrupt can occur for that device until its Done flag is set to 1. Therefore, setting the Done flag to 1 initiates a program interrupt request. At this point, the program can either start the next operation by signaling the device controller to set its Done flag to 0 and its Busy flag to 1; or the program can clear (idle) the device by signaling the device controller to set both flags to 0.

For a relatively simple device, the Busy and Done flags alone may furnish enough status information to allow the program to service the device adequately. A more complex device, however, will generally require additional status flags (generally a status register) to specify its internal operating conditions more completely to the program. The difference between these additional status flags and the Busy and Done flags is that the Busy or Done flags may be tested directly with a single I/O instruction while any other status flag requires that its value first be read into an accumulator from the device's status register.

Status flags that indicate errors or malfunctions in the operation of a device are either passive or active (according to their effect on the device operation when they are set). A device controller sets

- a passive error flag in the course of the operation when the associated error occurs with no immediate indication of this type of error given to the program. In this case, the operation is allowed to continue to completion.
- an active type of error flag when the program attempts to start an operation or, during an operation, attempts to perform a process that is not allowed. In this case, the operation never begins and the Done flag is set to 1 immediately to notify the program.

Control Registers

Control registers allow the program to supply the device controller with the information necessary to operate the device, such as drive or transport numbers, data block sizes, and command specifications. Control parameters (units of control information) typically allow the program to select one of a number of device units in a subsystem, the operation to perform, and the initial values for flags and counters in the device controller. The program transfers control parameters to the device controller with an I/O instruction specifying an accumulator containing the desired parameters.

Data Registers

Data registers (or data buffers) store data in the device controller as it passes between the device and the computer. Data buffers must contain the data word (or the entire number of data words for each BMC burst transfer) to be transferred before the actual input transfer can begin. Likewise, the data buffer must be able to receive the data word (or the entire number of data words for each BMC burst transfer) before an output transfer can begin. To properly transfer data, device controllers generally have two sets of data buffers: one for transferring data to or from the device and another for transferring data from or to memory through the channel controller.

- Data buffers of devices that transfer data with programmed I/O are directly accessible to the program (data is transferred between the device controller register and an accumulator by an I/O instruction).

- Devices that transfer data under DCH or BMC control transfer the data between the device controller register and memory automatically (under the control of the appropriate channel controller). Data buffers in device controllers that use the DCH or BMC need not be — and usually are not — accessible to the processor through programmed I/O.

Word/Block Counter

The program loads the word/block counter with the size of the data block to be transferred. This value indicates either the number of words (word counter) or the number of blocks of words (block counter) to be transferred. The counter value loaded should be the two's complement of the block size. The device controller automatically increments this counter by one for each word or block transferred. When the counter overflows, the device controller terminates the transfer and initiates an interrupt to the processor.

The size of the word/block counter varies from one device controller to another, depending on the block size associated with the device. A typical word/block counter has either 12 or 15 bits, allowing for up to 4,096 or 32,768, respectively, words or blocks of words. Since the counter expects a negative value, the most significant bit of the word/block register need not be a 1 (it is not the sign bit for the number). Thus, a word count of 0 is valid; it specifies the largest possible word or block size. Table 8-9 illustrates the correspondence between some word or block counts and the program values that must be loaded into a 12-bit or 15-bit counter.

Table 8-9 *Word/block counter values*

(Negative) Word Count (Decimal)	15-Bit Value (Octal)	12-Bit Value (Octal)
-1	77777	7777
-2	77776	7776
-100	77634	7634
-2048	74000	4000
-4096	70000	0000
-8192	60000	Not applicable
-32767	00001	Not applicable
-32768	00000	Not applicable

Memory Address Register

The memory address register contains the memory address that the device controller will use for the next data transfer. This address can specify either a physical or logical starting memory address. The program loads the memory address register with the memory address of the first word in the block to be transferred. The device controller automatically increments this counter for each word (or block) transferred. Therefore, successive transfers are from consecutive memory locations.

NOTE: *Because it is the logical memory address that is incremented during mapped transfers, successive physical memory locations may not be accessed. If incrementing the logical address results in a carry from the page offset address into the logical page number, the logical page number is incremented and the physical page number is taken from the next sequential logical page number map slot. Therefore, in mapped transfers, successive data words may not be accessed at consecutive memory locations if the transfer overlaps two logical pages.*

BMC Burst Counter

BMC device controllers include a circuit that specifies the size (number of words) of the burst transfer. The size is usually selected with hardware switches or jumpers and is normally not accessible by the program. Data bursts may range from 1 through 256 words, but when more than one BMC controller is connected to the BMC, this value is generally 4, 8, or 16 words.

Device Controller Programming

This section presents an overview of device controller programming. Routines for setting up and initiating I/O operations for device controllers supported by Data General Corporation's operating systems are included in the operating system software. In addition, DGC operating systems provide a system call that allows users to define non-DGC device controllers. Refer to the respective operating system programmer's manual for information on the system call.

Programming a device controller for a block transfer typically involves the following steps:

- Check the device status (usually by testing the Busy flag and/or reading a status word and checking one or more error or ready bits).

  If an error has occurred, ideally the program should take appropriate action.

  If no error has occurred but the device is not yet ready, the program should wait for the device to complete its operation.

When the device is ready, the program should

- Specify where the data block is located on the device, usually by giving a device address (such as, specifying a unit number, channel number, sector number, or whatever may be required by the device).

- Specify where the data block is to be located in memory by loading the memory address register with the memory address of the first word of the block. This may also include setting up the appropriate channel's map.

- Load the word/block counter with the value to specify the number of 16-bit words contained in the data block or the number of blocks to transfer.

- Specify the type of transfer and initiate the operation. If the device is capable of several different operations, specifying the type of transfer usually involves loading a control register in the device controller. The operation itself is usually initiated by a programmed I/O instruction with the start (S) or I/O pulse (P) device flag controls.

  NOTE: *Because of the ECLIPSE MV/Family memory organization, an improvement in system performance may be realized by starting BMC data transfers on quad doubleword memory address boundaries (three least significant word address bits = 000). Also, the word/block count should be even and in multiples of eight 16-bit words.*

Assemble the necessary information in the accumulators and transfer that information to the device controller using programmed I/O instructions.

Where intelligent device controllers are used, the commands to set up the transfer can be assembled in a control block in memory. The control block is then transferred to the device controller under either DCH or BMC control on the respective bus (the program must first supply a starting memory address for the control block and initiate the device controller with programmed I/O instructions). Multiple device controller operations can be performed without additional program intervention when a group of control blocks are linked together.

Setting up and initiating the I/O operations are the major parts of programming either a DCH or BMC block transfer. If any errors could have occurred during the operation, the program should check for these errors when the operation is complete and take appropriate action.

Data Transfer Latency

Systems that depend heavily on I/O transfers may overload the I/O facilities. This overloading means that certain devices may lose data or have poor performance because the system cannot respond to them in time.

Programmed I/O

Nearly all devices operating under programmed I/O request processor service by setting their Done flag to 1. The processor determines that the Done flag is 1 either by using the program interrupt facility to respond to interrupt requests or repeatedly checking the device with programmed I/O instructions (polling). Programmed I/O latency is the delay between the time that a device requests service and the time that the processor carries out that service.

When using a polling routine, programmed I/O latency has two components:

- The interval between the time the Done flag is set to 1 by the device and the time the flag is checked by the processor. This component can be diminished by performing frequent checks on the Done flag.

- The time required by the device service routine to transfer data to or from the device and set the Done flag to 0 (by idling the device or instructing it to begin a new operation). This component can be diminished with an efficient device service routine.

When the processor interrupt facility is used, programmed I/O latency has at least four components:

1. The time from the setting of the Done flag to 1 to either the end of the instruction being executed by the processor or to the point where an interruptible instruction can be interrupted.
2. The time the interrupt facility needs to store the program counter and enter the interrupt handler (this component is a fixed time).
3. The time required by the interrupt handler to save the state of the machine, identify the device, and transfer control to the service routine. (This component time is fixed when using the **XVCT** instruction; otherwise this time is determined by the software that handles the interrupt.)
4. The time required by the service routine to transfer data to or from the device and set the Done flag to 0. (The service routine software determines this time.)

Programmed I/O latency may be extended by three other time periods:

5. When processor operation is suspended because of other system activity. (This time is dependent on the nature of the activity and the number of other activities in progress.)
6. When the processor does not respond to the device's interrupt request because the interrupt system is disabled (for example, during the servicing of an interrupt from another device). (The service routine software determines this time.)
7. When the device's interrupt disable flag is set to 1 during the servicing of an interrupt of a higher priority device. (The service routine software determines this time.)

Components 4, 6, and 7 account for the bulk of programmed I/O latency.

Any device that must wait too long for program service from the processor may suffer from degraded performance. The maximum programmed I/O latency for a device is the longest allowable delay between the time that a device sets its Done flag to 1 and the time that the processor transfers data to or from that device and sets the Done flag to 0.

When the actual programmed I/O latency for a device exceeds the maximum programmed I/O latency, the specific effects depend on the device in question. (In the worst case, data may be incorrectly read or written.) The maximum allowable programmed I/O latencies for each device may be found in the programming manual for the device.

A device service routine must usually perform certain computations (updating pointers to buffers, byte counters, etc.) but rarely are these computations so complex that they cannot be accomplished within the constraints of the maximum allowable programmed I/O latency. If several devices are competing for service at the same time, however, it may be necessary to jeopardize the performance of some devices by deferring their request for program service until the processor has serviced the higher priority requests. For this reason, ECLIPSE MV/Family computers incorporate the priority interrupt facility.

The object of the priority interrupt facility is to minimize the loss of data. Thus the assignment of the software priority levels should be made with the following considerations in mind:

- The maximum allowable programmed I/O latency for each device.
- The result of exceeding the maximum allowable programmed I/O latency for each device (slowdown or data loss).
- The cost of losing data.

Data Channel and Burst Multiplexor Channel

Time constraints may also be encountered when transferring data via the data channel or burst multiplexor channel. When a device needs service, it makes a request. There may be more than one device, however, waiting to access the DCH or BMC at any one time. Consequently, there may be a significant delay between the time when a device requests access to the channel and the time when the transfer actually occurs. This latency also includes the time required to complete transfers to or from any higher priority devices that are also requesting channel access.

The length of DCH or BMC latency depends on the number of DCH or BMC devices operating in the system at a higher priority and the frequency of their use.

Most devices operate under fixed time constraints. For devices such as disk drives, diskette drives, or magnetic tape transports, if data is not read or written at the correct instant, the controller will have to wait for another revolution of the disk, or in the case of tape, reverse tape direction and perform the operation again. (Since BMC device controllers transfer data in bursts, the time constraints are related to the size of the data burst.)

Consequently, on input, such devices must be allowed to write a word (or burst of words) into memory before the next word (or burst) is assembled by the device controller. On output, the device controller must be able to read the data from memory before the surface is positioned under the write head. In either case, if the latency time is too long, data cannot be properly transferred. Most devices operating under either DCH or BMC control set an error flag (data late) when this happens, so that the service routine can take appropriate action to recover from the error, if possible. Most magnetic media device controllers also provide extra levels of data buffering or larger data buffers that usually eliminate data late issues. With these controllers, no data is moved to memory or written to the device until the device controller's buffers are full.

The maximum allowable latency period of a device is the longest time the device can wait for a transfer. During system configuration, DCH or BMC priorities should be assigned to devices on the same basis as programmed I/O devices:

- The maximum allowable latency period of the device. A device with a short allowable latency should usually receive a higher priority than one with a long allowable latency.
- The recovery time of a device (how long before it can repeat a transfer that failed because of excessive latency) if the device can recover.
- The cost of losing data from the device if the device cannot recover.

NOTE: *Device controllers that have the potential for using large portions of the data channel bandwidth and effectively locking out controllers of lower priority should be placed in lower priority positions. These controllers generally include local area networks (LANs), intelligent synchronous controllers (ISCs), network bus adapters (NBAs), and graphics display controllers (GDCs).*

DCH latency might be improved by less frequent use of I/O instructions. In addition, there is an upper limit on the number of DCH transfers per second that an I/O channel can support. In cases where this limit is exceeded, one solution is to reduce the number of devices using the data channel at the same time. Refer to the machine-specific supplement for DCH and BMC bandwidths.

BMC latency might be improved by decreasing the size of the data burst from the BMC device controllers. Reducing the size of the data burst means a BMC device controller must request BMC bus controller service more often and the controller must buffer fewer words before requesting service.

Integral Devices

The following sections of this chapter describe instructions for manipulation of these integral devices:

- central processing unit
- timing mechanisms (architectural clocks or programmable interval timer and real-time clock) *
- primary asynchronous line input/output
- system control processor (or program)
- data channel and burst multiplexor channel
- universal power supply controller †
- power supply controller †

* Either the architectural clocks or the programmable interval timer and real-time clock may be loaded with a microcode load instruction (these timing devices are mutually exclusive).

† ECLIPSE MV/Family systems support either the universal power supply controller (UPSC) or the power supply controller (PSC). Refer to the machine-specific supplement to determine which power supply controller your system supports.

The machine-specific "Standard I/O Device Codes" appendix lists device codes, device mnemonics, and priority mask bit assignments.

Central Processor

Device Code	77_8
Assembler Mnemonic	CPU
Priority Mask Bit	None

The central processor (CPU) is considered an internal device with control and status flags and is accessible using I/O instructions. The control flag is the interrupt on flag; the status flag is the powerfail flag (refer to the section, "General I/O Instructions"). The I/O instructions to the CPU may use either the standard I/O instruction form, or a special CPU-specific form. Some CPU-specific instructions may be interpreted differently from their standard I/O instruction equivalent. For information on ECLIPSE MV/Family systems that may support more than one central processor, refer to the section, "Multiple Central Processing Units."

Device Flag Control

Device flag commands to the CPU determine whether or not the processor can interrupt the current program with a program interrupt request. When the interrupt on flag (ION) equals 1, the processor can interrupt the program (once the instruction following the enable has begun). The processor cannot interrupt the program when the interrupt on flag equals 0. The CPU interrupt on flag is controlled by the device flag commands as follows:

f=omitted	ION unchanged.
f=S	Sets ION to 1.
f=C	Sets ION to 0.
f=P	Causes an unimplemented instruction interrupt.

The assembler interprets the I/O instructions for the CPU using either the standard I/O instruction format or a special I/O instruction format. For instance, the instruction that initializes the devices and sets the priority mask bits to 0 (I/O Reset) uses the following standard form:

DIC*[f]* *ac*,**CPU**

The same instruction can take the following special form:

IORST

The special assembler statement **IORST** is equivalent to the standard assembler statement **DICC 0,CPU**

Both statements set ION and all I/O device Busy and Done flags to 0. A device flag control (**S**, **C**, or **P**) cannot be appended to the special form of a CPU instruction (such as **IORST**).

NOTE: *The assembler detects a fatal format error when a device flag is appended to a special CPU instruction.*

CPU Instructions

Table 8-10 lists the I/O instructions — both standard and special forms — that affect the CPU.

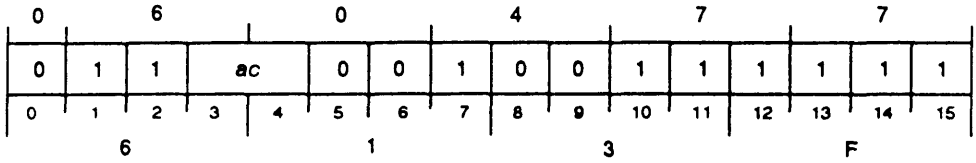
Table 8-10 *I/O instructions for the CPU*

Assembler Statement		Function
Special Form	**Standard Form**	
READS *ac*	**DIA***[f]* *ac*,**CPU**	Returns the code of the device that the system booted from.
PRTSEL	**NIO CPU**	On a single I/O channel machine, performs no operation On a multiple-I/O channel machine, sets the default I/O channel to contents of AC0.*
PRTRST	**PIO** *0,0*	On a single I/O channel machine, performs no operation On a multiple-I/O channel machine, initializes an I/O subsystem.*
INTA *ac*	**DIB***[f]* *ac*,**CPU**	Returns the device code of the interrupting device.
IORST	**DIC***[f]* *ac*,**CPU**	Initializes the I/O system (sets ION to 0, resets the I/O device Busy and Done flags and all the priority mask bits to 0; clears certain CPU registers and disables the DCH mapping and address translator).
MSKO *ac*	**DOB***[f]* *ac*,**CPU**	Initializes or changes the priority mask.
HALT	**DOC***[f]* *ac*,**CPU**	Stops the processor
INTDS	**NIOC CPU**	Disables interrupts (sets ION to 0).
INTEN	**NIOS CPU**	Enables interrupts (sets ION to 1)
SKP*t* **CPU**	**SKP***t* **CPU**	Tests the condition of ION or the powerfail flag, and when true, skips the next word in the program.

* *If a single I/O channel is implemented on a machine capable of supporting multiple I/O channels, these instructions execute as multiple-I/O channel instructions.*

Read Switches — READS

READS *ac*



Function: boot device code → *ac*
unchanged → ION

Parameters: None

NOTE: **READS *ac* = DIA *ac*,CPU**

The Read Switches instruction places the code of the device that the system booted from into the specified accumulator.

Arguments

ac(16–31) After execution contains device code. (Refer to the machine-specific supplement for a list of the device codes.)

Registers, Flags, and Stacks

AC0–AC3 Can be individually specified as *ac*; otherwise unused.

ION Unchanged

Carry Unchanged

Overflow Unaffected

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

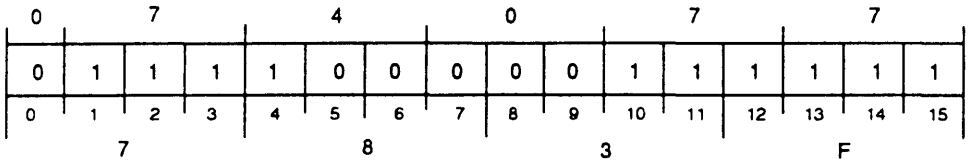
DIA The assembler recognizes **READS *ac*** to be equivalent to **DIA *ac*,CPU**.

Exceptions

None

I/O Channel Select — PRTSEL

PRTSEL



Function: Select or return default I/O channel

Parameters: AC0 = I/O channel → unchanged

NOTE: If AC0 initially = -1, then default I/O channel → AC0

PRTSEL performs no operation on those machines which implement a single I/O channel.

On multiple I/O channel machines (or those machines capable of multiple I/O channels), the I/O Channel Select instruction specifies the default I/O channel that the ECLIPSE 16-bit compatible I/O instructions use.

If bits 16 through 31 of AC0 initially contain -1, then **PRTSEL** places the current default I/O channel into AC0.

PRTSEL unmasks interrupts on the selected channel and masks interrupts on all other channels.

NOTE: *Use this instruction carefully in multiple I/O channel machines.*

NOTE: *In multiple-CPU systems, this instruction changes the default I/O channel on all processors.*

Arguments

None

Registers, Flags, and Stacks

AC0(29-31)	Before execution, contains I/O channel (bits 16 to 28 set to 0). After execution, contents unchanged unless AC0 initially contains all ones in bits 16-31, then the processor returns the current default I/O channel number.
AC1-AC3	Unused
Carry	Unchanged
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

Table 8-11 shows the effect of using the I/O channel numbers with various I/O instructions. The I/O Instruction column indicates the instruction executed (in both standard and special formats). The remaining columns describe the action on the default I/O channel, and as the result of executing the instruction using a Program I/O (**PIO**) instruction to a specific channel. (**PIO** issues a programmed I/O command to an I/O device on a specified I/O channel; refer to the *Instruction Dictionary* for a complete description.) In the table, *n* equals a value in the range of 0 to the maximum number of implemented I/O channels.

Table 8-11 *CPU device instructions with I/O channels*

I/O Instruction	Default I/O Channel	PIO to I/O Channel n*	PIO to I/O Channel 7
READS *ac* **DIA***[f] ac,CPU*	Returns code of device system booted from	Undefined	Undefined
INTA *ac* **DIB***[f] ac,CPU*	Return highest priority device on default channel	Return highest priority device on channel *n*	Return highest priority device on highest priority channel
IORST *ac* **DIC***[f] ac,CPU*	Perform I/O Reset instruction function on all channels	Perform I/O Channel Reset (PRTRST) function on channel *n*	Perform I/O Channel Reset (PRTRST) function on all channels
MSKO *ac* **DOB***[f] ac,CPU*	Mask out devices on default channel	Mask out devices on channel *n*	Mask out devices on all channels
HALT *ac* **DOC***[f] ac,CPU*	Halt the CPU	Undefined	Undefined
INTDS **NIOC** *ac,CPU*	Disable interrupts (ION=1)	Undefined	Undefined
INTEN **NIOS** *ac,CPU*	Enable interrupts (ION=0)	Undefined	Undefined
SKPBN CPU	Skip if ION = 1	Undefined	Undefined
SKPBZ CPU	Skip if ION = 0	Undefined	Undefined
SKPDN CPU	Skip if powerfail = 1	Undefined	Undefined
SKPDZ CPU	Skip if powerfail = 0	Undefined	Undefined

Exceptions

On powerup or after a system reset, the default I/O channel becomes 0.

An I/O reset does not change the default I/O channel.

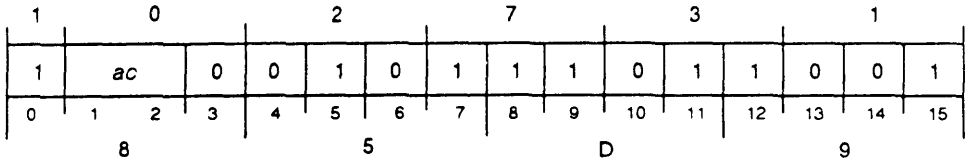
For I/O instructions that specify a device code other than 77_8 (CPU):

- The ECLIPSE 16-bit compatible I/O instructions use the default I/O channel.
- The **PIO** instruction (with an ECLIPSE 16-bit compatible I/O instruction) uses any implemented I/O channel.

In either case, results are undefined with any channel number other than those implemented.

I/O Channel Reset PRTRST

PRTRST *ac*



Function: I/O channel devices → clear states
0 → priority mask
0 → I/O channel mask bit
f → ION

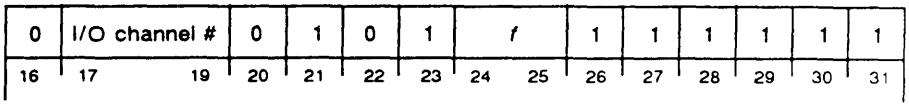
Parameters: *ac* = I/O channel #, ION control → unchanged

PRTRST performs no operation on those machines that implement a single I/O channel.

On multiple-I/O channel machines (or those capable of supporting multiple I/O channels), **PRTRST** sends a reset signal to all devices on the I/O channel specified in *ac*. This signal instructs the devices to clear their states. In addition, the instruction sets the addressed I/O channel's 16-bit priority mask and the mask bit for the addressed I/O channel in the I/O channel mask register to 0. The device control flag (*f*) determines the state of the interrupt on flag (ION). A **PRTRST** issued to an I/O channel resets only that channel; issued to channel 7, it resets all implemented I/O channels.

Arguments

ac(16-31) Specifies I/O channel in bits 17-19 and ION control in bits 24 and 25 (bits 0-15 are undefined). Format is as follows:



Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
Carry	Unchanged
ION	After execution, set according to *f*.
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

PIO The assembler recognizes a **PIO** *acs,acd* instruction to be equivalent to the **PRTRST** instruction if *acs* contains the specified bit pattern (see *ac* description). In this case, *acs* specifies the accumulator containing the I/O channel number, and *acd* is not used.

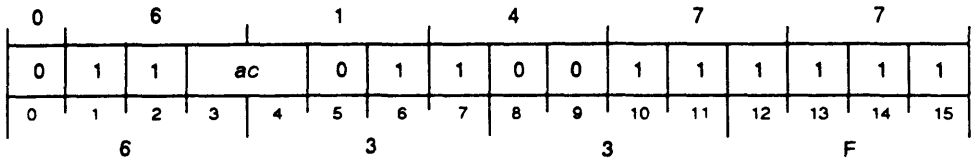
Exceptions

A Command I/O (**CIO**) instruction that reads the I/O channel mask register will have undefined results.

Specifying an unimplemented I/O channel number produces undefined results.

Interrupt Acknowledge — INTA

INTA *ac*



Function: device code → *ac*
unchanged → ION

Parameters: None

NOTE: **INTA** *ac* = **DIB** *ac*,**CPU**

The Interrupt Acknowledge instruction places a device code into the specified accumulator. The code indicates the device requesting an interrupt which has the highest priority on the highest priority I/O channel.

Arguments

ac(26–31) After execution contains device code; bits 0–25 are set to 0.

NOTE: *If you execute an* **INTA** *instruction using the* **PIO** *instruction, the I/O channel number is returned to ac(23–25).*

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
Carry	Unchanged
ION	Unchanged
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

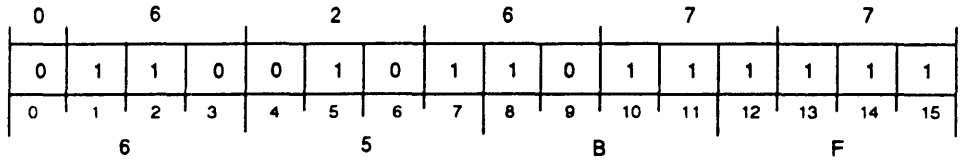
DIB The assembler recognizes **DIB***[f]* *ac*,**CPU** to be equivalent to **INTA** *ac*. The **DIB** form of the Interrupt Acknowledge instruction allows ION to be manipulated.

Exceptions

Do not use the **DIBP** *ac*,**CPU** form for the Interrupt Acknowledge instruction as this bit pattern is reserved for the VCT instruction on some ECLIPSE 16-bit computers.

I/O Reset IORST

IORST



Function: Clear all I/O devices
0 → priority mask
0 → PSR
0 → FPSR(0-8)
0 → ION
0 → Busy and Done flags
off → address translator

Parameters: None

NOTE: **IORST** = **DICC 0,CPU**

IORST sends a reset signal to all devices on all I/O channels to clear their states. The instruction disables logical address translation and sets the following to 0: the 16-bit priority mask, the PSR, bits 0 through 8 of the FPSR, and ION.

NOTES: *In multiple-I/O channel environments,* **IORST** *also sets the following to 0: the I/O channel mask register flag for channel 0, and bits 0, 3, 4, 7, 8, 9, and 14 of the I/O channel definition register (6000_8).*

In multiple-CPU systems, **IORST** *also clears all pending cross interrupts and redirects all IOC traffic to the CPU that issued the* **IORST** *instruction.*

Arguments

None

Registers, Flags and Stacks

AC0-AC3	Unused
Carry	Unchanged
ION	Set to 0
Overflow	Unaffected
PC	PC + 1
PSR	Set to 0
FPSR(0-8)	Set to 0
Stack	Unchanged

Related Instructions

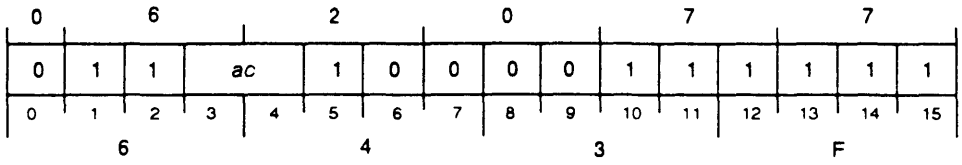
DIC The assembler recognizes **DICC** *0*,**CPU** to be equivalent to **IORST**. The **DIC***[f] ac*,**CPU** form of the I/O Reset instruction allows manipulation of ION. When using the **DIC***[f] ac*,**CPU** form of the I/O Reset instruction, an accumulator value must be coded to avoid assembly errors. During execution, the processor ignores the accumulator field, and the contents of the accumulator remain unchanged.

Exceptions

None

Mask Out MSKO

MSKO *ac*



Function: *ac* → priority mask
unchanged → ION

Parameters: None

NOTE: **MSKO** *ac* = **DOB** *ac*,**CPU**

The Mask Out instruction places the contents of the specified accumulator into the 16-bit priority mask.

NOTE: *Masking out a device when interrupts are enabled is not recommended.*

Arguments

ac(16-31) Before execution, contains new priority mask. A 1 in a bit position disables interrupt requests for devices that use that bit as a mask. (Refer to the machine-specific "Standard I/O Device Codes" appendix for device code mask bits.)

After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3 Can be individually specified as *ac*, otherwise unused.

Carry Unchanged

ION Unchanged

Overflow Unaffected

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

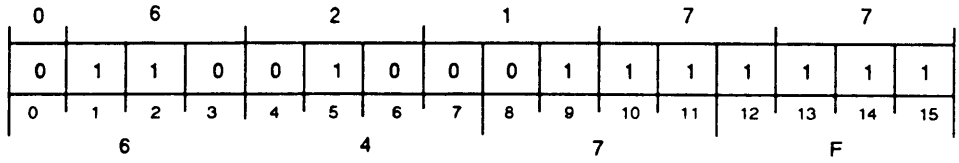
DOB The assembler recognizes **DOB**[*f*] *ac*,**CPU** to be equivalent to **MSKO** *ac*. The **DOB** form of the Mask Out instruction allows manipulation of ION.

Exceptions

None

Halt HALT

HALT



Function: Stops the processor unchanged → ION

Parameters: None

NOTE: HALT = DOC 0,CPU

The Halt instruction stops the processor.

Arguments

None

Registers, Flags, and Stacks

AC0–AC3	Unused
Carry	Unchanged
ION	Unchanged
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

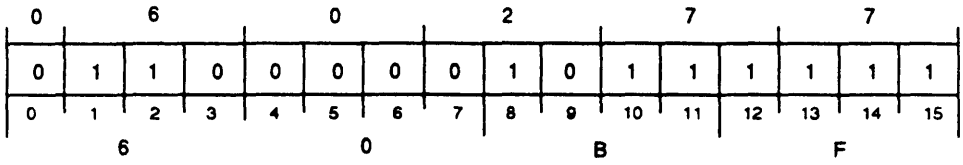
DOC The assembler recognizes **DOC** *0*,**CPU** to be equivalent to **HALT**. The **DOC** form of Halt allows manipulation of ION. When using this form, an accumulator must be coded to avoid assembly errors. During execution, the processor ignores the accumulator field, and the contents of the accumulator remain unchanged.

Exceptions

None

Interrupt Disable INTDS

INTDS



Function: 0 → ION

Parameters: None

NOTE: INTDS = NIOC CPU

The Interrupt Disable instruction sets the interrupt on flag (ION) to 0, thus concealing a device interrupt.

Arguments

None

Registers, Flags, and Stacks

AC0–AC3	Unused
Carry	Unchanged
ION	Set to 0
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

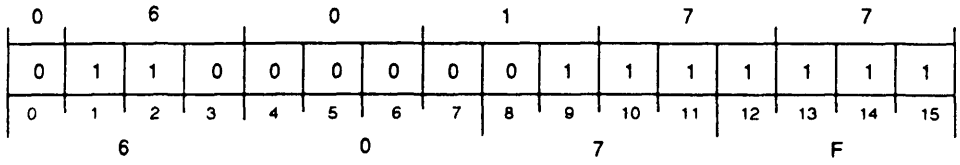
NIO	The assembler recognizes **NIOC CPU** to be equivalent to **INTDS**.

Exceptions

None

Interrupt Enable INTEN

INTEN



Function: 1 → ION

Parameters: None

NOTE: **INTEN = NIOS CPU**

The Interrupt Enable instruction sets the interrupt on flag (ION) to 1, allowing the CPU to recognize a device interrupt.

Arguments

None

Registers, Flags, and Stacks

AC0–AC3	Unused
Carry	Unchanged
ION	Set to 1
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

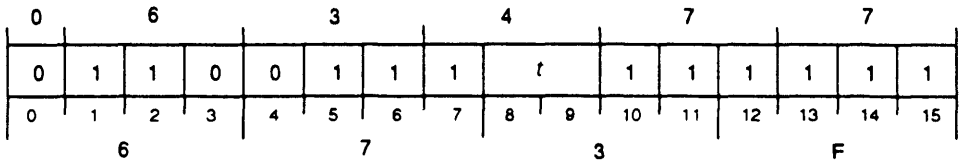
NIO	The assembler recognizes **NIOS CPU** to be equivalent to **INTEN**.

Exceptions

If the Interrupt Enable instruction changes the state of ION, the CPU allows one more instruction to execute before the first I/O interrupt can occur. If, however, the instruction is interruptible, then interrupts can occur as soon as the instruction begins to execute.

CPU Skip — SKP*t* CPU

SKP*t* CPU
(false test return)
(true test return)



Function: If *t* = true then skip
unchanged → ION and powerfail flags

Parameters: None

The CPU Skip instruction tests the specified flag. If the test condition is true, the processor skips the next sequential word.

Arguments

t Specifies the test. The following lists the possible test conditions.

Assembler Code for *t*	Bits 8 9	CPU Flag and Test
BN	0 0	ION = 1
BZ	0 1	ION = 0
DN	1 0	Powerfail = 1
DZ	1 1	Powerfail = 0

Registers, Flags, and Stacks

AC0–AC3	Unused
Carry	Unchanged
ION	Unchanged
Overflow	Unaffected
PC	PC + 1 (false test) PC + 2 (true test)
PSR	Unchanged
Stack	Unchanged

Related Instructions

None

Exceptions

None

Timing Mechanisms

ECLIPSE MV/Family systems support either the Architectural Clocks or a combination of the PIT and the RTC as timing devices; the two clock implementations are mutually exclusive. Systems which provide the option of supporting either type of timing devices expect a value indicating the type to be coded with the Load Control Store (LCS) instruction (refer to the appendix, "Load Control Store Instruction"). Systems which support only one or the other type of timing mechanism load the appropriate microcode.

An ECLIPSE MV/Family system which does not support the Architectural Clocks generates an unimplemented instruction trap if an Architectural Clock instruction is issued.

Architectural Clocks

The Architectural Clocks include

- an alarm clock,
- a time-slice timer,
- a boot clock.

The alarm clock and the time-slice timer both represent a time value as a 64-bit integer with the following clarifications:

- High-order bits (0 through 45) are supported on all ECLIPSE MV/Family systems — bit 31 ticks at a 1.6384 second interval, bit 45 ticks at a 100 microsecond interval.
- The number of low-order bits (46 through 63) clocked is machine-dependent — if supported, bit 63 ticks at a frequency of $2^{18}(10^4)$ hertz.

The boot clock uses a separate time format — refer to the SCP section in this chapter.

After powerup (but before any Architectural Clock instructions have executed), or after an I/O Reset (IORST) instruction, the clocks are in the following condition:

- Alarm clock

  The time-of-day clock value is invalid (on powerup only; IORST leaves the time-of-day clock unaffected).

  The alarm portion of the alarm clock contains a value indicating a number of years in the distant future.

- Time-slice timer

  The contents of the time-slice timer and the time-slice timer fault handler address are undefined.

  Time-slice timer faults are disabled.

- Boot clock — The boot clock is always valid.

The following sections describe the Architectural Clocks and the instructions which affect them.

Alarm Clock

The alarm clock is an I/O device which implements a time-of-day (TOD) clock with an alarm for generating an I/O interrupt.

There is one alarm clock per system. The alarm clock counter has a roll-over of 222.99 years. Setting the date and time base is the responsibility of the operating system — the processor only interprets the clock contents as a 64-bit integer and never performs any conversions on this value.

The device code for the alarm clock is 14_8, with a mask bit of 13. The alarm clock does not support Busy or Done flags. Communication between the processor and the alarm clock is with special alarm clock instructions. Note that the standard I/O instructions (**DIA, DIB, DIC, DOA, DOB, DOC, PIO, SKP*t*,** and **NIO**) to the alarm clock device code produce undefined results. Table 8-12 lists the instructions that affect the alarm clock.

Table 8-12 *Instructions affecting the alarm clock*

Assembler Statement	Function
RTOD	Returns the current time of day.
STOD	Sets the time of day.
ALARM	Sets the alarm value.
INTA	Returns the device code for the alarm clock.
MSKO	Masks out the alarm clock.
IORST	Sets the alarm to a value in the distant future; does not affect the time-of-day portion of the clock.
HALT	Stops the processor, but does not affect on the alarm clock's time-of-day function.

The time-of-day counter is set with the **STOD** instruction and read with the **RTOD** instruction; the alarm value can be set with the **ALARM** instruction. When the **STOD** instruction executes, the TOD counter is loaded with a 64-bit value which represents the current time and date. The TOD counter counts up from this value, keeping the current time without any additional software support.

Two successive **RTOD** instructions will return two different time-of-day values. If the time-of-day counter ever reaches its highest value (-1), it is invalid. An invalid condition may occur when the counter counts up to or through the highest value, or when an **STOD** instruction, containing the highest value, executes. The highest value is relative to the actual number of bits the processor supports. For instance, if your system supports 50 out of 64 bits, then the processor treats all ones in the 50 bits as the highest value.

In the event of an invalid time-of-day, the **RTOD** instruction will return the highest time. Any successive **RTOD** instructions will continue to return this value until the time-of-day is validated by issuing an **STOD** instruction (counting begins after execution of this **STOD** instruction). At powerup, the time-of-day value is invalid; an **STOD** instruction must be executed to set the time-of-day to a valid value.

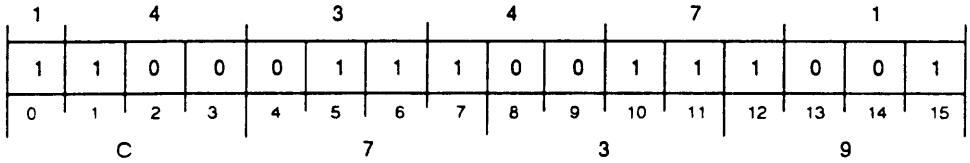
NOTE: *The time-slice functions will still operate correctly even with an invalid time-of-day value.*

To set the alarm, issue an **ALARM** instruction using a value in time-of-day units. When the TOD counter reaches the value specified by the **ALARM** instruction, an alarm interrupt occurs for device code 14_8. The alarm is disabled when the time-of-day is set. If you issue an **ALARM** instruction with a value less than or equal to the current value of the TOD counter, an interrupt is immediately generated. If the time-of-day is invalid, then the **ALARM** value is considered to be less than or equal to the TOD value, and an interrupt is immediately generated.

Read Time of Day

RTOD

RTOD



Function: TOD → AC0&AC1

Parameters: AC0 = ? → high-order TOD value
AC1 = ? → low-order TOD value

RTOD atomically reads the counter portion of the alarm clock, placing the resulting time-of-day value into AC0 and AC1. (An **RTOD** instruction returns a time-of-day value greater than or equal to the value set by the last **STOD** instruction.)

Arguments

None

Registers, Flags, and Stacks

AC0	After execution, contains high-order 32 bits of time-of-day value.
AC1	After execution, contains low-order 32 bits of time-of-day value.
AC2, AC3	Unused
Carry	Unchanged
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

STOD	Set Time of Day

Exceptions

If the time-of-day counter is invalid, **RTOD** returns all ones.

Two successive **RTOD** instructions will return two different values. This also applies to two separate processors in a multiple-processor system attempting to read the TOD counter at the same time.

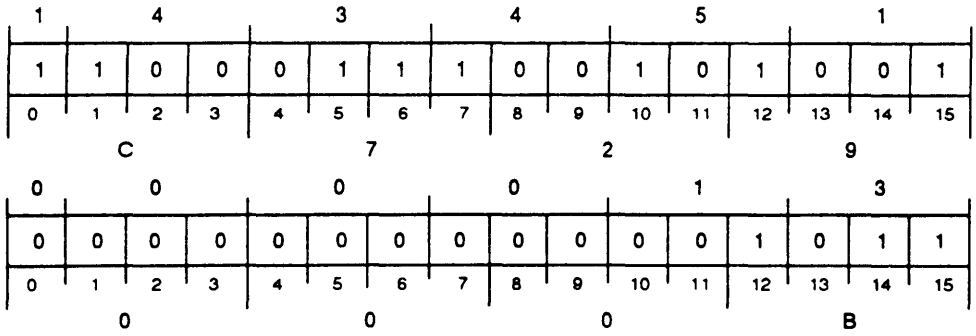
An **RTOD** instruction always returns a time value equal to or greater than the time value specified by the most recent **STOD** instruction. This applies even when the **STOD** instruction specifies nonzero data in unsupported low-order bits.

Set Time of Day

STOD

Privileged Instruction

STOD



Function: Time-of-day value → TOD clock

Parameters: AC0 = high-order TOD value → unchanged
AC1 = low-order TOD value → unchanged

STOD clears any pending alarm clock interrupts, disabling the alarm portion of the alarm clock (no device code 14_8 interrupts can be posted until a subsequent **ALARM** instruction executes). The instruction places the time-of-day value, contained in AC0 and AC1, into the counter portion of the clock. **STOD** then enables the counter, leaving the alarm function disabled.

Arguments

None

Registers, Flags, and Stacks

AC0	Before execution, contains high-order 32 bits of time-of-day value. After execution, contents unchanged.
AC1	Before execution, contains low-order 32 bits of time-of-day value. After execution, contents unchanged.
AC2, AC3	Unused
Carry	Unchanged
Overflow	Unaffected
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load immediate	Use these instructions to place values into AC0 and AC1.
RTOD	Read Time of Day
ALARM	Set Alarm

Exceptions

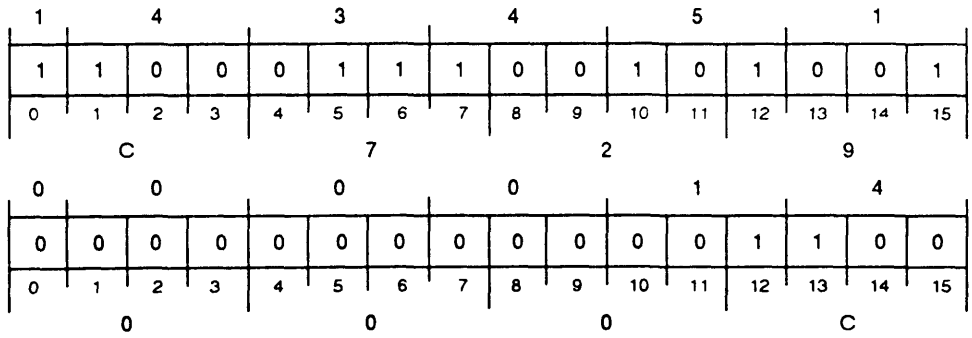
If the time-of-day counter is set to its highest value (all ones), or ever counts to its highest value, the TOD becomes invalid. When the TOD is invalid, the **RTOD** instruction will continue to return all ones until another **STOD** is issued.

Set Alarm

ALARM

Privileged Instruction

ALARM



Function: TOD → alarm

Parameters: AC0 = high-order TOD value → unchanged
AC1 = low-order TOD value → unchanged

ALARM atomically places the time-of-day value, contained in AC0 and AC1, into the alarm portion of the alarm clock. Issuing an **ALARM** instruction clears a pending alarm interrupt.

When the time-of-day counter becomes greater than or equal to the alarm value, the alarm clock generates an interrupt for device code 14_8. If interrupts are disabled (ION equals 0), the interrupt will be held until interrupts are again enabled.

Arguments

None

Registers, Flags, and Stacks

AC0	Before execution, contains high-order 32 bits of time-of-day value. After execution, contents unchanged.
AC1	Before execution, contains low-order 32 bits of time-of-day value. After execution, contents unchanged.
AC2, AC3	Unused
Carry	Unchanged
Overflow	Unaffected
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load immediate	Use these instructions to place values into AC0 and AC1.
RTOD	Read Time of Day
STOD	Set Time of Day

Exceptions

If an alarm value that is less than or equal to the current time-of-day value is loaded into the alarm, the alarm clock immediately posts an interrupt.

Time-Slice Timer

The time-slice ti· : is a count-down timer that causes a time-·lice fault to occur when a specified time-sl. expires. Each processor (in a multiple-proc ssor system) contains one time-slice timer; all timer functions (read time-slice, set time-slice, set fault handler) apply only to the timer associated with that particular processor. The timer interface is in terms of time-of-day units. Table 8-13 lists the instructions that affect the time-slice timer.

Table 8-13 *Instructions affecting the time-slice timer*

Assembler Statement	Function
RTS	Returns the current contents of the time-slice timer.
STS	Sets a time-slice.
STSFH	Identifies routine to handle future time-slice faults.
IORST	Disables the time-slice timer.

When an **STS** instruction executes, the time-slice timer is loaded with a 64-bit value that represents the processor's time slice. The timer counts down from this value and generates a time-slice expiration fault when the timer reaches 0. The time-slice timer continues to count down below 0 after the time-slice fault has been initiated. In order for a time-slice fault to occur, a time-slice fault handler must have been specified using the **STSFH** instruction.

When a time-slice fault is initiated, the processor clears the time-slice fault and disables future time-slice faults. (Though the timer continues to count down below 0, no further time-slice faults are generated on this processor until a subsequent **STS** instruction is issued.)

The actions that then occur depend upon whether or not a **JPLOAD** (or **JPFLOAD**) instruction has been executed on the processor. Note that both **JPLOAD** and **JPFLOAD** instructions apply only to multiple-processor systems (single-processor systems assume that neither instruction is issued). If either of these instructions:

- has been issued — the processor performs a **JPFLUSH** instruction in the current ring of execution, crosses to ring 0, and jumps to the time-slice fault handler.
- has not been issued — the processor crosses to ring 0, pushes a wide return block onto the ring 0 wide stack, and jumps to the time-slice fault handler. Use a **WPOPB** instruction to return from the fault handler.

NOTE: *On a time-slice fault, the contents of the accumulators are undefined. (The old accumulator values are saved in the return block on the stack.)*

The first instruction of the time-slice fault handler executes before interrupts are again acknowledged (the state of ION is unaffected when a time-slice fault is initiated).

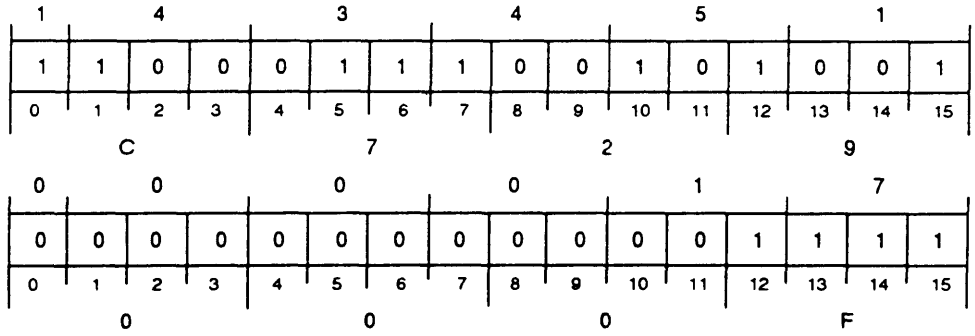
Since the time-slice timer is not an I/O device, neither the I/O channel mask bit nor the **MSKO** instruction will mask a time-slice fault. The interrupt on flag (ION) will mask a time-slice fault (in multiple-processor systems, ION masks an interrupt only for its associated processor). If a time-slice timer counts down to 0 while ION is 0, the pending time-slice fault will be held until ION is set to 1.

Read Time-Slice

RTS

Privileged Instruction

RTS



Function: Time-slice timer value → AC0&AC1

Parameters: AC0 = ? → high-order time-slice value
AC1 = ? → low-order time-slice value

RTS loads the current contents of the time-slice timer into AC0 and AC1. (This instruction has no effect on the time-slice timer.)

Arguments

None

Registers, Flags, and Stacks

AC0	After execution, contains high-order 32 bits of current time-slice timer value.
AC1	After execution, contains low-order 32 bits of current time-slice timer value.
AC2, AC3	Unused
Carry	Unchanged
Overflow	Unaffected
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

STS	Set Time-Slice

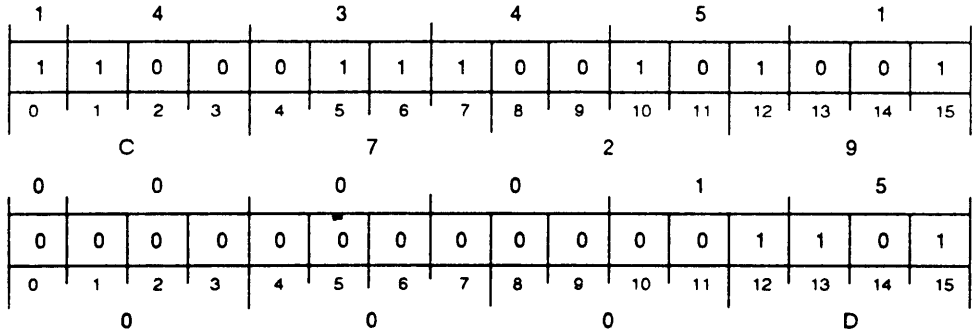
Exceptions

None

Set Time-Slice STS

Privileged Instruction

STS



Function: Time-slice value → timer

Parameters: AC0 = high-order time-slice value → unchanged
AC1 = low-order time-slice value → unchanged

STS clears any pending time-slice expiration faults for this processor. The instruction then loads the time-slice timer value, contained in AC0 and AC1, into the time-slice timer. STS enables time-slice timer expiration faults.

When the residual time-slice decrements to 0, a time-slice fault occurs on the associated processor. If interrupts are disabled (ION equals 0), the fault will be held until interrupts are again enabled. Once the fault is initiated, execution of the first instruction in the fault handler is guaranteed.

The time-slice timer continues to decrement until a new value is loaded with another STS instruction.

Arguments

None

Registers, Flags, and Stacks

AC0	Before execution, contains high-order 32 bits of desired time-slice value. After execution, contents unchanged.
AC1	Before execution, contains low-order 32 bits of desired time-slice value. After execution, contents unchanged.
AC2, AC3	Unused
Carry	Unchanged
Overflow	Unaffected
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load immediate Use these instructions to place values into AC0 and AC1.

RTS Read Time-Slice

STSFH Set Time-Slice Fault Handler

Exceptions

A time-slice expiration fault can not occur until the time-slice fault handler address has been specified by the Set Time-Slice Fault Handler instruction.

If 0 is loaded into the time-slice timer, the time-slice value will continue to be less than or equal to 0, and no time-slice expiration fault will occur for at least 222 years.

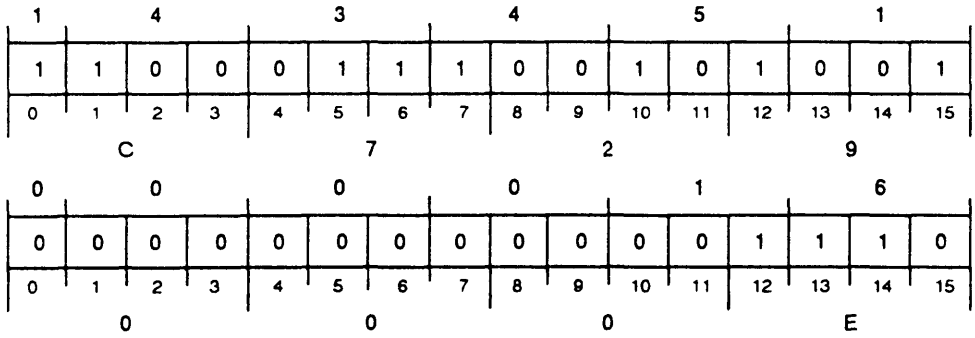
If the **STS** instruction loads a very small value into the time-slice timer, a time-slice fault may be generated immediately. Thus, execution of the instruction following the **STS** instruction is not guaranteed.

Set Time-Slice Fault Handler

STSFH

Privileged Instruction

STSFH



Function: Time-slice handler address → timer

Parameters: AC0 = fault handler address → unchanged

Note: Address must be in logical ring 0.

STSFH identifies the starting address (in segment 0) of the time-slice fault handler. The processor saves this address internally and refers to it when a time-slice fault is initiated.

Arguments

None

Registers, Flags, and Stacks

AC0	Before execution, contains logical ring 0 address of time-slice fault handler. After execution, contents unchanged.
AC1–AC3	Unused
Carry	Unchanged
Overflow	Unaffected
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load effective address
: Use these instructions to place a value into AC0.

Exceptions

The **STSFH** instruction must be issued before a time-slice expiration fault can occur. If an **STS** instruction attempts to force a time-slice fault before the handler address has been defined, the processor ignores the fault and clears the time-slice fault condition.

If the address in AC0 is other than a ring 0 address, a protection fault occurs, and AC1 contains error code 7 (outward ring call).

Boot Clock

Each ECLIPSE MV/Family system contains one boot clock. Access to the boot clock is provided by the SCP interface on device code 45_8. The boot clock can be read after powerup to get the initial time of day. The boot clock parameters are machine dependent, but in general, the clock

- Is powered by a battery during a power interruption.
- Provides at least 1-second resolution (even when on battery).
- Returns at least hours, minutes, seconds, day, date, and year.

Refer to the SCP section for further information on boot clock support.

Programmable Interval Timer

Device Code	43_8
Assembler Mnemonic	PIT
Priority Mask Bit	6 or 11 (See machine-specific appendix, "Standard I/O Device Codes")

The programmable interval timer (PIT) is a CPU-independent time base that is set to initiate program interrupts at fixed intervals ranging from 100 microseconds to 6.5536 seconds in increments of 100 microseconds. The PIT can also be sampled with I/O instructions at any point in its cycle to determine the time that elapses before the next interrupt. Use the PIT 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 processor loads the PIT counter with the contents of the initial count register. The processor then increments the counter at 100-microsecond intervals until the count goes from 177777_8 to 0. If interrupts are enabled, the PIT then initiates a program interrupt request. The value of the PIT counter is undefined after it reaches 0 and initiates an interrupt. (Depending on the machine, the PIT counter either is reset to the contents of the initial count register or continues counting from 0.) A Busy flag and a Done flag control the operation of the device.

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 processor immediately loads the count into the counter. At the first 100-microsecond pulse, the processor again loads the count into the counter. This is done to synchronize the program and the counter.

Device Flag Control

Device flag commands to the PIT start or stop the counting cycle for program interrupts.

f=omitted	Busy and Done flags unchanged.
f=S	Sets the Busy flag to 1 and the Done and interrupt request flags 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.

PIT Instructions

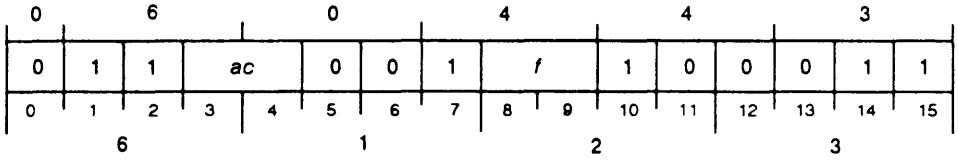
Table 8-14 lists the I/O instructions that affect the PIT device.

Table 8-14 *Instructions affecting the PIT*

Assembler Statement	Function
DIA*[f] ac*,PIT	Places the PIT counter value into the accumulator.
DOA*[f] ac*,PIT	Loads the initial count register with the value in the accumulator.
IORST	Stops the counting cycle and sets the Busy and Done flags, the interrupt mask bit, and the counter to 0.

Read Count

DIA[*f*] *ac*,PIT



Function: PIT counter → *ac*
[f] → Busy and Done flags

Parameters: *ac* = ? → PIT Counter

The Read Count instruction places the value of the PIT counter in the specified accumulator.

Arguments

ac(16–31)	After execution contains the current value of the PIT counter within one count cycle (in two's complement); bits 0–15 are undefined.
[f]	Specify from **S**, **C**, and **P** for desired Busy and Done flag function.

Register, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
Carry	Unchanged
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

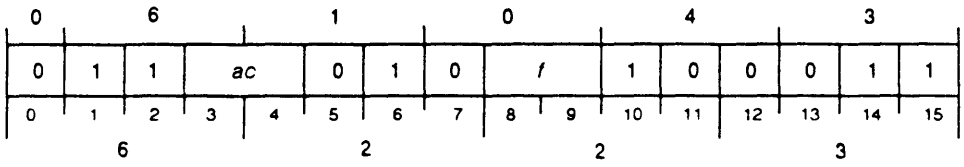
DOA[*f*] *ac*,**PIT** Specify Initial Count

Exceptions

None

Specify Initial Count

DOA[*f*] *ac*,PIT



Function: *ac* → PIT Initial Count register
[*f*] → Busy and Done flags

Parameters: *ac* = Initial count (two's complement) → unchanged

The Specify Initial Count instruction loads the contents of the specified accumulator into the initial count register of the PIT.

Arguments

ac(16-31)	Before execution, contains the signed 16-bit integer specifying the number of 100-microsecond intervals between interrupts. After execution, contents unchanged.
[*f*]	Specify from **S**, **C**, and **P** for desired Busy and Done flag function.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
Carry	Unchanged
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

DIA[*f*] *ac*,**PIT** Read Count

Exceptions

None

Real-Time Clock

Device Code	14_8
Assembler Mnemonic	RTC
Priority Mask Bit	13

The real-time clock (RTC) generates low-frequency I/O interrupts for performing time calculations independent of CPU timing. The interrupts can be used as a time base in programs that require it. The frequency of the clock is program-selectable to 10 Hz, ac-line frequency, 100 Hz, or 1000 Hz. The Busy and Done flags control the operation of the device.

Once the RTC starts, the first program interrupt request can occur at any time up to the selected clock period. After the first interrupt occurs, succeeding interrupts occur at the clock frequency, provided that the program always sets the Busy flag to 1 before the clock period expires.

After powerup or when an I/O Reset (**IORST**) instruction is issued, the processor sets the clock to the ac-line frequency. After powerup, 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.

Device Flag Control

Device flag commands to the RTC enable or disable RTC interrupts.

f=omitted	Busy and Done flags unchanged.
f=S	Sets the Busy flag to 1 and the Done and interrupt request flags to 0; enables RTC interrupts.
f=C	Sets the Busy, Done, and interrupt request flags to 0; disables RTC interrupts.
f=P	No effect.

RTC Instructions

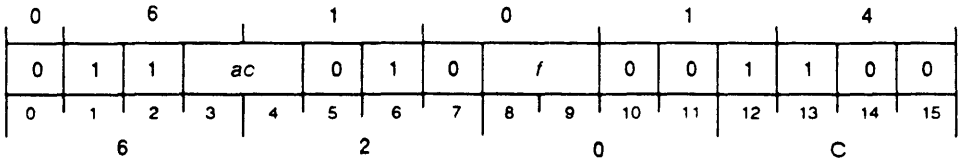
Table 8-15 lists the I/O instructions that affect the RTC device.

Table 8-15 *Instructions affecting the RTC*

Assembler Statement	Function
DOA*[f] ac*,RTC	Loads the RTC with a clock frequency value from an accumulator.
IORST	Disables RTC interrupts and selects the ac-line frequency; also, sets the Busy and Done flags and the priority mask bit to 0.

Select RTC Frequency

DOA*[f]* *ac*,RTC



Function: *ac* → clock frequency

Parameters: *ac* = frequency value → unchanged

The Select RTC Frequency instruction sets the clock frequency according to the contents of the specified accumulator.

Arguments

ac(30–31) Before execution, contains clock frequency (bits 0 through 29 should be set to 0). The clock frequency values are:

Bits 30, 31	Frequency Selected
00	ac-line
01	10 Hz
10	100 Hz
11	1000 Hz

After execution, contents unchanged.

[f] Specify from **S**, **C**, and **P** for desired Busy and Done flag function.

Registers, Flags, and Stacks

AC0–AC3 Can be individually specified as *ac*; otherwise unused.

Carry Unchanged

Overflow Unaffected

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

None

Exceptions

None

Primary Asynchronous Line Input/Output

INPUT		OUTPUT	
Device Code	10_8	Device Code	11_8
Assembler Mnemonic	TTI	Assembler Mnemonic	TTO
Priority Mask Bit	14	Priority Mask Bit	15

The primary asynchronous interface (TTY) is the communication link between the processor and the master terminal. This interface supports asynchronous communication at selected rates from 110 to 4800 baud in 7-bit codes with program-generated parity or 8-bit codes with no parity (depending on the machine, the baud range may be greater). You can use one or two stop bits with either format.

The asynchronous interface consists of separate input (TTI) and output (TTO) devices. As the TTI and TTO devices can generate program interrupts independently, each has its own device code and is controlled by its own set of Busy and Done flags.

The TTY interface is set up to transmit and receive 8-bit characters without parity checking. A process can send or receive 7-bit characters with even, odd, or mark parity by using the high-order bit in the 8-bit character (accumulator bit number 24) as a parity bit. On transmission, the program that drives the interface 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 interface. As the TTI device receives each character, it places the character in an input buffer and sets the Done flag to 1 and the Busy flag to 0. If the program controlling the receiver does not transfer the character before receiving the next character, the contents of the input buffer are overwritten and the previous character is lost. Typically, the intercharacter time at 110 baud is 100 milliseconds; at 4800 baud, the intercharacter time is approximately 2.08 milliseconds.

Device Flags

Device flag commands to the TTY interface determine the flag settings and the transmission of an output character.

f=omitted	Busy and Done flags unchanged.
f=S	Sets the Busy flag to 1 and the Done flag to 0. When the S flag is used with the TTO device, the interface transfers the character from the output buffer to the shifter and begins transmission of the character. The interface sets the Busy flag to 0 and the Done flag to 1 when the character passes from the output buffer to the shifter.
f=C	Sets the Busy, Done, and interrupt request flags to 0.
f=P	No effect.

TTI/TTO Instructions

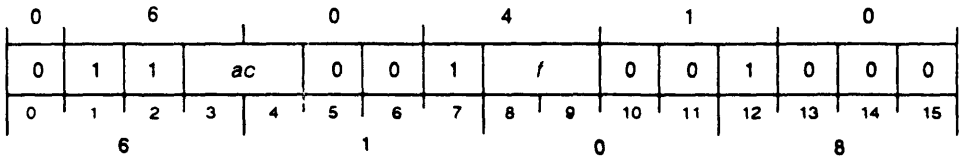
Table 8-16 lists the I/O instructions that affect the TTY interface.

Table 8-16 *I/O instructions for TTI and TTO*

Assembler Statement	Function
DIA*[f] ac*,TTI	Reads a character from the TTI device into an accumulator.
DOA*[f] ac*,TTO	Sends a character from an accumulator to the TTO device.
IORST	Sets the Busy and Done flags and the priority mask bits to 0.

Read Character Buffer

DIA*[f] ac*,**TTI**



Function: TTI (buffer) → *ac*

Parameters: *ac* = ? → character

The Read Character Buffer instruction places the contents of the controller's input buffer in the specified accumulator.

Arguments

ac(24-31) After execution, contains the 8-bit character (or 7-bit character with parity in bit number 24) read from the input buffer.

[f] Specify from S, C, and P for desired Busy and Done flag function.

Registers, Flags, and Stacks

AC0-AC3 Can be individually specified as *ac*; otherwise unused.

Carry Unchanged

Overflow Unaffected

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

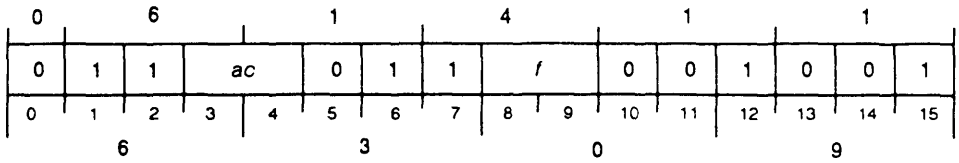
DOA*[f] ac*,**TTO** Load Character Buffer

Exceptions

None

Load Character Buffer

DOA*[f] ac*,TTO



Function: *ac* → TTO (buffer)

Parameters: *ac* = character → unchanged

The Load Character Buffer instruction loads the contents of the specified accumulator into the controller's output buffer.

Arguments

ac(24–31)	Before execution, contains 8-bit character (or 7-bit character with parity in bit number 24) to be placed in the output buffer. After execution, contents unchanged.
[f]	Specify from **S**, **C**, and **P** for desired Busy and Done flag function.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
Carry	Unchanged
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

DIA*[f]* *ac*,**TTI** Read Character Buffer

Exceptions

None

System Control Processor/Program

Device Code	45_8
Assembler Mnemonic	SCP
Priority Mask Bit	15

The system control processor, described in Chapter 1, is a system within most ECLIPSE MV/Family computers that has its own microcomputer. The system control program is a diagnostic-type operating system which initially runs on both the CPU and (in systems which support it) the system control processor. We use the mnemonic, SCP, to refer to both the system control processor and system control program. Communication between the CPU and the SCP is handled by the system control program. When another operating system is running on the CPU, the SCP runs either in the background or on the system control processor.

The SCP runs programs designed to help isolate hardware problems. It maintains an error log which lists the error type, its location, and the time it occurred; and also provides all the system timing for ECLIPSE MV/Family computer systems.

Device Flag Control

Device flag commands to the SCP determine the settings of the Busy and Done flags.

f=omitted	Busy and Done flags unchanged.
f=S	Sets the Busy flag to 1 and the Done flag to 0.
f=C	Sets the Busy and Done flags to 0.
f=P	No effect.

SCP Instructions

Table 8-17 lists the instructions that provide communications between the CPU and the SCP.

Table 8-17 *SCP instructions*

Assembler Statement	Function
DOBS *ac*,SCP	Enables or disables CPU error reporting, and performs the indicated command.
DIBC *ac*,SCP	Returns the current status of the SCP.
SKP*t* SCP *	Tests the SCP Busy/Done flag and skips the next instruction if *t* is true.
NIOC SCP *	Clears the SCP Busy and Done flags; leaves the SCP in diagnostic mode.
IORST *	Disables CPU error reporting.

* *The* **IORST** *instruction is described earlier in this chapter; the* **SKP** *and* **NIO** *instructions are explained in the* Instruction Dictionary. *Note that a* **DIA**, **DOA**, **DIC**, *or* **DOC** *instruction to the SCP is a no-op.*

CPU-to-SCP Protocol

Communication with the SCP generally involves the transfer of data between the CPU and device code 45 (SCP) using the SCP's B register. The Enable/Disable Error Reporting (**DOBS** *ac*,**SCP**) instruction requests information from the SCP using commands coded in an accumulator. The Return SCP Status (**DIBC** *ac*,**SCP**) instruction retrieves the information, placing it into the specified accumulator. Some forms of the Enable/Disable Error Reporting and Return SCP Status instructions require an interface block of one or more words for transferring data between the SCP and the CPU. Depending on the direction of the transfer, we refer to this block as an SCP-to-CPU or CPU-to-SCP buffer.

The CPU/SCP protocol is as follows (see Figure 8-9):

1. The CPU/SCP interface block must have been defined to the SCP using a **DOBS SCP** instruction with a Set Block command (004_8).

2. The CPU must test the availability of the SCP's Busy flag (with the **SKPBZ SCP** or **SKPBN SCP** instruction) and its B register. If the Busy flag is 0, the SCP is ready to accept the next **DOBS** instruction.

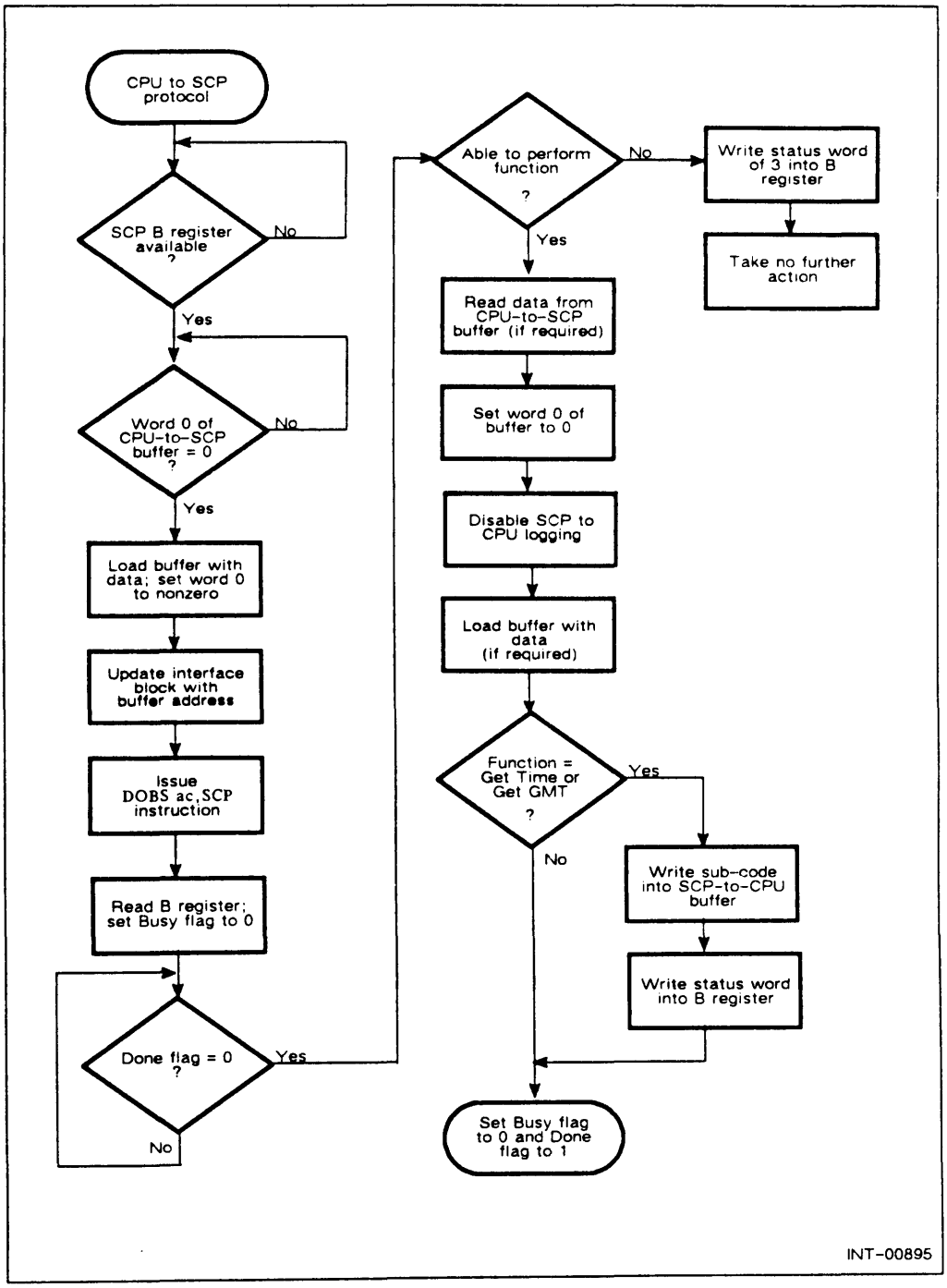


Figure 8-9 *CPU/SCP communications sequence*

3. If data must be transferred from the CPU to the SCP, the CPU should test the CPU-to-SCP buffer for availability. Protocol requires the CPU to set word 0 of this buffer to a nonzero value before using it; the SCP sets word 0 to 0 after the SCP has used the information in the buffer. Therefore, if word 0 of the buffer is nonzero, the CPU waits before placing information into the buffer and sending a function request.

4. The CPU loads the buffer with the appropriate data (see the **DOBS SCP** instruction) and updates the interface block with the buffer address.

5. The CPU issues a **DOBS** *ac*,**SCP** instruction. The accumulator contains the function code (command), a request to enable or disable error reporting, and possibly a new interface block address. The **DOBS** *ac*,**SCP** instruction places the data in the accumulator into the B register of the SCP; the S pulse of this instruction notifies the SCP that the B register is full by setting the Done flag to 0.

6. The SCP reads its B register and sets the Busy flag to 0.

7. The SCP tests the Done flag to ensure that the B register is available for use (Done equals 0).

8. The SCP performs the desired function, reading the data from the CPU-to-SCP buffer (if required).

   If the SCP can not perform the desired function, it writes a status word with a value of 3 into its B register and takes no further action regarding the requested function.

9. The SCP sets word 0 of the buffer to 0 after it has finished using the information stored in that buffer.

10. The SCP disables SCP-to-CPU error logging.

11. The SCP writes the appropriate information into the SCP-to-CPU buffer, starting at offset 1.

    If the CPU command is either a Get Time or Get GMT function, the SCP writes a sub-code value indicating the contents of the buffer into word 0 of the SCP-to-CPU buffer (instead of a 0). The remainder of the buffer contains the data requested. The SCP also writes a status word into its B register to either acknowledge the receipt of the information from the CPU or to tell the CPU the data it has requested is available in the buffer.

12. The SCP writes a status word into its B register (either to acknowledge the receipt of information from the CPU or to tell the CPU the data it has requested is available in the buffer).

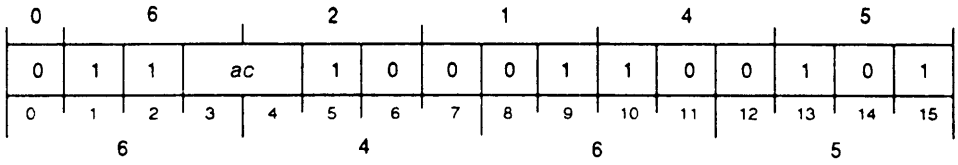
13. The SCP sets the Done flag to 1 causing an interrupt from device code 45.

The Enable/Disable Error Reporting instruction description that follows explains the commands.

NOTE: *The following describes all current SCP commands. Refer to your machine-specific supplement for the commands implemented on your system.*

Enable/Disable Error Reporting

DOBS *ac*,SCP



Function: SCP error reporting → enable/disable
Perform command
1 → Busy
0 → Done

Parameters: *ac* = command → unchanged

The Enable/Disable Error Reporting instruction sets the SCP Busy flag, clears the Done flag, and uses the contents of the specified accumulator to enable or disable CPU error reporting and to perform the command (function) contained in the command field.

Arguments

ac(16-31) Before execution, contains function word as follows:

Bits	Contents or Function
16	This bit enables or disables the SCP error reporting. 1 = enable; 0 = disable
17-23	Command. The SCP performs the function defined by these bits: **Command (octal) — Name** 000 — No-op 001 — Set time 002 — Select SCP powerdown mode 003 — Disable SCP powerdown mode 004 — Set block 005 — Enable all ERCC 006 — Mask ERCC page 007 — Mask soft ERCC 010 — Mask all sniff error reporting 011 — Disable all ERCC 012 — Size 013 — Set boot clock time 014 — Get boot clock time 015 — Set GMT offset 016 — Get GMT offset 017 — Reserved (used by Data General's AOS/VS) 020-176 — Undefined 177 — Enter diagnostic sequence
24-31	Interface Block. The CPU or SCP uses the contents of these bits dependent upon the command in bits 17-23. For any command requiring a CPU/SCP buffer, this value is a physical address pointer to a multiple word block in page zero (in the range of 0 to 377_8). For the Size command (012_8), this value is a type indicator.

The following explains the enable/disable SCP error reporting bit and the SCP commands.

The enable/disable bit (bit 16) enables or disables CPU error reporting. When the CPU or SCP reports an error, it uses the page zero address specified by the interface block (bits 24-31) as a pointer to a doubleword physical address. This address in turn points to a 16-word block that the CPU or SCP can use to report error data. The first word

of the block receives the error code. The remaining 15 words are available for reporting extended error status information. (The actual buffer lengths are specified by the functions that use them.)

If the SCP interrupts the CPU, the SCP disables error reporting until the CPU issues a new enable command.

For example, under a Data General operating system, the CPU uses the first word of the error block as the ERRLOG code number. Any error that requires extended error status also causes the entire 16-word block (including the code number) to be logged as the data area of the ERRLOG entry.

The allocated and implemented commands (bits 17-23) are defined as follows:

- No-op (command 000_8)

  The No-op command enables or disables SCP-to-CPU logging only; no other function is specified. The command enables ERCC error reporting, but does not change the current ERCC reporting mode.

- Set Time (command 001_8)

  The Set Time command sends the SCP the current time, and requires three words of the CPU-to-SCP buffer as follows:

Word	Contents
0	Number of days since December 31, 1967.
1, 2	Number of seconds since midnight.

  NOTE: *Word 0 of the buffer is set to 0 when this function is complete.*

- Select SCP Powerdown Mode (command 002_8)
- Disable SCP Powerdown Mode (command 003_8)
- Set Block (command 004_8)

  The Set Block command specifies to the SCP the address of the interface block (bits 24--31).

- Enable All ERCC Error Reporting (command 005_8)

  The Enable All ERCC Error Reporting command enables the SCP to detect and report any memory error. This function requires a CPU-to-SCP buffer of five words. The SCP detects the following ERCC errors:

  Single-bit 1-bit ERCC error during memory read.

  Multibit 2-bit (or more) ERCC error during memory read.

  Soft-sniff 1-bit ERCC error during memory refresh.

  Hard-sniff 2-bit (or more) ERCC error during memory refresh.

  NOTE: *After a system reset or power restoration, reporting of ERCC codes is turned off until an Enable All ERCC Error Reporting command is issued.*

- Mask ERCC Page (command 006_8)

  The Mask ERCC Page command turns off error reporting for a given page of physical memory.

- Mask Soft ERCC (command 007_8)

  The Mask Soft ERCC command disables the reporting of all one-bit correctable ERCC errors. Only two-bit correctable errors and all uncorrectable errors are reported.

- Mask All Sniff Error Reporting (command 010_8)

  The Mask All Sniff Error Reporting command disables the reporting of all soft-sniff and hard-sniff errors occurring during memory refresh.

- Disable All ERCC Error Reporting (command 011_8)

  The Disable All ERCC Error Reporting command tells the SCP to disable all memory error reporting and detection.

- Size (command 012_8)

  The Size command returns various system parameters, such as memory size, microcode revision, and node number. The Size command uses bits 24-31 of the accumulator to contain the value for the type of information being requested. The SCP requires that the Size command disable error reporting (bit 16 equals 0). Since the Size command uses word 0 of the SCP-to-CPU buffer to contain certain status data, the setting of word 0 to 0 has no meaning for this command. However, when the buffer is full, the SCP writes a status word with a value of 0 into its B register.

  The types currently specified in bits 24-31 are listed below and described in the following tables along with the contents of their CPU-to-SCP and SCP-to-CPU buffers.

Type #	Returns Information on
1	CPU
2	Memory
3	I/O device type (or reserved)

CPU Type (code 1 in bits 24-31 of command word)

Buffer	Address Offset (Octal)	Value		
CPU-to-SCP	+0	Node number -1 (myself) 0 (SCP)		
SCP-to-CPU	+0	137_8 (data identifier code)		
	+1	Error code		
	+2,3	Node number		
	+4,5	Status type word		
		Bit	Value	Description
		0	0	I/O processor
			1	Job processor
		1	0	Slave
			1	Master
		2-31	Undefined	Reserved
	+6,7	Model number *		
	+10,11	Microcode revision (major/minor) *		
	+12,13	Memory size *		
		* see NCLID instruction for a description		

MEMORY Type (code 2 in bits 24-31 of command word)

Buffer	Address Offset (Octal)	Value
CPU-to-SCP	+0,1	Physical page number which must be at the beginning of a module
SCP-to-CPU	+0	137_8 (data identifier code)
	+1	Error code
	+2,3	Number of pages
	+4,5	Code indicating memory type (Bit 0 of offset 4 indicates whether this memory is standard main memory or special memory.) **Standard Main Memory** **Bit Contents** 0 0 (indicates standard main memory) 1-31 Universal memory module code **Special Memory (Machine-specific) *** **Bit Contents** 0 1 (indicates special memory) 1-31 1 - Bit map graphics 2 - GIS memory $7FFFFFFF_{16}$ - Nonexistent
	+6,7	Status of memory area 0 = Believed to be good Nonzero = undefined

* The special memory for GIS and bitmap graphics will use additional words of the SCP-to-CPU buffer to return sizing information as follows:

GIS Memory

The format for additional GIS values starts at offset address $+10_8$ as follows:

Buffer	Address Offset (Octal)	Value
SCP-to-CPU	+10, 11	Model number
	+12, 13	Bits per pixel

Bitmap Graphics

The special memory for bitmap graphics will give the rest of the sizing information at the memory locations starting at 0 in the page where it was found. An example of what sizing information might look like follows.

Address (octal)	Contents	Function
0	t	2**t is the size of the block reserved
2	#	Model number (identifier for graphics subsystem)
4	p	Bits per pixel
6	pa	2**pa is offset to palette base
10	b	2**b is the offset to bitmap base
12	y	2**y is the number of pixels per line (y pitch)
14	z	2**z is the number of lines in the plane
16	n	Number of planes
20	np	Number of palette entries

I/O DEVICE Type (code 3 in bits 24-31 of command word)

Buffer	Address Offset (Octal)	Value
SCP to CPU	+0, 1	Number of device blocks
	+2, +3	Device block 1 +2 contains the device code +3 contains the device model number
	+4, +5	Device block 2
	...	Device code and device model number repeat in blocks of two words per device. Some possible device codes and model numbers are as follows:

Device Code (octal)	Model ID (octal)	Description
10	050000	TTI
11	050000	TTO (same as TTI)
14	060000	Real-time clock
21	060000	Printer
24	000000	Winchester disk
43	140000	Programmable interval timer
45	170105	SCP
63	110000	Magnetic tape
77	160000	CPU

- Set Boot Clock (command 013_8)

  The Set Boot Clock command disables error reporting and sends new boot clock values to the SCP. This command uses seven words of the CPU-to-SCP buffer as follows (words one through six must be binary coded decimal values):

Word Offset	Contents
0	Nonzero protocol word
1	Seconds
2	Minutes
3	Hours
4	Days
5	Months
6	Years

  Upon completion, the SCP sets word 0 of the SCP-to-CPU buffer to 0, returns a status value to its B register, and sets the Done flag. A status value of 2 indicates an acknowledgement; a value of 3 means the SCP could not perform this function (no further action is taken by the SCP). Note that if the boot clock is changed through the SCP-CLI, the SCP places the value 0 into its B register, enters code 206_8 into its error log, and posts an interrupt.

- Get Boot Clock (command 014_8)

  The Get Boot Clock command disables error reporting and requests the boot clock time from the SCP. The SCP writes the appropriate information into the SCP-to-CPU buffer as follows (words one through six are binary coded decimal values):

Word Offset	Contents
0	1 (sub-code indicating this buffer contains boot clock data)
1	Seconds
2	Minutes
3	Hours
4	Days
5	Months
6	Years

Upon completion, the SCP returns a status value to its B register, and sets the Done flag. A status value of 4 indicates that the requested information is now in the buffer; a value of 3 means the SCP could not perform this function (no further action is taken by the SCP).

- Set GMT Offset (command 015_8)

  The Set GMT Offset command disables error reporting and sends new Greenwich Mean Time values to the SCP. This command uses eight words of the CPU-to-SCP buffer as follows (words one through six must be binary coded decimal values):

Word Offset	Contents
0	Nonzero protocol word
1	Seconds
2	Minutes
3	Hours
4	Days
5	Months
6	Years
7	Offset sign (0 = positive, 1 = negative)

Upon completion, the SCP sets word 0 of the SCP-to-CPU buffer to 0, returns a status value to its B register, and sets the Done flag. A status value of 2 indicates an acknowledgement; a value of 3 means the SCP could not perform this function (no further action is taken by the SCP). Note that if the GMT offset is changed through the SCP-CLI, the SCP places the value 0 into its B register, enters code 207_8 into its error log, and posts an interrupt.

- Get GMT Offset (command 016_8)

  The Get GMT Offset command disables error reporting and requests the Greenwich Mean Time from the SCP. The SCP writes the appropriate information into the SCP-to-CPU buffer as follows (words one through six are binary coded decimal values):

Word Offset	Contents
0	2 (sub-code indicating the buffer contains GMT offset data)
1	Seconds
2	Minutes
3	Hours
4	Days
5	Months
6	Years
7	Offset sign (0 = positive, 1 = negative)

Upon completion, the SCP returns a status value to its B register, and sets the Done flag. A status value of 4 indicates that the requested information is now in the buffer; a value of 3 means the SCP could not perform this function (no further action is taken by the SCP).

- Enter Diagnostic Sequence (command 177_8)

  The Enter Diagnostic Sequence command disables CPU error reporting and all previously enabled functions. The SCP does not use the interface block address entered with this **DOBS SCP** instruction. Instead, the SCP uses the previous address as a pointer to the SCP/CPU interface block. The SCP clears its Busy flag. The SCP remains in diagnostic mode until either a console reset occurs or the CPU issues another **DOBS SCP** instruction.

  When the CPU issues the second **DOBS SCP** instruction, the SCP first places the contents of bits 16–31 of the specified accumulator into word 0 of the SCP-to-CPU buffer. The SCP then reads words 1–7 from the CPU-to-SCP buffer, inverts them, and writes them back to their respective locations in the SCP-to-CPU buffer. Upon completion, the SCP returns a status value of 0 to its B register, sets the Done flag, and interrupts the CPU.

  NOTES: *The following will also clear the SCP diagnostic mode: an* **IORST** *instruction, or the SCP-CLI commands,* RESET, START, *and* BOOT.

  The SCP-to-CPU interface block address is lost when diagnostic mode is terminated.

Registers, Flags and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
Busy flag	Set to 1
Carry	Unchanged
Done flag	Set to 0
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

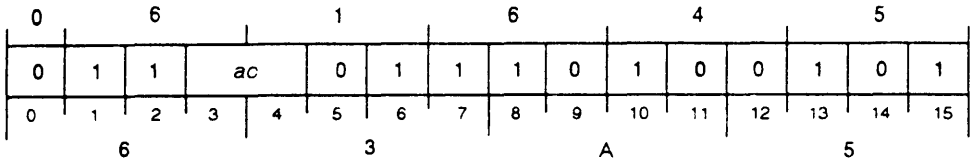
None

Exceptions

Issuing the **IORST** instruction causes single-bit errors to be checked and corrected, but not reported.

Return SCP Status

DIBC *ac*,SCP



Function: Status code → *ac*
0 → Busy
0 → Done

Parameters: *ac* = ? → status code

The Return SCP Status instruction clears the SCP Busy and Done flags and returns a code to the specified accumulator denoting the current status of the SCP. The CPU expects all information except status information to be passed using the SCP-to-CPU buffer. If this buffer contains information, the CPU does not expect this data to be valid until after it issues the **DIB** instruction.

Arguments

ac(16–31) After execution, contains codes denoting current status of SCP as follows:

Status (octal)	Meaning
000000	Log information is in current SCP-to-CPU buffer. SCP logging is disabled. The first word of the log buffer (word 0) is interpreted as a log code; the use of the remaining 15 words is dependent on the log code (extended status). The status codes and their definitions are: **Code (octal) Definition** 007 Powerfail detected 050 Power restore detected 053 Single-bit ERCC error detected (see ERCC extended status) 054 Multiple-bit ERCC error detected (see ERCC extended status) 055 Sniff or I/O detected multiple-bit ERCC error (see ERCC Extended Status) **ERCC Extended Status** The four-word extended status for ERCC codes 053, 054, and 055 is **Word — Contents** 0 — Status code (053, 054, or 055) 1 — Status **Bits — Definition** 0–11 — Reserved 12 — CPU access 13 — I/O access 14 — Reserved 15 — Sniff access 2 — Physical page number 3 — Doubleword on module 4 — Syndrome bits 140 SCP error logging enabled 141 SCP error logging disabled 142 Processor halted 143 SCP BOOT command has been executed 144 Power down 145 Power up 146 Reserved 147 Reserved 150 Battery backup complete 153 Microsequencer parity error 154 System cache parity error 155 System cache to Bank Controller parity error 156 IOC parity error 157 Sbus time-out 160 Sbus parity error

Status (octal)	Meaning
	Code Definition (octal) 161 Operating system error 162 SCP error log disk error 163 Infinite protection fault 164 Infinite page fault 165 Instruction cache enabled 166 Instruction cache disabled 167 Reserved 170 Reserved 171 SCP RESET command has been executed 172 Address Translation Unit enabled 173 Address Translation Unit disabled 174 Illegal PIO command 175 Reserved 176 Reserved 177 SCP HALT command has been executed 200 SCP CONTINUE command has been executed 201 SCP START command has been executed 202 SCP INIT command has been executed 203 SCP has disabled interrupts initiated by the Bank Controller for soft ERCC errors. (This occurs when there are multiple "stuck on one" or "stuck on zero" soft ERCC errors and the interrupt frequency is so high that it locks out the system console.) 204 Reserved 205 Hard interrupt from an unknown source 206 Boot clock time changed through SCP-CLI command. 207 GMT offset changed through SCP-CLI command.
000001	SCP reset. The SCP is reset and must be reinitialized with the DOBS *ac*,SCP instruction and a command 4. (All previously enabled functions have been disabled, and the SCP-to-CPU interface block address has been lost.)
000002	SCP function request acknowledge. This acknowledgment indicates to the CPU that the SCP has completed a requested function.
000003	SCP requested function is in error. The SCP reports an unknown error with this code. The SCP issues this code if the required SCP/CPU interface block has not been defined, an undefined function request is made, or invalid data is passed to the SCP (through the CPU-to-SCP buffer).
000004	Data requested is in buffer. The SCP has placed the requested information into the SCP-to-CPU buffer. The sub-code values returned to word 0 of the buffer indicate the contents of the buffer: **Sub-code** **Definition** 001 Boot clock data 002 GMT offset data
177777	SCP is in diagnostic sequence.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
Busy, Done flags	Set to 0
Carry	Unchanged
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

DIB *ac*,**SCP** The **DIB** *ac*,**SCP** and the **DIBC** *ac*,**SCP** instructions perform identical functions. The **DIBS** *ac*,**SCP** instruction is a no-op.

Exceptions

Issuing the I/O Reset instruction (**IORST**) causes single-bit errors to be checked and corrected, but not reported to the CPU.

Power Supply Controllers

Universal Power Supply Controller		Power Supply Controller	
Device Code	4_8	Device Code	4_8
Assembler Mnemonic	UPSC	Assembler Mnemonic	PSC
Priority Mask Bit	13	Priority Mask Bit	13

The power supply controllers for the ECLIPSE MV/Family systems contain an on-board microprocessor which perform functions such as: a power-up diagnostic self-test, monitoring the system power, and reporting failures, problems, and status information. These controllers are either the Power Supply Controller (PSC) or the Universal Power Supply Controller (UPSC) — refer to the machine-specific supplement for which type of controller your machine supports.

Both types of controllers provide powerup and powerdown sequencing, the transfer to battery operation, I/O operations with ECLIPSE MV/Family systems, and output voltage margining. A major difference between controllers is the communications pathway — the CPU communicates directly with the UPSC; the CPU-to-PSC communication generally is through an intermediate processor (usually the diagnostic remote processor), though this is transparent to your program.

Both controllers use the same device code (4_8) and the same priority mask bit (13). In multiple-I/O channel configurations, the CPU uses device code 4 on the primary I/O controller (IOC0) as the gateway to their power supply controller.

In addition, the controllers monitor the following:

- problems with the power supplies (such as overtemperature and overcurrent conditions);
- ac overvoltages and undervoltages (PSC only);
- overloads on the output voltages;
- state of the power switch;
- battery backup faults;
- fan or blower failures.

Device Flag Control

Device flag commands to the power supply controllers determine the enabling or disabling of controller interrupts.

f=omitted	Busy and Done flags unchanged.
f=**S**	Sets the Busy flag to 1 and the Done flag to 0.
f=**C**	Sets the Busy and Done flags to 0.
f=**P**	Sets the Busy flag to 1 and the Done flag to 0.

Power Supply Controller Instructions

Instructions to the PSC or UPSC differ in their functions; Table 8-18 lists the I/O instructions that affect the controllers.

Table 8-18 *I/O instructions for the power supply controllers*

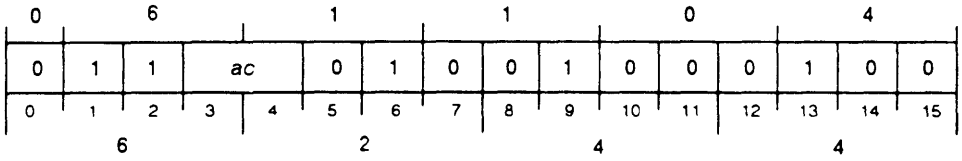
Assembler Statement	Function
DOAS *ac*,U/PSC	Writes data to UPSC or PSC.
DOAP *ac*,UPSC	Requests data from UPSC.
DIA*[f]* *ac*,U/PSC	Reads data from UPSC or PSC.
IORST	Clears Busy and Done flags and interrupt priority mask bit.
NIO*[f]* U/PSC	Manipulates Busy and Done flags.

The **IORST** instruction is described earlier in this chapter; the **NIO** instruction is explained in the *Instruction Dictionary*. Note that the I/O instructions — **DIB**, **DIC**, and **DOC** — to the controllers are no-ops.

In the instruction descriptions that follow, if the instruction pertains only to the Universal Power Supply Controller or only to the Power Supply Controller, we use their respecitive mnemonics (UPSC or PSC). If the instruction applies to both controllers, we use the mnemonic, U/PSC. When the arguments to the assembler differ, we present the UPSC code followed the PSC code. (When developing code, use the mnemonic appropriate for your controller — UPSC or PSC.) The forms of data written to or read from the controllers differ between type of controller.

Write Data to U/PSC

DOAS *ac*,U/PSC



Function: *ac* → U/PSC
1 → Busy
0 → Done

Parameters: *ac* = data → unchanged

The Write Data to U/PSC instruction sends the contents of the accumulator to the power supply controller. When you issue this instruction, the controller sets the Busy flag to 1 and the Done flag to 0.

Upon completion of the operation, the controller clears the Busy flag to 0 and sets the Done flag to 1.

Arguments

ac(16–31) Before execution, contains data to be sent to the power supply controller. Bits 16–23 are either undefined or contain all zeros dependent upon which controller register is being written to. Bits 24 and 25 specify which register on the controller will receive the data. Bits 26–31 contain data pertinent to the register chosen.

The following lists the 16-bit power supply registers (specified by bits 24 and 25) that can be written on the controller.

ac Bits 24, 25	Register Selected	Contents or Function
00	Control register	Selects reporting mode, power margining, and enable/disable battery backup.
01	Power margining register	Enables logic and memory voltages to be increased or decreased (when the back panel is jumpered for margining or margining is selected using the control register).
10	Reserved	Reserved for future use.
11	Diagnostic test register (UPSC only)	Verifies the data path between the computer and UPSC or enables the battery test. NOTE: *These bits are reserved for diagnostic purposes on the PSC.*

The following diagrams and tables describe the various controller registers and explain the functions when an accumulator bit is set.

Control Register (Register 0)

Undefined		0	0	CLR/FLT	BT	ALT	COM	BBU	PFM
16	23	24	25	26	27	28	29	30	31

ac Bit	Name	Contents or Function
16-23	Undefined	Unused.
24,25	00	Selects the control register.
26	CLR/FLT	If 1, clears fault code register. NOTE: *This bit applies only to the PSC. The UPSC reserves this bit for future use.*
27	BT	If 1, removes AC power to allow battery testing. NOTE: *This bit applies only to the UPSC. The PSC reserves this bit for diagnostic pusposes.*
28	ALT	If 1, masks out powerfail interrupts (depending on the system, a powerfail interrupt may be non-maskable). If this bit is 1, the CPU powerfail skip instructions (SKPDN and SKPDZ) will still identify the state of the powerfail flag.
29	COM	Enables or disables interrupts. If 1, the controller can interrupt CPU when a fault occurs. If 0, disables all I/O interrupts from the controller.
30	BBU	If 1, disables the battery backup unit.
31	PFM	If 1, enables power margining through program control.

Power Margining Register (Register 1)

A voltage is in the nominal state when the corresponding bit is 0. The voltage is margined when the corresponding bit is 1 and the computer is either jumpered or programmed for margining.

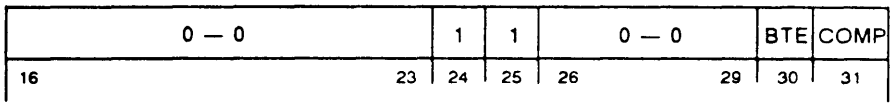
Undefined		0	1	+5LI	+5LD	ALLI	ALLD	+5MI	+5MD
16	23	24	25	26	27	28	29	30	31

ac Bits	Name	Contents or Function
16-23	Undefined	Unused.
24,25	01	Selects the power margining register.
26	+5LI	Increases +5 logic voltage.
27	+5LD	Decreases +5 logic voltage.
28	ALLI	UPSC — Increases +5MEM and -5 and +12 logic and memory voltages. PSC — Increases -5 logic and +12 logic voltages.
29	ALLD	UPSC — Decreases +5MEM and -5 and +12 logic and memory voltages. PSC — Decreases -5 logic and +12 logic voltages.
30	+5MI	UPSC and PSC — Increases +5 memory voltage.
31	+5MD	UPSC and PSC — Decreases +5 memory voltage.

Diagnostic Test Register (Register 3) — UPSC only

The UPSC performs the battery test or bit test specified by bits 30 and 31 of the accumulator.

To complete the command, the UPSC requires a second **DOAS** *ac*,**UPSC** instruction. When the UPSC fails to detect the second **DOAS** instruction, the UPSC automatically exits the diagnostic test. The UPSC indicates a time-out by setting the Done flag to 1 and placing the appropriate fault code in the fault code register. The fault code register may be read with the Read Data From UPSC instruction.



ac Bits	Name	Contents or Function
16-23	0 — 0	Reserved and must be zero.
24,25	11	Selects the diagnostic test register.
26-29	0 — 0	Reserved and must be zero.
30	BTE	Enables battery test. If 1, the battery test is enabled. Initiate the actual test with a second **DOAS UPSC** specifying the Control register (00)with bit 27 (BT) set to 1. Note that the BTE bit must be set before the BT bit.
31	COMP	Complements data. When a second **DOAS UPSC** instruction executes, the UPSC reads the data from its A buffer, and, if COMP is 1, complements the data. It then returns the data to the A buffer, which can be read with the **DIA UPSC** instruction.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
Busy flag	Set to 1
Carry	Unchanged
Done flag	Set to 0
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

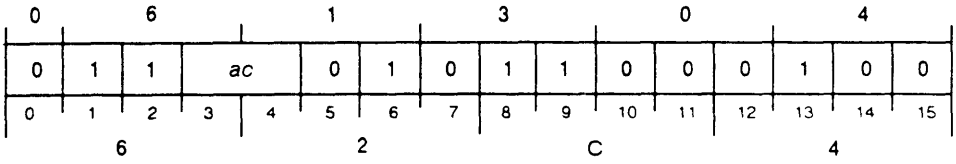
None

Exceptions

None

Request Data From U/PSC

DOAP *ac*,U/PSC



Function: Requests controller data

Parameters: *ac* = Data requested → unchanged

The Request Data From U/PSC instruction uses the specified accumulator to request specific information from the power supply controller.

Arguments

ac(28–31) Before execution, contains request for information from the controller. Bits 0 through 27 are reserved and must be set to 0. The contents of bits 28–31 are defined as follows:

Bits 28–31 (octal)	Controller Request	Function
00	Both	Read control bits.
01	UPSC PSC	Read battery backup and margining bits. Read margining bits.
02	Both	Read power supply system status.
03	Both	Read fault code register (return latest fault code/status data).
04	UPSC PSC	Read UPSC code revision number. Read PSC code revision number (major).
05	PSC	Read PSC code revision number (minor).
06	PSC	Read fault code register (return second latest fault code/status data).
07	PSC	Read fault code register (return third latest fault code/status data).
10	PSC	Return 252_8 (used for sizing).

After execution, contents unchanged.

Registers, Flags, and Stacks

AC0–AC3 Can be individually specified as *ac*; otherwise unused.

Busy flag Set to 1

Carry Unchanged

Done flag Set to 0

Overflow Unaffected

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

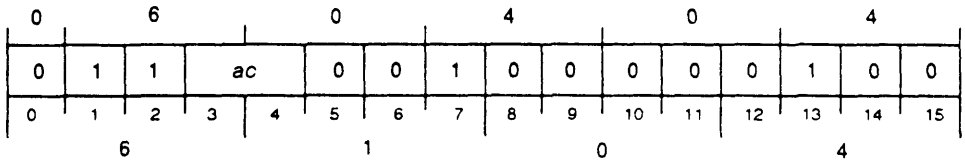
None

Exceptions

None

Read Data From U/PSC

IA[*f*] *ac*,U/PSC



Function: U/PSC (A buffer) → *ac*

Parameters: *ac* = ? → requested data

The Read Data From U/PSC instruction loads the data from the controller's A buffer into the accumulator. This instruction either returns the data requested (with the previous Request Data From U/PSC instruction) or reads the latest fault code following an interrupt from the power supply controller.

Arguments

ac(16-31) After execution, contains data from the controller's A buffer.

The following describes the data returned for each type of request.

Read Control Bits

0 — 0	JFM	ALT	COM	BBU	PFM
16 — 26	27	28	29	30	31

ac Bit	Name	Contents (if set to one)
16-26	0 — 0	Reserved and returned as zeros.
27	JFM	Power margining is enabled through hardware jumpering. (UPSC only) NOTE: *The PSC reserves this bit.*
28	ALT	Powerfail interrupt is masked out.
29	COM	The power supply controller can interrupt the CPU when a fault occurs.
30	BBU	The battery backup unit is disabled.
31	PFM	Power margining is enabled through program control.

Read Battery Backup and Margining Bits

0 — 0	BAT	0	+5LI	+5LD	ALLI	ALLD	+5MI	+5MD
16 — 23	24	25	26	27	28	29	30	31

ac Bit	Name	Contents (if set to one)
16-23	0 — 0	Reserved and returned as zeros.
24	BAT	The battery backup is connected and in use. (This bit is cleared if the system is not running on batteries, a battery fault occurs, or the BBU flag is set). NOTE: *This bit applies only to the UPSC; the PSC reserves this bit.*
25	0	Reserved and returned as zero.
26	+5LI	+5 logic voltage is increased.
27	+5LD	+5 logic voltage is decreased.
28	ALLI	UPSC — +5MEM and -5 and +12 logic and memory voltages are increased. PSC — -5 and +12 logic voltages are increased.
29	ALLD	UPSC — +5MEM and -5 and +12 logic and memory voltages are decreased. PSC — -5 and +12 logic voltages are decreased.
30	+5MI	UPSC and PSC — +5 memory voltage is increased.
31	+5MD	UPSC and PSC — +5 memory voltage is decreased.

Read Power Supply System Status

0 — 0		PART	FULL	RUN	CHAR
16	27	28	29	30	31

ac Bit	Name	Contents (if set to one)
16-27	0 — 0	Reserved and returned as zeros.
28	PART	The system is equipped with partial battery backup. NOTE: *This bit applies only to the UPSC; the PSC reserves this bit.*
29	FULL	The system is equipped with full battery backup.
30	RUN	The system is running on the batteries.
31	CHAR	The batteries are recharging.

Read Fault Code Register

0 — 0		Fault Code		Fault Category	
16	23	24	28	29	31

ac Bit	Name	Contents
16-23	0 — 0	Reserved and returned as zeros.
24-28	Fault Code	Specifies the fault code for a specific fault category.
29-31	Fault Category	Specifies the fault category (ranging from 0 through 7).

When the power system detects a fault, it loads the fault code and category into the fault code register and then displays the code on the front panel. The fault code register retains the code of the last fault, even if the fault clears. For example, if a fan takes too long to reach an effective speed, it can cause a fan fault. When the fan is running, however, the fault code register retains the fault, even though the fault clears. The "Fault Codes" appendix contains a categorized list of fault and status codes for the UPSC and PSC.

[f] Specify from **S**, **C**, and **P** for desired Busy and Done flag function.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*, otherwise unused.
Busy flag	Set by *f*
Carry	Unchanged
Done flag	Set by *f*
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

None

Exceptions

None

Multiple Central Processing Units

Some ECLIPSE MV/Family systems support more than one central processing unit per system. Each processor is identical in terms of architecture, features, options, and function, and every processor is independent of any other processor. These multiple-processor computers are tightly coupled systems; their processors share the same memory subsystem. The ECLIPSE MV/Family instruction set includes a subset of multiple-processor instructions for intra-processor communication.

Since each processor runs independent of the others, programming multiple processors is generally identical to programming a single-processor system. Differences include system initialization, a processor's view of memory, and intra-processor and I/O communications, which are described in the following sections.

NOTE: *Each processor in a multiple-CPU system should be running the same revision of microcode, otherwise results may be undefined.*

Initialization

The processors are equal except at boot time. One processor is designated the "initial processor," according to a state established at initial powerup. This state may be changed through the virtual console program running on the system (operator's) console.

The initial processor performs diagnostic routines and initialization functions on itself first. When the initial processor has passed all of its tests and has loaded its own control store, it may then communicate with another processor. Once running, each processor operates independently of the others, with actions determined by program control.

From a hardware point of view, the initial processor is the one which is booted first, has the default connection to the operator's console, and at powerup receives all interrupts. In all other hardware aspects, the processors are equal in function.

In the following discussion, references to the initial processor is for identification purposes only and is valid only when the processors are initialized. The following sequence starts at the system console:

1. The initial processor is "booted,"and issues an **LCS** instruction to load all of its control store.
2. The initial processor issues a Store State Pointer instruction (**SSPT**).
3. The initial processor may then determine the state of any other processor by issuing a Return Processor Status instruction (**JPSTATUS**).
4. If the initial processor decides to bring up another processor (target processor), the initial processor

   - Issues a Load Control Store into JP instruction (**JPLCS**) to the target processor. This procedure continues until the target processor has loaded all of its control store. (Note that issuing a **JPLCS** instruction is only necessary if the results of the **JPSTATUS** instruction indicate that the target processor has not had its control store loaded.) At this point, the target processor has the ability to respond to all other multiple-processor operations.
   - Places the starting program counter value and other data, such as accumulator values, into the appropriate sections of a processor state block.

- Issues a Start Another Processor instruction (**JPSTART**) to start the target processor. The **JPSTART** instruction also transfers the address of the JP state block to the target processor.

  The target processor then performs the following:

  - Loads its processor state from the JP state block.
  - Checks segment base register 0 (SBR0) for validity. If SBR0 is invalid (bit 0 is 0), the processor indicates a validity protection fault by placing error code 3 into AC1.
  - Changes the current ring of execution to the ring specified by the program counter in the JP state block.
  - Loads the wide stack registers from page zero of the new ring of execution.
  - Transfers execution to the instruction specified by the program counter in the JP state block.

5. The state (running/stopped) of any processor may be read at any time with the **JPSTATUS** instruction.

Processor State Block

Each processor maintains a processor state block (also known as the JP state block) in physical main memory. This block contains information which is stored inside the processor during program execution. The information in the block includes register and accumulator values and other processor-specific data. The state block must

- reside wholly within a single page,
- be doubleword aligned, and
- be resident.

The state block generally consists of 50 words in the following format:

Word Offset	Description
0-1	PSR
2-3	AC0
4-5	AC1
6-7	AC2
8-9	AC3
10-11	Carry (word 10, bit 0) and next program counter
12-15	FPSR
16-19	FPAC0
20-23	FPAC1
24-27	FPAC2
28-31	FPAC3
32-47	SBRs(0-7) — Each SBR uses a doubleword.
48-49	Physical **JPLOAD/JPFLUSH** address. Should be initially set to -1. (This value is written by the **JPSTOP** instruction and read by the **JPSTART** instruction and should be preserved over multiple starts and stops.)
50	Processor flags. Bits 0-13 are reserved and should be initialized to 0. Bit 14 is the state of the interrupt enable flag (ION); bit 15 is the state of the ATU ON flag.

Memory Views

Multiple processors may read or write the same, overlapping, or entirely different areas of memory. A single processor's view of memory is the result of a serial ordering of all processors' writes, though different processors may see different serial orderings. When a single processor writes to its view of memory, the changes eventually appear in all processors' views.

Certain ECLIPSE MV/Family instructions are either indivisible, uninterruptible, or serializable with respect to multiple processors. In the following discussion, "observer" refers to any entity which accesses memory, such as processors, I/O channel controllers (IOCs), and (indirectly) I/O devices.

- Indivisible — The operation is performed as one atomic unit; no intermediate results of the operation are visible to another observer. To another observer, the operation appears to have either never started or fully completed. The *Instruction Dictionary* describes those instructions which are indivisible.

  Certain store-type, memory-reference instructions are guaranteed to be indivisible if the target address is aligned to the data width, (such as, **STB** = 8 bits, **XWSTA** = 32 bits, **BTO** = 1 bit). Note that **WBTO** and related instructions are considered single-bit store instructions. Other instructions, which adhere to these alignment constraints, are also guaranteed to be indivisible.

  NOTE: *Atomic memory instructions (such as* **ISZ** *and* **DSZ***) executing in multiple-processor environments may produce an undesireable impact on system performance. Use an instructions such as* **XWADI**, *in place of* **XWISZ**, *to increment a pointer in memory.*

- Uninterruptible — I/O interrupts are not honored during execution of these instructions, unless an exception occurs that forces a partial completion of the instruction (such as a page fault).

- Serializable — Each operation goes to completion in all observers' views before the next operation begins. The result (memory views) of a set of operations is said to be serializable if there is a serial execution of those same operations that will produce the same result.

  The following instructions are serializable and indivisible:

  - **ISZ, EISZ, XNISZ, LNISZ, XWISZ, LWISZ, DSZ, EDSZ, XNDSZ, LNDSZ, XWDSZ,** and **LWDSZ**.

  - **SZBO, WSZBO,** and **WMESS**. (If interrupts are enabled, these instructions honor interrupts before starting execution. If these instructions skip, the interrupt is held off until the next instruction executes.)

  - **DEQUE, ENQH,** and **ENQT**. (These queue instructions are serializable and indivisible with respect to themselves. They are not serializable or indivisible with respect to instructions such as **XWLDA**. In this case, an **XWLDA** instruction will see a queue pointer in either the before- or after-execution state, as long as the pointer is even-aligned.)

The following demonstrates how different processors have different memory views.

```
     \  Processor #        A                        B
Time \
  0                 STA   0, 0, 0            STA   0, 1, 0
  1                 XWLDA 0, 0, 0            XWLDA 0, 0, 0
```

Initially, AC0(processor A) contains X', AC0(processor B) contains Y', and doubleword 0 contains XY.

At Time 1, AC0(processor A) will get either X'Y or X'Y' and AC0(processor B) will get either XY' or X'Y'.

There is no serial execution of the two store instructions which gives either X'Y or XY'. If these instructions were serializable, either processor's AC0 would get X'Y'.

I/O Communication

Multiple-processor ECLIPSE MV/Family systems support execution of I/O instructions, such as **CIO** or **PIO**, from any of the processors residing on the system bus. In a multiple-IOC configuration, all processors can do I/O to all devices on all IOCs. I/O instructions that are broadcast to all I/O channels, such as **MSKO**, are executed by all IOCs in the system simultaneously.

Multiple I/O Channels

Some multiple-processor ECLIPSE MV/Family systems may support more than one I/O channel (IOC). Each I/O channel supports its own data channel (DCH) and Burst Multiplexor Channel (BMC). In these configurations, IOC0 supports the following integral devices:

Mnemonic	Device Code (octal)	Description
TTI	010	TTY input
TTO	011	TTY output
PSC	004	Power supply
SCP	045	System control processor/program (or diagnostic remote processor)
CPU	077	Central processing unit
RTC	014	Real-time clock
PIT	043	Programmable interval timer

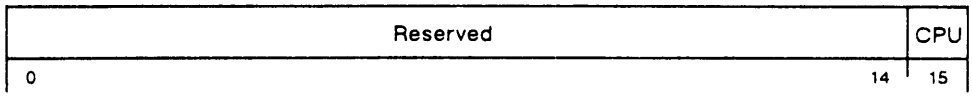
The device codes for these devices are reserved on all IOCs and may not be used for any other purpose.

I/O Interrupt Handling

A multiple-processor system handles I/O interrupts using a dedicated device mode — each IOC dedicates itself to a particular processor. All interrupts generated by an IOC are directed to a single processor; a particular processor can receive interrupts only from the IOC that is attached to it. In multiple-IOC configurations, the IOCs may be split between the processors or all IOCs may interrupt one processor, leaving the other processors to run uninterrupted.

Each IOC contains a 16-bit register (7702_8), that controls which CPU is to receive I/O interrupts. Systems which support only a single processor ignore the contents of this register. This read/write register is available only while a multiple-processor system is in dedicated mode; an attempt to read or write this register while the system is in any mode other than dedicated may cause unpredictable results.

The format of the CPU dedication control register is diagrammed below and described in the following table.



Bits	Name	Contents or Function
0-14	Reserved	Reserved for future use; must be set to 0.
15	CPU	Processor number to which all NOVA type interrupts (except cross interrupts) will be directed. On a system reset, this number is set to the value of the initial processor. Upon execution of an IORST instruction, this number is set to the value of the processor that issued the IORST instruction.

NOTE: *The actual number of "Reserved" and "CPU" bits implemented is dependent on the number of processors in the system.*

Intra-Processor Communication

The processors in multiple-processor configurations communicate using a set of privileged multiple-processor instructions. Communications between these tightly coupled processors occurs in shared and/or reserved locations in main memory. Each processor in the system is identified by a unique Processor ID number. The Processor ID number is a 16-bit unsigned integer assigned by the hardware. Acceptable values range from 0 to the maximum number of processors minus 1.

A multiple-processor instruction executing on one processor affects one or more processors. Table 8-19 lists the multiple-processor instructions; the complete instruction descriptions are given in the *Instruction Dictionary*. Some standard ECLIPSE MV/Family non-multiple-processor instructions affect multiple-processor systems to varying degrees; Table 8-20 lists these instructions. All other ECLIPSE MV/Family instructions (including I/O to the processor) executing on one processor affect only that processor.

NOTE: *All multiple-processor instructions are privileged instructions.*

Table 8-19 *Multiple-processor instructions*

Mnemonic	Description
CINTR	Request, remove, or query a request for a cross interrupt.
JPFLUSH	Fill in the JP state block for this processor.
JPID	Return a Processor ID number.
JPFLOAD	Load a subset of a JP state block into this processor.
JPLCS	Load control store into a target processor.
JPLOAD	Load a JP state block into this processor.
JPSTART	Request a processor to continue from the stopped state.
JPSTATUS	Return information on a processor's status.
JPSTOP	Request a processor to stop.

Table 8-20 *ECLIPSE MV/Family instructions with multiple-processor functions*

Mnemonic	Description
IORST	I/O Reset. In addition to its normal functions, this instruction also clears all pending cross interrupts, and redirects all IOC traffic to the processor that issued the IORST instruction.
HALT	Halt. Halts this processor and notifies the SCP that it is stopped.
SSPT	Store State Pointer. Regardless of the number of processors, there is only one state pointer per system. It is recommended that the state area not be moved, as the system maintains some parameters and synchronization information within the state area. If the state area is moved, certain operations will be disrupted and data may be lost.
PRTSEL	I/O Channel Select. Changes the default I/O channel on ALL processors.

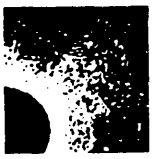
Error Codes

Conditions which produce errors during execution of multiple-processor instructions return a value to AC1. Table 8-21 lists these errors.

Table 8-21 *Error values returned to AC1*

Value	Instruction	Description
1	JPLCS, JPSTART	Not stopped. The processor to receive the request is currently running.
2	CINTR, JPLCS, JPSTART JPSTATUS, JPSTOP	Nonexistent processor. The processor ID specifies a value for a processor which does not exist.
3	JPLCS	LCS error.
4	CINTR, JPLCS, JPSTART, JPSTATUS, JPSTOP	Processor failure. The processor to receive the request is not working.
5	JPFLUSH	No JP state block.
6	CINTR	Illegal option.
7	JPSTOP	Processor not running.

End of Chapter



9

Memory and System Management

The processor supports memory management and system management facilities for an operating system. This chapter presents basic information to assist in writing operating system software.

The memory management facilities transform a logical address into a physical address and monitor the contents of the physical memory. The system management facilities return or modify implementation-dependent information about the system and the service faults.

The processor supports a virtual memory size of 4 Gbytes, which it distributes through eight segments. Each segment can support up to 512 Mbytes of logical address space. As the logical address space is larger than the physical address space, the processor uses a demand-paging scheme.

This chapter explains memory management functions (segment access and address translation), and system functions (processor identification and fault handling of privileged violations).

Page Access

Pages of logical memory are maintained on disk until the processor needs them in physical memory. (A page equals 2 Kbytes.) When referring to an instruction or to data that currently resides on disk, the processor moves the page to physical memory. When physical memory is full, however, the processor may first copy a page from memory to disk before moving the referenced page into memory. To facilitate the operation, the processor maintains tables in memory that determine

- Where a page resides (memory- or disk-resident).

  Bits 11-31 of a segment base register specify a physical address of a pagetable in memory. Each segment is described by a pagetable, which occupies at least 2 Kbytes and begins on an integral 2-Kbyte boundary. A pagetable contains entries that indicate where the pages reside in memory.

- When to overwrite a page in memory with a page from disk.

  The processor maintains a table of referenced and modified bits.

Segment Access and Address Translation

To access a memory word or words, the processor translates a logical address (indirect or effective address) to a physical address, and accesses the physical page, which contains the word or words.

The following paragraphs describe the segment base registers, pagetables, and the logical-address-to-physical-address translation.

Segment Base Registers

To access a segment, the processor first checks the segment base register specified by the logical address. Bit 0 of the segment base register controls access to the segment by specifying if the processor can refer to the segment to execute the instruction. If the processor cannot refer to the segment, the processor aborts the instruction and services a segment validity protection fault. Refer to the section, "Protection Violations," in this chapter for further information on protection fault handling.

The processor maintains one segment base register for each of the eight segments. The segment base registers (SBR0 through SBR7) contain information which

- validates segment access.
- validates I/O access.
- specifies a one- or two-level pagetable.
- specifies for its segment the address of the first entry in the pagetable.

The segment base registers can be modified with the privileged instructions, **LSBRA** and **LSBRS**, which loads a block of doublewords from memory into the segment base registers.

NOTE: *Privileged instructions must execute in segment 0; otherwise, a protection violation occurs.*

Figure 9-1 shows the format of a segment base register; Table 9-1 describes the format.

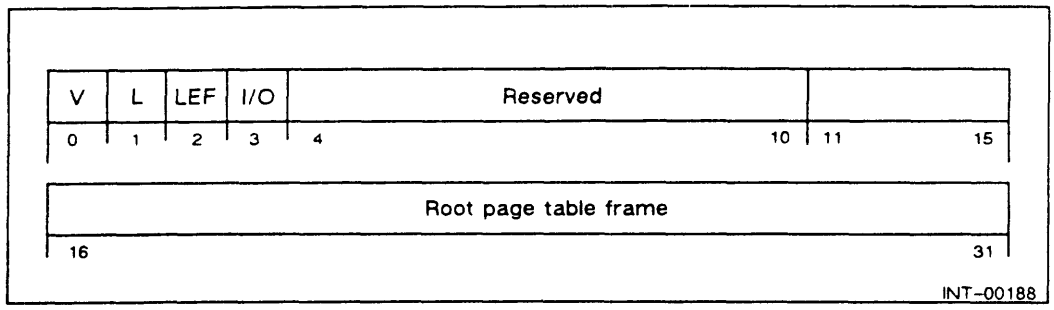


Figure 9-1 *Segment base register format*

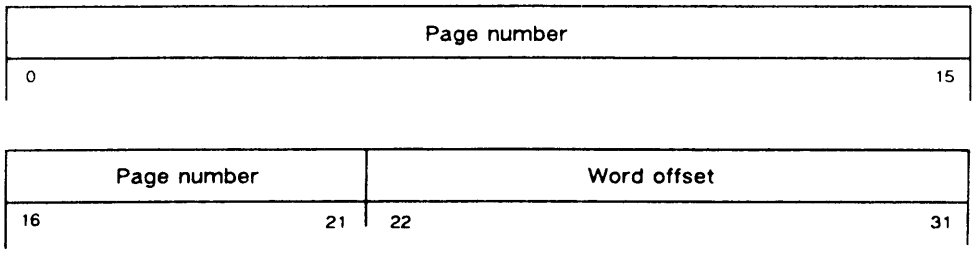
Table 9-1 *Segment base register format description*

Bit	Name	Contents or Function
0	V	Segment-validity flag. The processor accesses a segment either to execute an instruction or to access data for an instruction that reads or writes data. The segment, however, must be a valid reference. If 0, this is an invalid segment. The processor aborts a memory reference instruction and services a protection violation when the logical address refers to an invalid segment. If 1, this is a valid segment. Following a valid segment check, the processor checks for a valid addressing range (translation level) in the logical address.
1	L	Translation-level flag. The processor can access the segment with either a one- or two-level pagetable. If 0, this is a one-level pagetable. The processor can use a one-level pagetable with a program that requires 1 Mbyte or less of logical address space in the segment. A one-level pagetable entry contains the pagetable offset for the physical address translation. If 1, this is a two-level pagetable. The processor must use a two-level pagetable with a program that requires from 1 Mbyte to 512 Mbytes of logical address space in the segment. A two-level pagetable entry contains the address of the second pagetable, which contains the pagetable offset for the physical address translation. Refer to the section, "Pagetable," for additional information on the pagetable. Refer to the section, "Address Translation," for an example of using a segment base register and one or two pagetables.
2	LEF	Mode flag. This flag controls the interpretation of the I/O or LEF opcode(s) (opcodes that begin with 011_2). If 1, the processor executes the instruction as an LEF instruction. If 0, the processor executes the instruction as an I/O instruction. Before executing the instruction as an I/O instruction, the processor checks the I/O validity flag.
3	I/O	I/O validity flag. The processor checks the I/O validity flag when executing an I/O instruction. If 0, I/O operations are illegal from this segment. The processor aborts executing the I/O instruction and services the protection violation. If 1, I/O operations are legal from this segment. The processor executes the I/O instruction.
4-10	Reserved	Reserved for internal Data General use.
11-31	Root pagetable frame	Specifies the most significant bits of the physical address for the root pagetable page. (The table begins on a 2-Kbyte address boundary.) The remaining bits of the address come from either bits 4-12 or 13-21 of the logical address.

Page Frames

A page frame address (or page number) is a page address shifted right 10 bits.

A physical address has the following format.



A page address is considered to be the page number with 10 zeros following it. This page number is also the page frame, because the page actually includes the addresses between the start of the page (word offset 0) and the end of the page (word offset $3FF_{16}$). For example:

Page 1 is:	$1\ 0000000000_2$ =	400_{16}
Page 55 is:	$0101\ 0101\ 0000000000_2$ =	15400_{16}

Page frame 1 corresponds to the page address 400_{16} and refers to the data words with addresses 400_{16} through $7FF_{16}$.

Pagetables

In each segment, the processor accesses a pagetable that specifies the status of the pages for the segment in memory. The pagetable manipulation instructions are Load Pagetable Entry (**LPTE**) and Store Pagetable Entry (**SPTE**). The pagetable contains an entry (PTE) for each page which

- indicates if a page is valid and the type of access.
- indicates if a page is currently in physical memory.
- contains information needed to translate a logical address to a physical address.

Figure 9-2 shows the format of a pagetable entry. Table 9-2 describes the format.

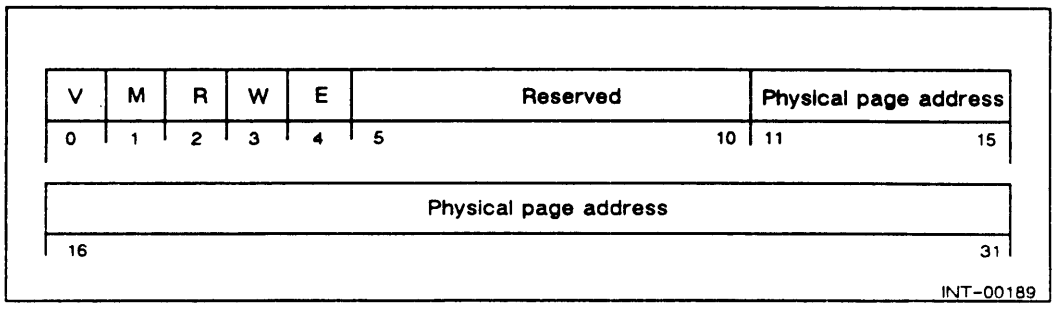


Figure 9-2 *Pagetable entry format*

Table 9-2 *Pagetable entry format description*

Bit	Name	Contents or Function
0	V	Valid-access flag. The processor accesses the page to read or write data, or to execute an instruction. The page reference must be a valid page. If 0, this is an invalid page. The processor aborts the memory reference instruction and services the protection violation when the logical address refers to an invalid page. If 1, this is a valid page. Following the valid page check, the processor checks for a valid page access (read, write, or execute).
1	M	Memory-resident flag. For the processor to access a page for reading or writing data, or for executing an instruction, the page must reside in physical memory. If 0, the page is on disk. The processor suspends the memory reference instruction and signals a page fault when the logical address refers to a disk-resident page. Following the page fault, the processor resumes executing the memory reference instruction. If 1, the page is in physical memory. The processor completes executing the memory reference instruction when the logical address refers to a memory-resident page.
2	R	Read-access flag. The processor must access the page for reading data. If 0, the processor cannot access the page for reading. The processor aborts the memory reference instruction and services the protection violation when the instruction requests a read operation, such as loading an accumulator or skipping on the condition of a memory word. If 1, the processor can access the page for reading. Following the valid read access, the processor checks for a disk- or memory-resident page status. NOTE: *A page with write or execute access also requires read access, otherwise, results are indeterminate.*
3	W	Write-access flag. The processor must access the page for writing data. If 0, the processor cannot access the page for writing. The processor aborts the memory reference instruction and services the protection violation when the instruction requests a write operation, such as storing an accumulator or modifying a bit of a memory word. If 1, the processor can access the page for writing. Following the valid write access, the processor checks for a disk- or memory-resident page status.
4	E	Execute-access flag. The processor must access the page for execution of its contents. If 0, the processor cannot access the page for execution. When the next instruction to be executed is from a page whose execute-access flag is zero, the processor aborts the instruction and services the protection violation. If 1, the processor can access the page for execution. Following the valid execute access, the processor checks for a disk- or memory-resident page status.
5-10	Reserved	Reserved for use by Data General software.
11-31	Physical page address	Identifies a page in memory. For a one-level pagetable translation, the physical page address refers to a page containing an instruction and/or data. For a two-level pagetable translation, the physical page address refers to a page containing the base of another pagetable. (Note that the processor ignores the page access bits -- bits 2, 3, and 4 — for a two-level pagetable translation.)

Address Translation

Following a valid segment reference, the processor checks the range of the logical address space within the segment, and compares it to the address range of the logical address. Bit 1 of the segment base register defines a one- or two-level pagetable which specifies the addressing range. Refer to the section, "Segment Base Register," for further details.

The processor compares bit 1 of the segment base register with bits 4–12 of the logical address. When bit 1 equals 0, the logical address bits 4–12 must be all zeros. The processor aborts the instruction and services the protection fault when any of the logical address bits 4–12 contain a 1.

Figure 9–3 illustrates an indirect or an effective logical address for a one- and two-level pagetable; Table 9–3 describes the formats. Refer to the chapter, "System Overview," for an explanation of calculating an indirect or effective logical address.

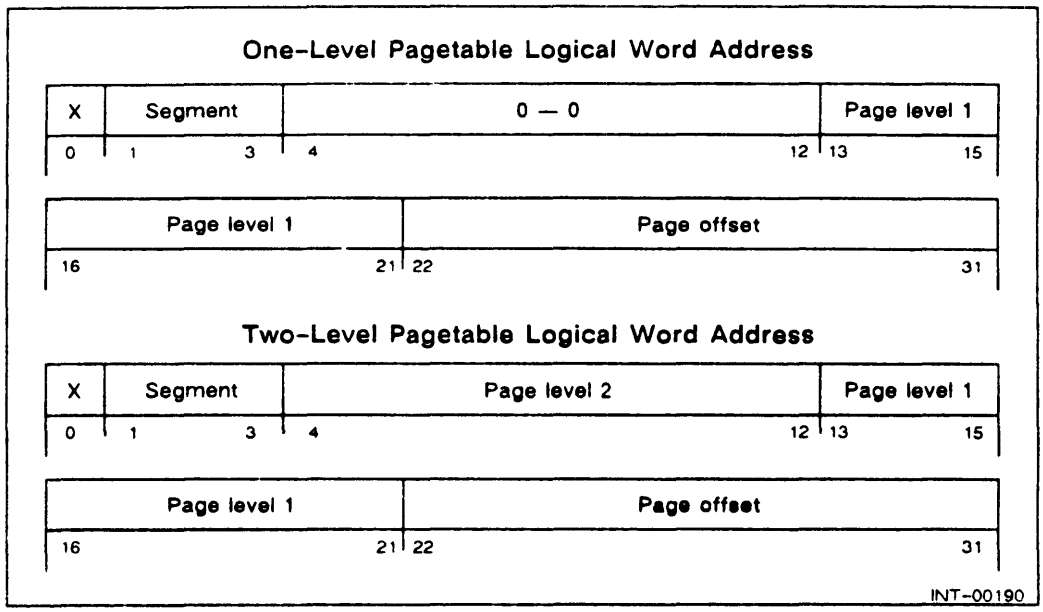


Figure 9–3 *Indirect and effective logical address formats*

Table 9–3 *Logical address format description*

Bit	Name	Contents or Function
0	X	The processor ignores this bit when using direct addressing. The processor tests bit 0 when using indirect addressing and continues to test it in subsequent indirect addressing until the bit equals zero.
1-3	Segment	These bits specify one of the eight segment base registers.
4-12	Page level 2	These bits specify an entry in the first of two pagetables for a two-level pagetable translation. The pagetable entry contains the address of the second pagetable. For a one-level pagetable translation, the page level 2 field must be all zeros. If they are not, the processor aborts the instruction and services a pagetable validity protection fault. Refer to the section, "Protection Violations," in this chapter for further information on protection fault handling.
13-21	Page level 1	These bits specify an entry in a pagetable. For both one- and two-level pagetable translation, the pagetable entry contains the address of the final page to be accessed for data or by an instruction.
22-31	Page offset	These bits specify the final entry in the final page. The page offset completes the address translation

Figures 9-4 and 9-5 present examples of one- and two-level pagetable translation, respectively. The circled numbers labeling the accompanying paragraphs correspond to the circled numbers shown in the figures.

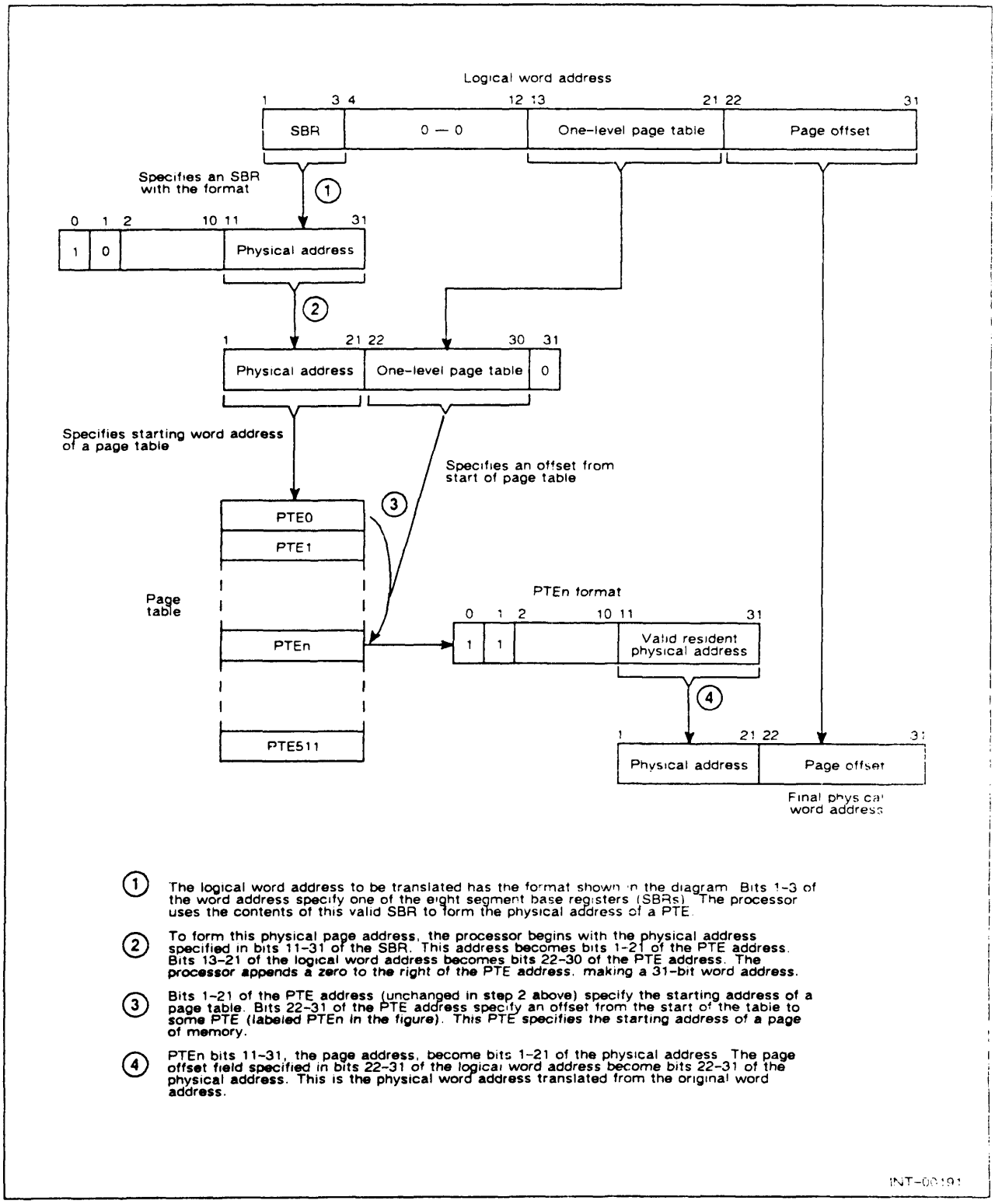


Figure 9-4 *One-level pagetable translation*

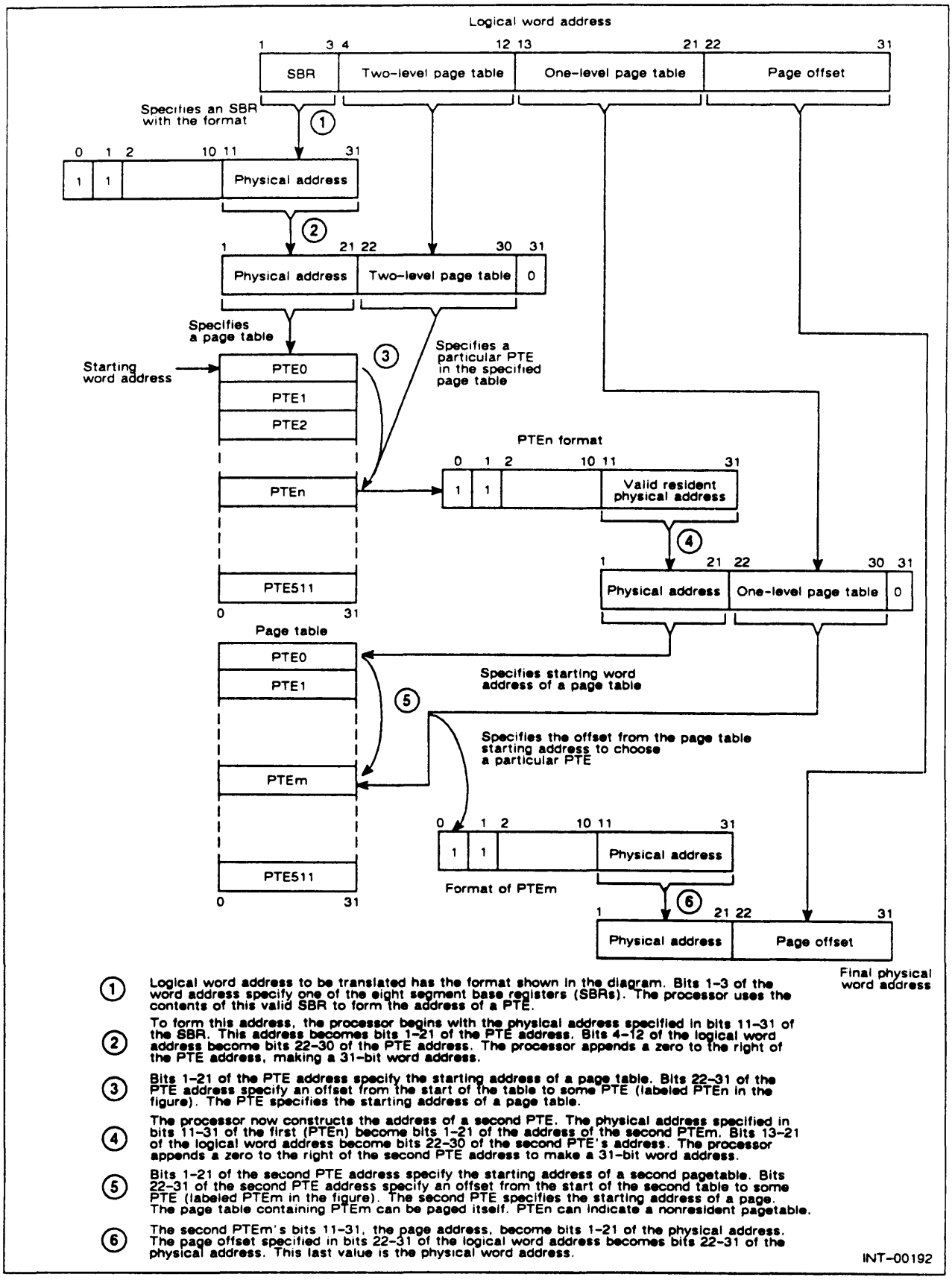


Figure 9-5 *Two-level pagetable translation*

Page Access

When an instruction refers to a page, the processor determines the validity of the access by checking the access request with the appropriate validation and access validation bits in the pagetable entry.

If an instruction refers to a valid page that is not currently in physical memory, a page fault occurs. The fault handler saves the current state of the processor in reserved memory (as a context block), moves a memory page to disk (if required), and then transfers the referenced page from disk to memory.

Access Validation

When a referenced page is valid, the processor determines whether the page is restricted to a particular access. Bits 2–4 of the referenced pagetable entry contain the access bits that specify any restriction.

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

NOTE: *In general, read access must always be available to any page with execute or write access.*

When the reference to memory is for writing, the processor checks bit 3. A 1 in bit 3 indicates a valid write, while a 0 indicates an invalid write. When the reference is invalid, a protection fault occurs, and AC1 contains the error code 1.

If the reference to memory is for executing, the processor checks bit 4. A 1 in bit 4 indicates a valid execute while a 0 indicates an invalid execute. When the reference is invalid, a protection fault occurs, and AC1 contains the error code 2.

Demand Paging

As logical address space is larger than the physical memory space, all pages cannot reside in physical memory at the same time. A paging facility (under control of the page fault handler) moves referenced pages in and out of memory whenever necessary. This process is called *demand paging*.

When an instruction refers to a valid page not currently in physical memory, a page fault occurs. A status field in the context block indicates the cause of the page fault (Refer to the section, "Page Faults," in this chapter for detailed information.). If an instruction refers to a location that requires a two-level pagetable when only a one-level pagetable is allocated, then a protection violation occurs. Refer to the section, "Protection Violations," in this chapter for more information.

Referenced and Modified Bits

A referenced bit and a modified bit, associated with each physical page in memory, indicate whether a page has been read from or written to. When the processor reads a word from memory, it sets the referenced bit associated with the physical page to one. When the processor writes a word to memory, the processor sets the referenced and modified bits associated with the physical page to ones. A read or write operation occurs when the processor accesses memory without a protection fault occurring on a memory resident page.

The referenced bit helps to determine which pa; physical memory the page fault handler should replace with a new page from di ne referenced bit allows an operating system and the page fault handler to determine the frequency of references to individual pages.

The modified bit indicates if the processor wrote to a memory page. When a modified bit equals one, the processor modified the contents of the page. The page fault handler must first copy the page to disk before moving a new page from disk to memory. If a modified bit equals zero, the processor did not modify the contents of the page, and the page fault handler can immediately move a new page from disk to memory.

NOTES: *A memory reference by the computer's I/O subsystem does not affect the state of the modified and referenced bits.*

When the processor accesses pagetable pages to perform logical-to-physical address translations, the referenced bit for the pagetable pages may or may not be set.

Table 9-4 lists the privileged instructions that manipulate the referenced and modified bits. Refer to the chapter, "Fixed-Point Computing," for a list of additional instructions that manipulate bit strings.

Table 9-4 *Instructions that manipulate referenced and modified bits*

Instruction	Operation
LMRF	Loads the modified and referenced bits for a page frame.
SMRF	Stores the modified and referenced bits for a page frame.

NOTE: *The results of these instructions are undefined for page frames outside main memory.*

Central Processor Identification

Central processor identification (CPUID) instructions load information about certain system parameters (such as the memory size and the microcode revision level) into one or more fixed-point accumulators. Table 9-5 lists the central processor identification instructions.

Table 9-5 *System identification instructions*

Instruction	Operation
ECLID, LCPID	Loads CPUID information into bits 0-31 of AC0.
NCLID	Loads CPUID information into bits 16-31 of AC0, AC1, and AC2.

Privileged Faults

While executing an instruction, the processor checks the operation and the data being operated on. If the processor detects an error, a privileged or nonprivileged fault occurs before the processor executes the next instruction in the instruction stream. A nonprivileged fault is serviced by the operating system within the segment where the instruction is executing. (The chapter, "Program Flow," discusses the handling of nonprivileged faults.)

Privileged faults are serviced by the operating system in segment 0. Upon detection of a privileged fault, the address translator generates either a page or protection fault.

- Page fault — The processor detects a page fault when the interpretation of the validity and appropriate access bits in a pagetable entry is coupled with an attempt to refer to a location that is part of the logical address space, but is not part of the physical address space.
- Protection fault — The processor detects a protection fault for an invalid memory reference, invalid I/O operation, or illegal instruction (such as a privileged instruction or an unimplemented opcode).

The following sections describe the actions the processor takes when a page fault or protection fault occur.

Page Faults

When a page fault occurs, the processor does the following (refer to Figure 9-6):

NOTE: *The following must always be resident in physical memory: page zero of the current segment of execution, page zero of segment 0, the context block, and the page fault handler.*

1. If the current segment is not 0, stores the frame pointer and stack pointer in their respective locations in page zero of the current segment and performs a segment crossing to segment 0.
2. Uses the contents of locations 32_8 and 33_8 of segment 0 as a base address to store a context block (the internal state of the machine) in memory. Refer to the appendix, "Context Block Formats," in the machine-specific supplement for the contents of the context block.
3. Initializes the segment 0 stack using the wide stack parameters from page zero of segment 0.
4. Stores one of the following codes into AC1:

Code	Explanation
0	Reserved
1	Reserved
2	Pagetable page fault
3	Reserved
4	Normal object reference

5. Disables interrupts for one instruction, and jumps indirect through locations 30_8 and 31_8 of segment 0.

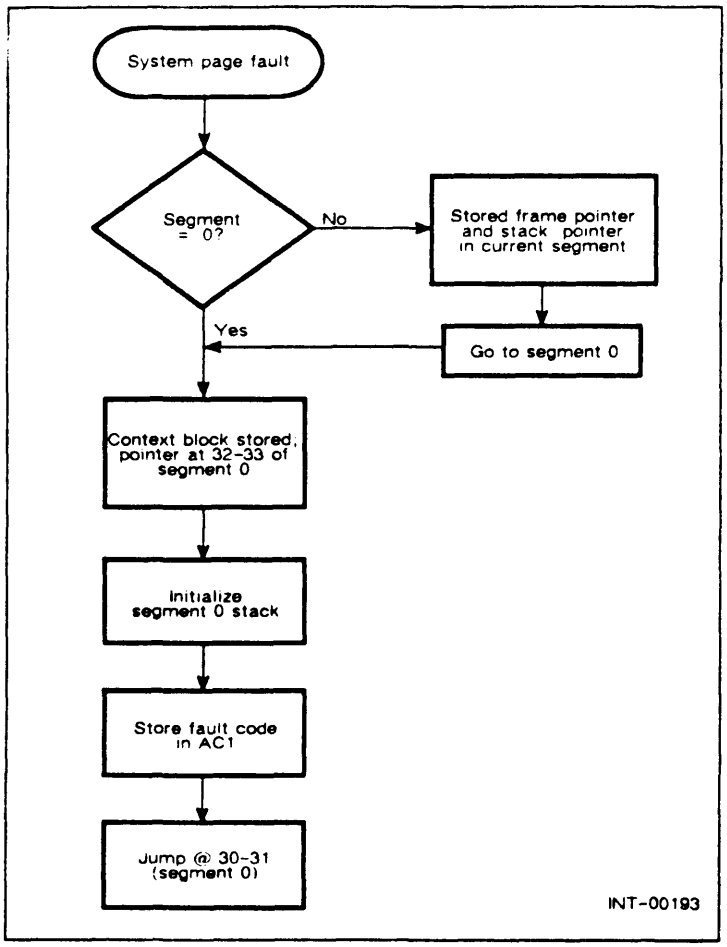


Figure 9-6 *Page fault sequence*

6. Executes the first instruction of the page fault handler.

   The page fault handler then,

   a. Begins restoring a page from memory to disk, if necessary. Refer to the section, "Referenced and Modified Bits," for more information on determining when a page needs to be restored to disk.

      The page fault handler invokes the I/O interrupt system to transfer the page to disk.

   b. Initiates loading the referenced page from disk to memory after the page fault handler restores the referenced page to disk.

      The page fault handler invokes the interrupt system to transfer the page from disk.

   c. Restores the state of the processor after the page fault handler loads the referenced page into memory.

      The page fault handler executes the **WDPOP** instruction, which restores the state of the processor and restarts the interrupted program. The **WDPOP** instruction accesses the information in the context block to restore the processor's state.

7. Completes the memory reference and continues executing the instruction.

NOTE: *A page fault must not occur during steps 1 through 5; otherwise, the processor halts.*

Protection Violations

The processor detects a protection violation for an invalid memory reference, invalid I/O operation, or illegal instruction (such as a privileged instruction or an unimplemented opcode).

Since an operation could produce multiple protection violations, the processor imposes priorities on the faults. When two or more faults occur simultaneously, the processor services the highest priority fault and ignores lower priority faults. Table 9-6 lists the protection violation faults in the order of priority. For instance, if writing to a write-protected page in an inner ring, the processor services the inward ring reference protection violation (with priority 2) and ignores the write protection violation (with priority 4).

Table 9-6 *Priority of protection violation faults*

Level of Priority	Fault Description
0	Privileged or I/O instruction violation
1	Indirect addressing violation
2	Inward reference violation
3	Segment validity violation
4	Pagetable validity violation
5	Read, write, or execute access violation
6	Segment crossing violation
7	Unimplemented opcode or instruction

Instructions which operate on characters (such as **WCMV** or **WCMT**) must use byte pointers that denote ring-valid addresses in the user's address space (even if the length of the string to be moved, or copied, etc., is zero). Any and all instances described by byte-pointer (or length) pairs must also be entirely ring-valid in the user's address space. Should these conditions not be met, a protection fault may occur, even if the offending address is not actually required to be accessed. This could occur in the case of zero-length strings or if the bytes are ignored due to truncating a long source string. The protection fault may occur either prior to any bytes being accessed, or at the time of the offending access.

Protection Violation Sequence

When the processor detects a protection violation (Figure 9-7), the processor aborts execution of the instruction which caused the protection fault and performs the following:

NOTE: *Page zero for segment 0 and page zero for the segment where the protection fault originated must both be resident in physical memory, otherwise, an "infinite page fault" results.*

1. Saves the offending program counter value (offending PC), the fault code describing the fault type, and the offending address (if any) in internal processor state.
2. If the offending PC is in segment 0, procedes to step number 6.
3. Stores the contents of the wide stack pointer and the wide frame pointer into the page zero locations of the current segment (locations 20_8 and 22_8, respectively). If another protection fault is detected during this operation, the stack values are not stored into their locations.
4. Crosses to segment 0.

5. Redefines the wide stack for segment 0. The processor initializes the wide stack pointer, wide stack limit, and wide stack base registers using the contents of locations 20_8 through 27_8 in page zero of segment 0 . (See the note following this discussion.)

6. Pushes a fault return block, as shown in Table 9-7, onto the segment 0 stack. (See the note following this discussion.)

7. Sets the PSR to zero.

8. Initializes AC0, AC1, and AC2.

   Sets AC0 equal to the address of the instruction (offending PC) causing the fault.

   Sets AC1 equal to a value identifying the fault. Table 9-8 lists the protection fault codes.

   Sets AC2 equal to the specific address (offending address) that caused the reference problem, if applicable (bit 0 is undefined). Table 9-8 lists the faults that return an address to AC2.

9. Checks for stack overflow.

   If stack overflow occurs, the processor pushes a stack fault return block onto the stack and process the stack fault. The stack fault return block contains the address of the protection fault handler. (See the note following this discussion.)

   If no stack overflow occurs, the processor continues to service the protection fault.

10. Jumps to the fault handler and executes the first instruction before any I/O interrupts are taken. Reserved memory location 36_8 of segment 0 contains the 16-bit starting address of the protection violation fault handler. (See the note following this discussion.)

NOTE: *If another protection fault(s) occurs before processing of the original protection fault is resolved, the processor*

- *Sets the PSR to 0.*
- *Places the offending PC into AC0, the fault code into AC1, and the offending address (if any) into AC2.*
- *Halts with a status of "infinite protection fault."*

If one or more protection faults should occur (during steps 5, 6, 9, or 10), the values in AC1 and AC2 are those for the last protection fault detected. The offending PC in AC0 is always that of the last instruction which started executing.

During the servicing of a protection violation:

- If an I/O interrupt request occurs, the processor executes the first instruction of the protection violation fault handler before servicing the interrupt request.
- If a protection violation fault occurs while handling any other type of nonprivileged fault, the processor aborts the first fault and processes the protection violation fault. The return block pushed onto the stack for the protection violation fault is undefined, as are the contents of AC0.

Table 9-7 *Protection fault return block*

Doubleword in Block Pushed	Contents
1	PSR (Bits 0-15 contain the processor status register; bits 16-31 contain zeros)
2	AC0
3	AC1
4	AC2
5	AC3
6	PC (Bit 0 contains Carry; bits 1-31 contain the PC of execution if fault type is privileged or I/O, otherwise these bits are undefined.)

Table 9-8 *Protection fault codes*

Code (octal)	Address in AC2	Meaning	Explanation
0	Y	Read violation	Bit 2 of the specified PTE contains a 0.
1	Y	Write violation	Bit 3 of the specified PTE contains a 0.
2	Y	Execute violation	Bit 4 of the specified PTE contains a 0.
3	Y	Validity violation (SBR or PTE)	Bit 0 of the specified SBR or PTE contains a 0.
4	Y	Inward address reference	Attempted data access to a location in an inner segment.
5	Y	Defer (indirect) violation	More than 15 levels (machine-dependent) of indirection specified.
6	Y	Illegal gate	Gate number specified in an inward call is greater than or equal to the maximum number of gates; or a gate bracket access violation.
7	Y	Outward call	Attempted transfer of control from the current segment to another with an outward subroutine call.
10	Y	Inward return	Attempted transfer of control from the current segment to another segment with an inward return from a subroutine.
11	N	Privileged instruction violation	Attempted use of a privileged instruction in a segment other than segment 0.
12	N	I/O protection violation	Attempted use of an I/O instruction when bit 3 of the current segment's SBR is set to 0.
13	N	Reserved	Reserved.
14	N	Invalid micro-interrupt return block	Return block created during a micro-interrupt is incorrect.
15	N	Unimplemented instruction	Specified instruction opcode is not implemented on this machine.
16	N	Reserved	Reserved.
17	N	Invalid form ID (GIS)	Specified form ID does not refer to a defined form.
20	N	Invalid attribute index (GIS)	Specified attribute index does not refer to a defined attribute.
21	N	Invalid CHARBLT source (GIS)	Specified source form is not a one-bit-per-pixel virtual form.

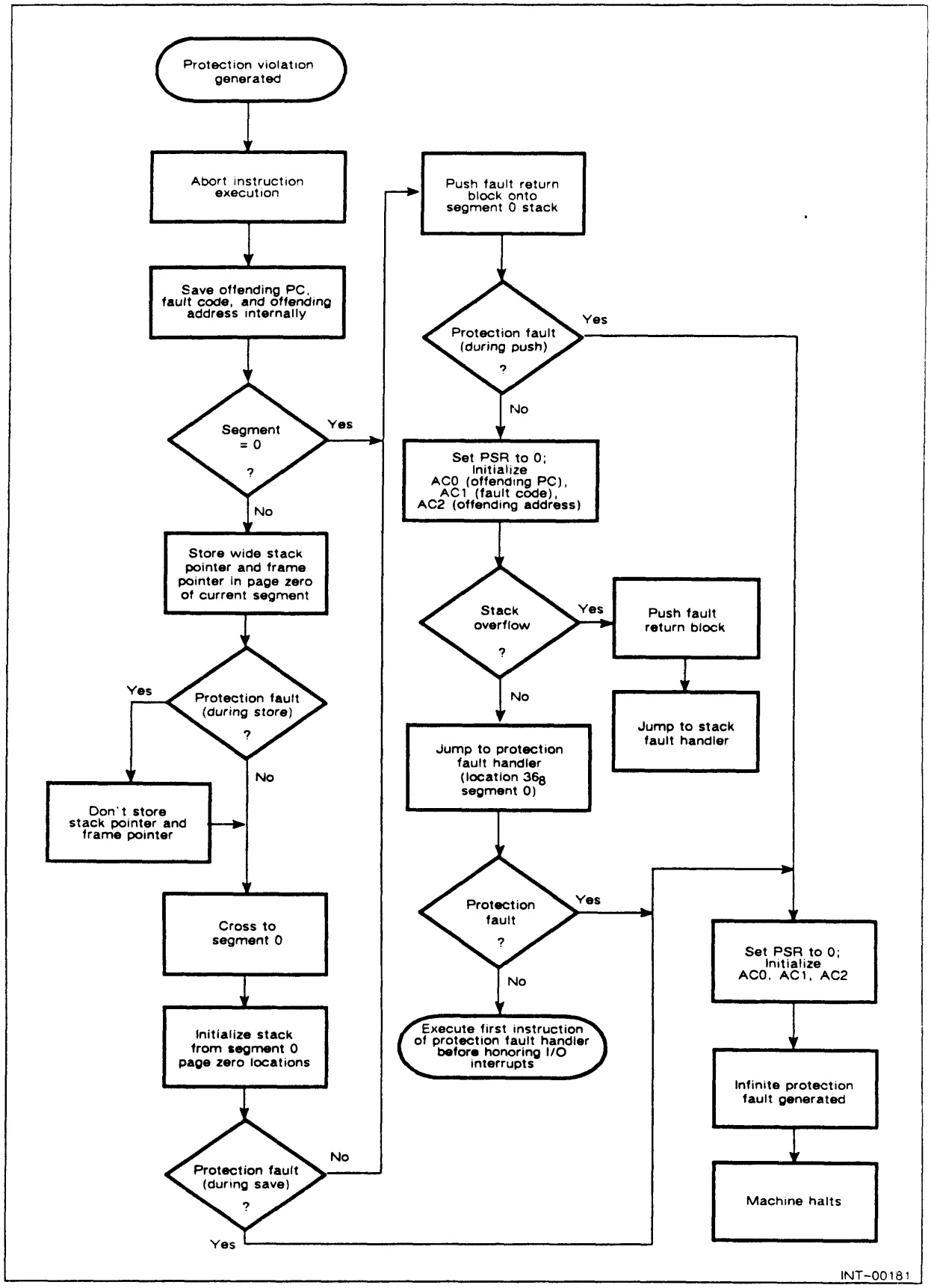


Figure 9-7 *Protection violation sequence*

Unimplemented Instructions

The processor checks for a valid instruction opcode before executing an instruction. If the instruction is implemented on your machine, the operation is performed by hardware.

If the instruction is not implemented or the opcode is invalid, the processor takes a protection fault and reports the fault code for an unimplemented instruction. In the return block pushed onto the wide stack, the PC and AC2 are undefined (unless the PC points to a **PBX** instruction — then AC2 contains the opcode to execute).

NOTE: *Since an unimplemented instruction does not depend on the address translator, the instruction can cause a protection fault when the machine is in physical mode. In this case, the processor uses the protection fault handler specified in physical location* 36_8.

User Protection Fault Handler

As an alternative, the operating system may forward the protection fault data to a "user protection fault handler." This fault handler should exist in the segment of the offending PC (when the protection fault occurs outside of segment 0).

User protection fault handlers can be invoked if the following two conditions, pertaining to the segment where the fault originated (faulting segment), are true:

- the segment has a wide stack to receive the return block, and
- page zero location 36_8 contains a protection fault handler address. (If this location contains a 0, the processor assumes that the program running in that segment does not want to be notified of its protection faults.

If location 36_8 contains a nonzero value, the protection fault can be forwarded to the faulting segment with a routine that does the following:

1. Preserves the accumulator values (passed to the segment 0 protection fault handler) for transmission to the faulting segment.
2. Copies the return block from the segment 0 stack to the faulting segment's stack, and adjusts the stack pointer in the faulting segment accordingly.
3. Tests the new stack pointer against the stack limit for that segment. If a stack overflow condition is detected, then the forwarding procedure must also emulate the stack fault.
4. Resolves the protection fault handler address contained in location 36_8 of the faulting segment. This address must be checked for all access bits required to permit instruction fetching (valid, read, and execute).
5. Returns control to the faulting segment with a **WPOPB** instruction, supplying the following data:
   - Carry (value in return block pushed by the protection fault).
   - PC (value resolved in step 4 above).
   - AC0–AC3 (values reported to segment 0 protection fault handler).
   - PSR (zero).

NOTE: *If a new protection fault occurs at any step in this procedure, the attempt to forward the protection fault to the offending segment should be abandoned.*

Reserved Memory

The processor reserves certain areas of memory for use by the processor and/or an operating system — *page zero* and the *state area*. Each segment has its own page zero; each ECLIPSE MV/Family system (which implements it) has a single state area. If these areas are overwritten (such as by a wide stack which crosses an upper segment boundary overwriting page zero of that segment), data will be lost, and results will be undefined.

Page Zero

When a privileged or nonprivileged fault occurs, the processor transfers control to the appropriate fault handler. The processor reserves memory locations 0 through 47_8 of page zero (locations 0 through 377_8) of each segment for storing certain parameters and the starting addresses of the fault handlers.

The processor interprets page zero locations for segment 0 differently from page zero locations for segments 1 through 7. For example, segment 0 contains pointers to privileged fault handlers, and segments 1 through 7 reserve these locations. Segment 0 locations are listed in Table 9-9; segments 1 through 7 locations are listed in Table 9-10.

Specified addresses for the fault handlers are not indirectable unless otherwise specified. Some pointers are 16 bits long; they can only refer to locations in the first 64 Kbytes of the segment containing the pointer. If the pointer is indirect, all pointers in the indirect chain will only refer to the first 64 Kbytes of the segment. With the address translator enabled, the processor interprets all locations in page zero as logical addresses. With the address translator disabled, only the contents of page zero in segment 0 are valid; the processor interprets page zero addresses as physical ones.

Table 9-9 *Page zero locations for segment 0*

Location (octal)	Name	Contents or Function
0	Interrupt level	Level of interrupt processing: 0 base-level processing nonzero intermediate-level processing
1	I/O handler	Address of I/O interrupt handler (indirectable).
2-3	I/O return address	Address of I/O interrupt return. Location 2 contains the high-order bits; location 3 contains the low-order bits.
4	Vector stack pointer	Low-order 16 bits of vector stack pointer, base, and frame pointer (high-order bits = 0).
5	Current 16-bit narrow mask	Current 16-bit narrow interrupt priority mask.
6	Vector stack limit	Low-order 16 bits of vector stack limit.
7	Vector stack fault address	Address of vector stack fault handler (indirectable).
10-11	Breakpoint address	Address of breakpoint handler (indirectable).
12-13	WXOP origin address	Address of beginning of extended operations table — see the WXOP instruction description.
14	Wide stack fault handler	Address of wide stack fault address handler (indirectable).
15-17	Reserved	Reserved.
20-21	WFP	Wide frame pointer.
22-23	WSP	Wide stack pointer.
24-25	WSL	Wide stack limit.
26-27	WSB	Wide stack base.
30-31	Page fault handler	Address of wide page fault handler.
32-33	Context block pointer	Address of base of context block save area.
34-35	WGP	Gate pointer; address of the gate array.
36	Protection fault handler address	Address of protection fault handler (indirectable).
37	Fixed-point fault handler address	Address of fixed-point fault handler (indirectable).
40	Stack pointer	Address of top of 16-bit narrow stack.
41	Frame pointer	Address of start of current narrow frame minus 1.
42	Stack limit	Address of last normally usable location in narrow stack.
43	Narrow stack fault handler	Address of ECLIPSE 16-bit narrow stack fault handler (indirectable).
44	XOP0 origin address	Address of beginning of narrow extended operations table. See the XOP0 instruction description.
45	Floating-point fault address	Address of floating-point fault handler (indirectable).
46	Decimal/ASCII fault handler	Address of decimal/ASCII fault handler (indirectable).
47	DERR error handler	Address of DERR instruction error/trap handler. See the DERR instruction description.

Table 9-10 *Page zero locations for segments 1 through 7*

Location (octal)	Name	Contents or Function
0-7	Reserved	Reserved.
10-11	Breakpoint address	Address of breakpoint handler (indirectable).
12-13	WXOP origin address	Address of beginning of extended operations table — see the WXOP instruction description.
14	Wide stack fault handler	Address of wide stack fault address handler (indirectable).
15-17	Reserved	Reserved.
20-21	WFP	Wide frame pointer.
22-23	WSP	Wide stack pointer.
24-25	WSL	Wide stack limit.
26-27	WSB	Wide stack base.
30-33	Reserved	Reserved.
34-35	WGP	Gate pointer; address of the gate array.
36	Reserved	Reserved (refer to the section, "Protection Violations," in this chapter for further information).
37	Fixed-point fault handler address	Address of fixed-point fault handler (indirectable).
40	Stack pointer	Address of top of 16-bit narrow stack.
41	Frame pointer.	Address of start of current narrow frame minus 1.
42	Stack limit	Address of last normally usable location in narrow stack.
43	Narrow stack fault handler	Address of ECLIPSE 16-bit narrow stack fault handler (indirectable).
44	XOP0 origin address	Address of beginning of narrow extended operations table. See the XOP0 instruction description.
45	Floating-point fault address	Address of floating-point fault handler (indirectable).
46	Decimal/ASCII fault handler	Address of decimal/ASCII fault handler (indirectable).
47	DERR error handler	Address of DERR instruction error/trap handler. See the DERR instruction description.

State Area

Some ECLIPSE MV/Family computers require that a physically contiguous block of main memory be allocated to the processor for control purposes. This *state area* is available for use by the processor as hardware reserved memory and contains information which the processor can not store in internal processor state. The operating system should treat this area as an extension of the internal processor state and consider the area as unusable memory.

On systems that require a state area, use the privileged Store State Pointer (**SSPT**) instruction to allocate memory; systems which do not require a state area treat this instruction as a no-op. The state area is a system-wide entity; when any processor (in a multiple-processor configuration) issues an **SSPT** instruction, all of the processors have knowledge of this.

The operating system must execute the **SSPT** instruction at system initialization time, before the address translator is enabled. After execution, the state area is available for use by the processor.

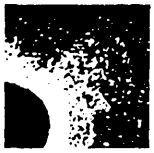
The state area should be set up as soon as practical after system powerup. The generally accepted procedure is to execute an **SSPT** instruction immediately after executing the Load Control Store instruction (**LCS**) and "several" ECLIPSE MV/Family instructions. The "several" instructions should be a few load-, store-, add-, subtract-, and jump-type instructions which would place the machine in an intermediate state and ready for the **SSPT** instruction (the set of valid instructions is machine-specific; these instructions do not check to determine if there is a valid state area). I/O interrupts should not occur between the execution of the **LCS** instruction and the **SSPT** instruction.

The **SSPT** instruction stores the base address for the contiguous block from an accumulator into the state pointer in memory. The operating system then defines the size of the block. For further information, refer to the **SSPT** instruction description.

If it becomes necessary to move the state area (for example, as a result of a hard memory failure within the state area), the operating system should stop any operations that may change the contents of the state area, such as **CIO** or **WLMP** operations. Then, the operating system may perform the move, reloading the state pointer by executing an **SSPT** instruction as a final step. Note that the processor moves the state pointer, but not the data in the state area.

The information within the state area may be lost when another **SSPT** instruction is issued. This information is machine-specific and may include: architectural clock state, GIS state, cross-interrupt state (multiple processors), all or some of the command I/O registers, stop-on-store data, and reference and modify bits. Additionally, in a multiple-processor system, the **JPSTATUS** instruction may incorrectly report the state of other processors in the system (unless they are stopped), and a "loaded" processor (in the **JPLOAD** instruction sense) may not flush correctly (refer to the appropriate instruction description for further information). Any emulated device may also have to be re-initialized with respect to any internal device state (including its mask bit).

End of Chapter



10

ECLIPSE 16-Bit Programming

The ECLIPSE MV/Family 32-bit processor executes 16-bit processor instructions to provide upward program compatibility and for development of 16-bit programs (for systems such as the ECLIPSE C/350 processor). This chapter discusses these capabilities as well as machine-specific restrictions.

Programs that include ECLIPSE 16-bit memory- and stack-reference instructions must meet certain requirements or restrictions as explained in this chapter.

Refer to the *ECLIPSE C/350 Principles of Operation* manual for a further explanation of ECLIPSE C/350 instructions, terms, and conventions.

This chapter explains the differences and similarities between the ECLIPSE 16-bit and ECLIPSE MV/Family (32-bit) systems in the areas of:

- registers and accumulators,
- stack operations,
- faults and interrupts,
- program and subroutine expansion,
- instruction execution,
- program flow,
- fault handling,
- reserved memory,
- CPU identification.

Within this manual, we refer to ECLIPSE 16-bit compatible instructions as ECLIPSE instructions, and ECLIPSE MV/Family-specific instructions as ECLIPSE MV/Family instructions.

ECLIPSE Registers

The majority of ECLIPSE 16-bit registers are physically one and the same as the ECLIPSE MV/Family registers — the differences are in the formats and the number of bits affected. For instance, you can load data into an accumulator with an ECLIPSE 16-bit instruction and then manipulate that data using an ECLIPSE MV/Family 32-bit instruction. This section describes the correspondence between ECLIPSE 16-bit registers and ECLIPSE MV/Family registers; Table 10-1 lists the registers.

The following ECLIPSE MV/Family registers are unaffected by the execution of ECLIPSE 16-bit instructions:

- Processor status register (PSR).

  Refer to the description of the Carry flag in this section.

- Wide stack management registers (WSP, WFP, WSL, WSB).

  ECLIPSE instructions function with the narrow stack. Thus, the instructions use reserved memory locations in page zero of each segment for stack management without affecting the stack management registers in ECLIPSE MV/Family systems. The section, "ECLIPSE Stack," in this chapter describes the narrow stack parameters.

ECLIPSE MV/Family systems implement the following ECLIPSE registers:

- Four 16-bit fixed-point accumulators (AC0, AC1, AC2, AC3).

  The ECLIPSE fixed-point accumulator bits 0 through 15 correspond to the ECLIPSE MV/Family fixed-point accumulator bits 16 through 31 (see Figure 10-1). When an ECLIPSE instruction loads data into an accumulator, bits 16-31 receive the data, while the contents of bits 0-15 are undefined (unless otherwise noted). When using these ECLIPSE instructions in the 32-bit environment, zero-extend (**ZEX** instruction) or sign-extend (**SEX** instruction) the results accordingly. Exceptions to this rule include the **EJSR**, **ELEF**, **FRH**, **JSR** and **LEF** instructions. Refer to the chapter, "Fixed-Point Computation," for the accumulator formats.

  An ECLIPSE instruction (such as the **CLM** instruction) does not alter the contents of an accumulator (bits 16-31) when it reads data from the accumulator.

  When using a fixed-point accumulator for accumulator-relative addressing, the ECLIPSE accumulator bits 1-15 correspond to the ECLIPSE MV/Family accumulator bits 17-31.

- Four 64-bit floating-point accumulators (FPAC0, FPAC1, FPAC2, FPAC3).

  The ECLIPSE floating-point accumulators are identical to the 64-bit ECLIPSE MV/Family floating-point accumulators. Refer to the chapter, "Floating-Point Computation," for the accumulator formats.

- One 32-bit floating-point status register (FPSR).

  The ECLIPSE 32-bit floating-point status register corresponds to bits 0 through 15 and 49 through 63 of the FPSR in ECLIPSE MV/Family systems (see Figure 10-1). Note that bit 48 is reserved and returned as 0. Refer to the chapter, "Floating-Point Computation," for the FPSR contents.

- One 1-bit carry flag (Carry).

  As executing ECLIPSE instructions do not generate fixed-point faults, they do not affect the processor status register (PSR). Certain ECLIPSE arithmetic instructions (such as **ADD** or **DIV**) set the state of the carry bit (Carry). To detect an appropriate fault, it is necessary to check the state of Carry upon completion of these instructions. The carry for ECLIPSE instructions comes from bit 16 of a fixed-point accumulator. The *Instruction Dictionary* describes the ECLIPSE 16-bit instruction set and those instructions which affect Carry.

- One 15-bit program counter (PC).

  The ECLIPSE program counter bits 1 through 15 correspond to the ECLIPSE MV/Family program counter bits 17 through 31 (see Figure 10-1). An ECLIPSE program flow instruction modifies bits 17-31, while the most significant bits are the current segment (bits 1-3) and zeros (bits 4-16). This constrains ECLIPSE addressing to the first 64 Kbytes of the current segment. The following example demonstrates the results of executing a program flow instruction in both the lower 64 Kbytes of a segment and above that limit:

```
Lower 64 Kbytes:    JMP .+1      ;No-op
Above 64 Kbytes:    JMP .+1      ;Resolves to some address in lower 64K memory
```

  NOTE: *Normal program flow and conditional skip instructions are not constrained to the first 64 Kbytes of memory. For example, the execution of the instruction* **MOV# 0,0,SNR** *produces a valid skip or no skip anywhere in memory.*

Table 10-1 *Comparison of ECLIPSE 16-bit and ECLIPSE MV/Family registers*

Register	ECLIPSE (16-bit) (applicable bits)	ECLIPSE MV/Family (32-bit) (applicable bits)
PSR	Not applicable	0-15
WSP, WFP, WSL, WSB	Not applicable	0-31
AC0, AC1, AC2, AC3	16-31	0-31
FPAC0, FPAC1, FPAC2, FPAC3	0-31 (single-precision)	0-31 (single-precision)
	0-63 (double-precision)	0-63 (double-precision)
FPSR	0-15, 49-63	0-63
Carry	0	0
PC	17-31	1-31

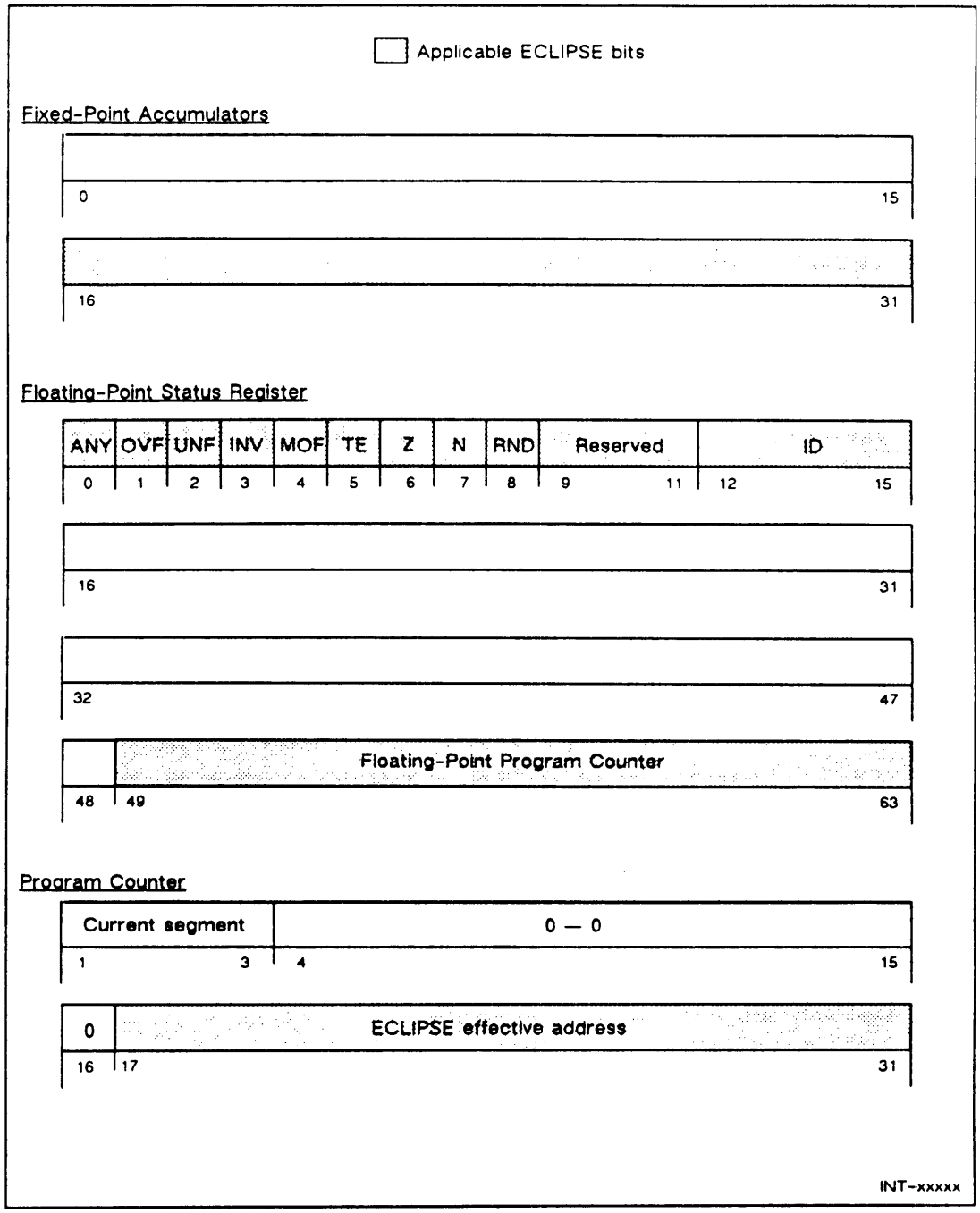


Figure 10-1 *ECLIPSE MV/Family registers with applicable ECLIPSE 16-bit register bits*

ECLIPSE Stack

The ECLIPSE stack (or narrow stack) supports ECLIPSE program development and upward program compatibility. Unlike the wide stack, the narrow stack uses three parameters in reserved page zero memory (of each segment) to define and control the narrow stack.

- The narrow stack limit (location 42_8) defines the upper limit of the narrow stack.

  Although specifying a 16-bit word, the narrow stack limit functions like the wide stack limit.

- The narrow stack pointer (location 40_8) initially defines the lower limit of the narrow stack.

  After accessing the narrow stack, the narrow stack pointer defines the current location of the last word written onto or read from the narrow stack. (Although specifying a 16-bit word, the narrow stack pointer functions like the wide stack pointer.)

- The narrow frame pointer (location 41_8) defines a reference point in the narrow stack.

  Although specifying a 16-bit word, the narrow stack frame pointer functions like the wide stack frame pointer.

NOTE: *There is no parameter for the narrow stack base. To enable narrow stack underflow, initialize the narrow stack pointer to 400_8 and start the narrow stack area at location 401_8.*

The standard ECLIPSE (or narrow) return block consists of five words (see Table 10-2). The return block contains only the least significant 16 bits of the four accumulators, and the least significant 15 bits of the program counter or the frame pointer. An instruction that uses the narrow stack such as **FPSH** (which pushes 18 words) should reserve an additional six words on the stack. The chapter, "Program Flow Management," presents narrow stack fault handling.

Table 10-2 *Standard ECLIPSE (narrow) return block*

Word Number in Block			
Pushed	Popped	Name	Contents
1	5	AC0	Contents of accumulator 0 (bits 16-31)
2	4	AC1	Contents of accumulator 1 (bits 16-31)
3	3	AC2	Contents of accumulator 2 (bits 16-31)
4	2	AC3	Contents of accumulator 3 (bits 16-31)
5	1	CRY/PC	Bit 0 contains Carry. Bits 1-15 contain the 15 least significant bits of either the PC return address or the narrow frame pointer before the push)

ECLIPSE Faults and Interrupts

The processor uses the same pointers a fault or interrupt handlers to service both ECLIPSE 16-bit and ECLIPSE MV/Family 32-bit floating-point faults, decimal/ASCII faults, and I/O interrupts. The ECLIPSE 16-bit processor and the ECLIPSE MV/Family 32-bit processor use different methods to flag an interrupted and resumable **EDIT** instruction.

- For floating-point faults, the processor pushes a return block onto either the narrow or the wide stack. Which stack the processor uses depends on whether the first instruction of the floating-point fault handler is a 16- or 32-bit instruction.
- For decimal/ASCII faults, the processor pushes a return block onto either the narrow or the wide stack. Which stack the processor uses depends on the contents of bit 16 in AC1 (AC1 contains the fault code). If bit 16 equals 1, this is an ECLIPSE fault; if bit 16 equals 0, this is an ECLIPSE MV/Family specific fault.

  Thus, you can upgrade a program written for an ECLIPSE 16-bit processor to incorporate 32-bit processor enhancements. Refer to the chapter, "Program Flow Management," for more information on the fault handlers.
- For I/O interrupts, the processor pushes a return block onto either the narrow or the wide stack. Which stack the processor uses depends on whether the first instruction of the I/O interrupt handler is a 16- or 32-bit instruction.

  Again, this allows you to upgrade a program written for an ECLIPSE 16-bit processor to incorporate 32-bit processor enhancements. Refer to the chapter, "Device Management," for more information on the interrupt handler.
- For an interrupted and resumable **EDIT** instruction, the ECLIPSE 16-bit processor sets AC0 to minus one (177777_8). The ECLIPSE MV/Family 32-bit processor sets the resume flag (IRES) in the PSR, and checks the flag after completing the interrupt. For compatibility, the ECLIPSE MV/Family processor also sets AC0 to minus one.

Expanding an ECLIPSE Program

An ECLIPSE 16-bit program can be expanded by using a specific set of ECLIPSE MV/Family (32-bit) instructions to

- Expand the program beyond 64 Kbytes.
- Use expanded data areas, such as large arrays.
- Use the ECLIPSE MV/Family 32-bit fixed-point arithmetic instructions.

Several methods are available to expand an ECLIPSE 16-bit program beyond 64 Kbytes. The most reliable is to rewrite one of the subroutines so it contains ECLIPSE MV/Family 32-bit instructions and to place it in the segment anywhere above the lower 64 Kbytes. The following requirements must be met when using this method.

- The program must call the expanded subroutine with the **XJSR** or **LJSR** instruction, and establish a wide stack for the subroutine's use.
- The subroutine must begin with a wide special save (**WSSVR** or **WSSVS**) instruction and end with a wide return (**WRTN**) instruction.
- The subroutine must use the ECLIPSE MV/Family 32-bit memory-reference instructions.

To expand data areas for large arrays or buffers, the processor must perform address calculations with 32-bit fixed integer arithmetic, and it must refer to data with the 32-bit memory-reference instructions. The program must then be changed to refer to the expanded data area.

You can also create additional subroutines to maintain the large arrays and to refer to the data through these routines. If you write an additional subroutine, be sure that you refer to the subroutine with the Wide Special Save (**WSSVS**) and Wide Return (**WRTN**) instructions. (Using **SAVE** and **RTN** result in the loss of bits 0 to 15 of the accumulators and the contents of the processor status register.)

To use 32-bit fixed-point arithmetic, all operations on the data (loading, calculations, and storing) must be performed with 32-bit instructions. This can be accomplished by making spot changes or by writing new subroutines; again, care must be taken when mixing these operations with 16-bit operations.

Expanding an ECLIPSE Subroutine

An ECLIPSE 16-bit subroutine can be called from an ECLIPSE MV/Family 32-bit routine using the changes listed in Table 10-3.

Table 10-3 *Alterations to ECLIPSE subroutines*

Changes to ECLIPSE Subroutine	Reason for Change
Replace SAVE with WSSVS or WSSVR and RTN with WRTN.	A routine can call the subroutine from an address which exceeds 16 bits. Also, the accumulators can contain 32-bit entities.
Check external references for 32-bit memory reference instructions.	A routine could pass 32-bit fixed-point data. Also, a called lower-level subroutine can be located in an address space which exceeds 16 bits.
Check short negative references on the stack that may require 32-bit displacements.	Using WSSVS or WSSVR in this subroutine changes the size of the pushed stack block, requiring the assembler to recalculate the negative reference.
Change a routine (to save the 31-bit PC) that calls a subroutine with a JSR through page zero by using LJSR or XJSR in the calling routine to save the 31-bit PC.	A long address requires 31 bits and can cause the program to exhaust page zero locations.

ECLIPSE Instructions

This section presents instructions that refer to memory or to the narrow stack. The remaining ECLIPSE 16-bit instructions (such as **ADD**) are presented with the other ECLIPSE MV/Family processor instructions.

Note that **NIO**[*f*] *ac*,**CPU** instructions (where *ac* is not specified as 0) are reserved or assigned an ECLIPSE MV/Family specific function. For example, the **NIO CPU** is the Load Control Store instruction (**LCS**).

The *Instruction Dictionary* and *ECLIPSE MV/Family Instruction Reference Booklet* identify the ECLIPSE 16-bit instructions supported on the ECLIPSE MV/Family processors. The machine-specific supplement lists all instructions supported by that particular processor.

ECLIPSE MV/Family Instruction Compatibility

The ECLIPSE MV/Family systems limit ECLIPSE (16-bit) program flow instructions to an addressing range of 0 to 64 Kbytes in the current segment.

ECLIPSE instructions that load AC3 with the address of the next instruction (such as *jump to subroutine*) or push the address of the next instruction onto the narrow stack (such as *push and jump*) calculate effective addresses within the lower 64 Kbytes of the present segment.

The ECLIPSE MV/Family processors treat certain ECLIPSE instruction opcodes as ECLIPSE MV/Family specific instructions rather than ECLIPSE 16-bit instructions. These include the ECLIPSE ALC (arithmetic and logic class) instructions which contain both the "no-load" and "never skip" codings, and the ECLIPSE **XOP** and **XOP1** opcodes (these have been replaced by **XOP0**). ECLIPSE (16-bit) programs that contain these instructions should be rewritten before running on an ECLIPSE MV/Family system.

In addition, the ECLIPSE MV/Family systems do not support the following ECLIPSE instructions:

- Floating-point function instructions (**FCOSD, FCOSS, FEXPD, FEXPS, FLOGS, FSIND, FSINS, FPLYD, FPLYS, FSQRD, FSQRS**).
- **VCT, SYC,** and **LMP.**

Note that the results of some ECLIPSE 16-bit instructions may differ when they execute on an ECLIPSE MV/Family processor and an ECLIPSE 16-bit processor. For instance, after execution, the ECLIPSE instructions, **LDI** and **STI**, return different results to AC3. If executing on an ECLIPSE MV/Family system, AC3 contains the address of the first byte following the integer field. If executing on an ECLIPSE 16-bit system, the contents of AC3 are undefined following execution of these instructions.

ECLIPSE Memory Reference Instructions

The processor considers the ECLIPSE memory reference instructions to be executing within the first 32 Kwords (64 Kbytes) of the current segment. If the processor executes an ECLIPSE memory reference instruction above the 32 Kword limit, the effective address reverts to within the ECLIPSE address space (lower 32 Kwords).

To refer to a word with an ECLIPSE memory reference instruction, the processor forms an effective address as shown in Figure 10-2 and described in Table 10-4. Figure 10-3 illustrates ECLIPSE effective addressing.

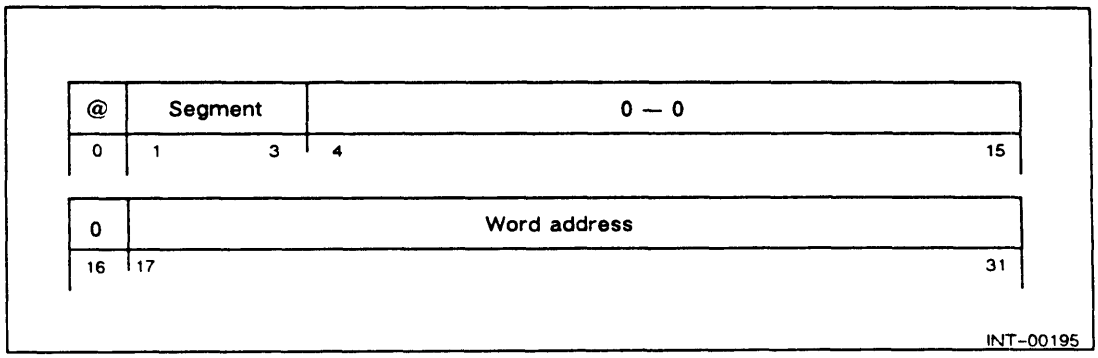


Figure 10-2 *ECLIPSE word addressing format*

Table 10-4 *ECLIPSE word addressing format description*

Bits	Contents	Description
0	@	Indirect bit — When set to 1, forces indirect addressing through a single word pointer.
1-3	Segment	Number of the current segment.
4-16	0 — 0	The processor sets these bits to 0.
17-31	Word address	Identifies a 16-bit word in the first 64 Kbytes of the current segment.

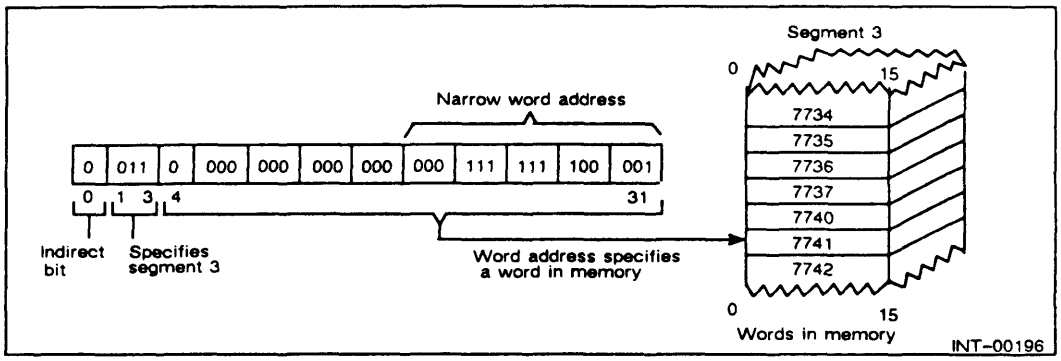


Figure 10-3 *ECLIPSE effective addressing*

To refer to a byte with an ECLIPSE memory reference instruction, the processor forms a byte address as shown in Figure 10-4 and described in Table 10-5. Figure 10-5 illustrates ECLIPSE byte addressing.

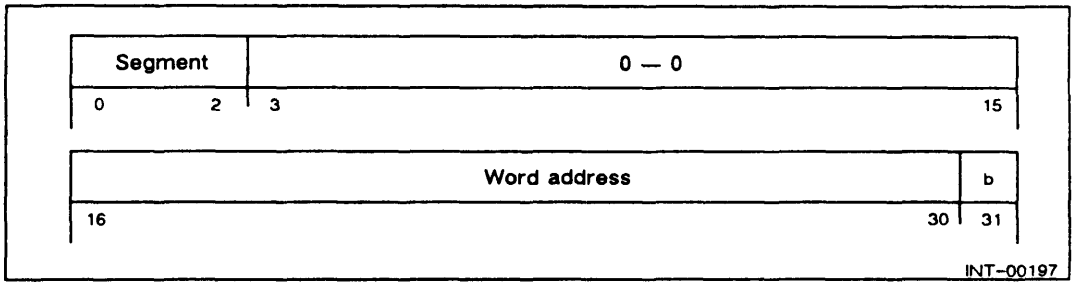


Figure 10-4 *ECLIPSE byte addressing format*

Table 10-5 *ECLIPSE byte addressing format description*

Bits	Contents	Description
0-2	Segment	Number of the current segment.
3-15	0 — 0	The processor sets these bits to 0.
16-30	Word address	Identifies a 16-bit word in the first 64 Kbytes of the current segment.
31	b	Byte indicator — Specifies the high or low byte. Set to 0, indicates the most significant byte (bits 0-7) of a word; set to 1, indicates the least significant byte (bits 8-15) of a word.

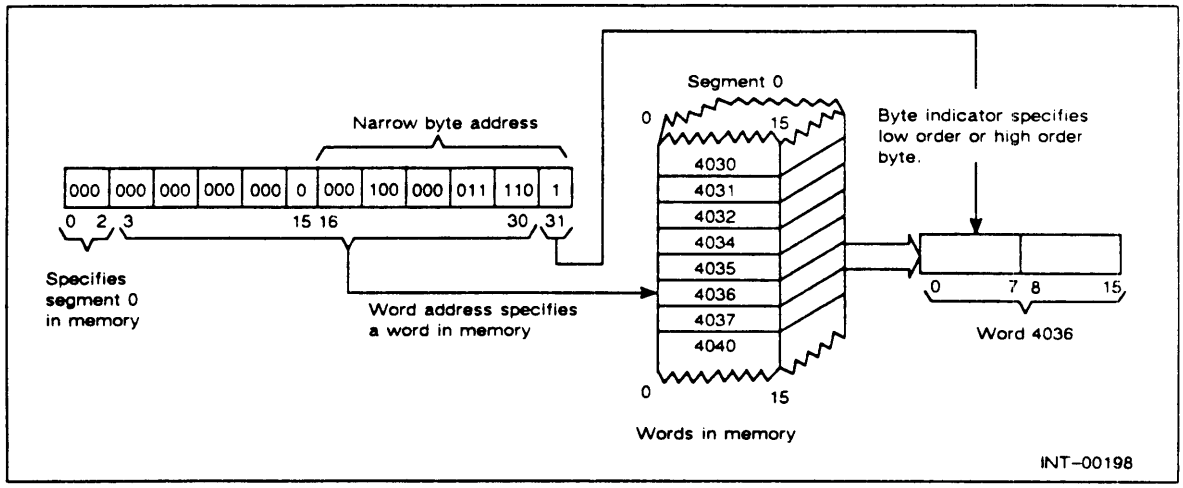


Figure 10-5*ECLIPSE byte addressing*

To refer to a bit with an ECLIPSE memory reference instruction (**BTO, BTZ, SNB, SZB,** and **SZBO**), the processor forms a bit pointer from the contents of two accumulators (*acs* and *acd*). The bit pointer is composed of a word pointer and a bit identifier. The word pointer consists of an effective address (in the *acs* accumulator) and a word offset (in the *acd* accumulator). The bit identifier is located in the least significant bits of the *acd* accumulator.

Figure 10-6 shows the accumulator formats for the **BTO, BTZ, SNB, SZB,** and **SZBO** instructions; Table 10-6 describes the accumulator formats.

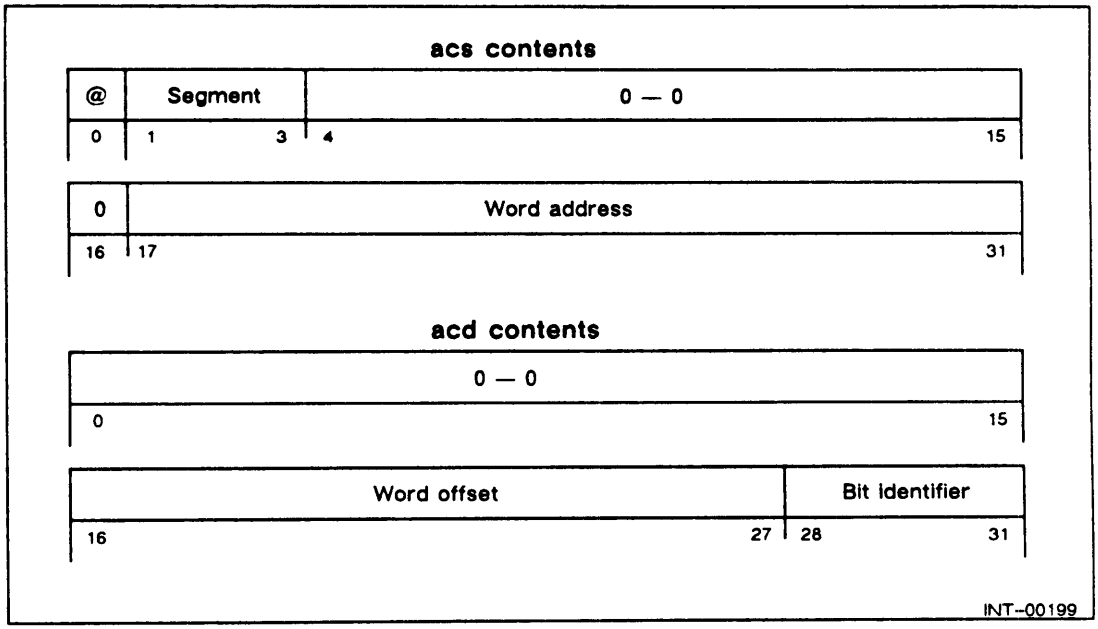


Figure 10-6 *ECLIPSE bit addressing format*

Table 10-6 *ECLIPSE bit addressing format description*

Bits	Contents	Description
acs		
0	@	Indirect bit — When set to 1, forces indirect addressing through a single word pointer.
1-3	Segment	Number of the current segment.
4-16	0 — 0	The processor sets these bits to 0.
17-31	Word address	Identifies a 16-bit word in the current segment.
acd		
0-15	0 — 0	The processor sets these bits to 0.
16-27	Word offset	The processor adds these bits, an unsigned integer, to the effective address to determine a final word address.
28-31	Bit identifier	Specifies the bit position (0-15) in the final word.

The processor uses the *acs* accumulator contents to calculate the effective address. For the **BTO, BTZ, SNB, SZB,** and **SZBO** instructions, the processor limits effective addressing to the first 64 Kbytes of the current segment. If a bit instruction specifies the two accumulators (*acs* and *acd*) as the same accumulator, then the effective address is 0 in the current segment.

In Figure 10-7, notice that the processor adds the word offset, an unsigned integer, to the effective address to determine a final word address. The processor then locates the bit using the bit identifier, which specifies the a bit position (in the range of 0 through 15) in the final word.

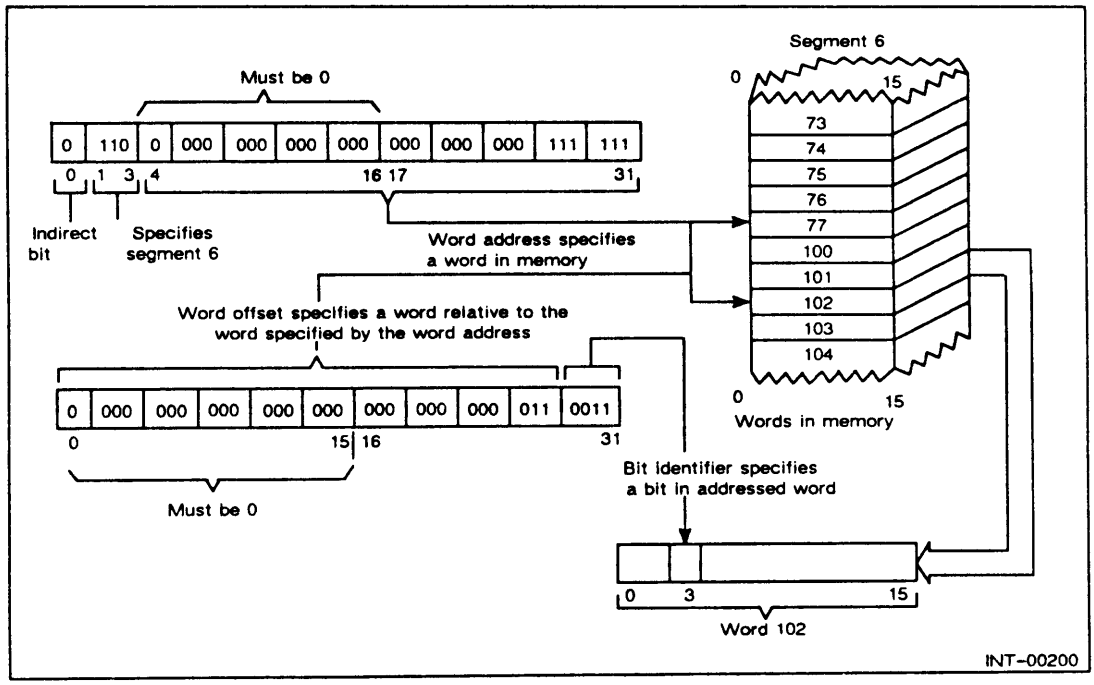


Figure 10-7 BTO, BTZ, SNB, SZB, *and* **SZBO** *bit addressing*

ECLIPSE Fixed-Point Instructions

Table 10-7 lists the ECLIPSE fixed-point instructions that refer to memory. The table also shows an equivalent ECLIPSE MV/Family (32-bit) processor instruction that can be substituted to expand (within the segment) the memory address range.

Unless otherwise stated, the ECLIPSE instruction and the ECLIPSE MV/Family processor equivalent instruction use identical

- Singleword or doubleword instruction length
- Argument string
- Data access for writing and for reading (register or memory)
- Data precision of 16 bits

An equivalent ECLIPSE MV/Family instruction, however, uses a doubleword indirect pointer, while the ECLIPSE instruction uses a single-word indirect pointer.

Table 10-7 *ECLIPSE fixed-point computing instructions*

ECLIPSE 16-bit Instruction	Operation	ECLIPSE MV/Family Equivalent Instruction
BAM	Block add and move	--
BLM	Block move	WBLM
BTO	Set bit to one	WBTO
BTZ	Set bit to zero	WBTZ
CLM	Compare to limits and skip	WCLM
CMP	Character compare	WCMP
CMT	Character move until true	WCMT
CMV	Character move	WCMV
COB	Count bits	WCOB
CTR	Character translate and compare	WCTR
DSZ	Decrement and skip if zero	XNDSZ *
EDIT	Edit decimal and alphanumeric 16-bit data	WEDIT
EDSZ	Extended decrement and skip if zero	XNDSZ
EISZ	Extended increment and skip if zero	XNISZ
ELDA	Extended load accumulator	XNLDA
ELDB	Extended load byte (from memory to AC)	XLDB
ESTA	Extended store accumulator	XNSTA
ESTB	Extended store byte (right byte of AC to byte in memory)	XSTB
ISZ	Increment and skip if zero	XNISZ *
LDA	Load accumulator	XNLDA *
LDB	Load byte (from memory to AC)	WLDB
LSN	Load sign	WLSN
POP	Pop multiple accumulators	WPOP
PSH	Push multiple accumulators	WPSH
SNB	Skip on nonzero bit	WSNB
SZB	Skip on zero bit	WSZB
SZBO	Skip on zero bit and set to one	WSZBO
STA	Store accumulator	XNSTA *
STB	Store byte (right byte of AC to byte in memory)	WSTB

* *The ECLIPSE MV/Family equivalent instruction is two words in length.*

ECLIPSE Floating-Point Instructions

Table 10-8 lists the ECLIPSE floating-point instructions that refer to memory. The table also shows an equivalent ECLIPSE MV/Family instruction that can be substituted to expand (within the segment) the memory address range.

Unless otherwise stated, the ECLIPSE instruction and the ECLIPSE MV/Family equivalent instruction use identical

- Singleword or doubleword instruction length
- Argument strings
- Data accesses for writing and for reading (register or memory)
- Data precisions of 16, 32, or 64 bits

An equivalent ECLIPSE MV/Family instruction, however, uses a doubleword indirect pointer, while the ECLIPSE instruction uses a single-word indirect pointer.

When the processor converts a floating-point number to a fixed-point integer, it correctly converts the largest negative number without a mantissa overflow. For single-precision, the processor converts the integer portion of floating-point numbers to an integer ranging from -32,768 to +32,767, inclusive. For double-precision, the processor converts the integer portion to an integer ranging from -2,147,483,648 to +2,147,483,647, inclusive.

Table 10-8 *ECLIPSE floating-point computing instructions*

ECLIPSE 16-bit Instruction	Operation	ECLIPSE MV/Family Equivalent Instruction
FAMD	Add double (memory to FPAC)	XFAMD
FAMS	Add single (memory to FPAC)	XFAMS
FDMD	Divide double (FPAC by memory)	XFDMD
FDMS	Divide single (FPAC by memory)	XFDMS
FFMD	Fix to memory (FPAC to memory)	WFFAD *
FLDD	Load floating-point double	XFLDD
FLDS	Load floating-point single	XFLDS
FLMD	Float from memory	WFLAD *
FLST	Load floating-point status register	LFLST **
FMMD	Multiply double (FPAC by memory)	XFMMD
FMMS	Multiply single (FPAC by memory)	XFMMS
FPOP	Pop floating-point state	WFPOP
FPSH	Push floating-point state	WFPSH
FSMD	Subtract double (memory from FPAC)	XFSMD
FSMS	Subtract single (memory from FPAC)	XFSMS
FSST	Store floating-point status register	LFSST **
FSTD	Store floating-point double	XFSTD
FSTS	Store floating-point single	XFSTS
LDI	Load integer (memory to FPAC)	WLDI
LDIX	Load integer extended (memory to FPAC)	WLDIX
STI	Store integer (FPAC to memory)	WSTI
STIX	Store integer extended (FPAC to memory)	WSTIX

* *The* **WFFAD** *and* **WFLAD** *instructions use the 32 bits of a fixed-point accumulator, while the equivalent ECLIPSE instructions use two memory words.*

** *The* **LFLST** *and* **LFSST** *instructions are triple-word instructions, while the ECLIPSE instructions are doubleword instructions.*

Floating-Point Numerical Algorithms

The ECLIPSE floating-point loads (**FLDS** and **FLDD**) do not correct impure zero input. All loads simply move the memory operand to the specified floating-point accumulator. No normalization and correction to true zero is performed. The Z and N bits of the floating-point status register (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 not normalized.

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 ECLIPSE **FFAS** and **FFMD** instructions leave the Z and N bits of the FPSR unchanged.

Otherwise, when bit 8 (RND) of the FPSR is 0, the results of the floating-point computation performed on the ECLIPSE MV/Family processor are identical to those obtained on an ECLIPSE 16-bit processor.

ECLIPSE Program Flow Instructions

Table 10-9 lists ECLIPSE program flow instructions that refer to memory. The table also lists an equivalent ECLIPSE MV/Family instruction that can be substituted to expand (within the segment) the memory address range and enable use of the wide stack.

Unless otherwise stated, the ECLIPSE instruction and the ECLIPSE MV/Family equivalent instruction use identical

- Singleword or doubleword instruction length
- Argument strings
- Data accesses for writing and for reading (register or memory)

The data precision changes from 16 to 32 bits.

An equivalent ECLIPSE MV/Family instruction, however, uses a doubleword indirect pointer, while the ECLIPSE instruction uses a single-word indirect pointer.

Table 10-9 *ECLIPSE program flow management instructions*

ECLIPSE 16-bit Instruction	Operation	ECLIPSE MV/Family Equivalent Instruction
DSPA	Dispatch	LDSP
EJMP	Extended jump	XJMP
EJSR	Extended jump to subroutine	XJSR
ELEF	Extended load effective address	XLEF
JMP	Jump	--
JMP ,1	Jump, relative to the program counter	WBR
JSR	Jump to subroutine	--
LEF	Load effective address	--
POPB	Pop block and execute (return from XOP0)	WPOPB
POPJ	Pop PC and jump (return with PSHJ)	WPOPJ
PSHJ	Push jump (return with POPJ)	XPSHJ
PSHR	Push return address (pop with POPJ)	--
RSTR	Restore (return from VCT -- mode E)	WRSTR **
RTN	Return	WRTN *
SAVE	Save (used with JSR)	WSSVR* WSSVS *
SAVZ	Save without arguments (used with JSR)	WSSVR* WSSVS *
XOP0 ***	Extended operation (return with POPB)	WXOP ***

* *The* WRTN, WSSVS, *and* WSSVR *instructions modify the fixed-point overflow mask (OVK) in the processor status register, and use a return block of six doublewords.*

** *The* WRSTR *and* XVCT *instructions use the wide stack and are equivalent to the ECLIPSE instructions,* RSTR *and* VCT *(mode* E).

*** *The* XOP0 *and* WXOP *instructions are doubleword instructions.*

ECLIPSE Stack Instructions

Table 10-10 lists ECLIPSE stack instructions that refer to memory. The table also lists an equivalent ECLIPSE MVFamily instruction that can be substituted to expand the memory address range (within the segment).

Unless otherwise stated, the ECLIPSE instruction and the ECLIPSE MV/Family equivalent instruction use identical

- Singleword or doubleword instruction length
- Argument strings
- Data accesses for writing and for reading (register or memory)

The data precision changes from 16 to 32 bits.

An equivalent ECLIPSE MV/Family instruction, however, uses a doubleword indirect pointer, while the ECLIPSE instruction uses a single-word indirect pointer.

Table 10-10 *ECLIPSE stack management instructions*

ECLIPSE 16-bit Instruction	Operation	ECLIPSE MV/Family Equivalent Instruction
MSP	Modify stack pointer	WMSP
POP	Pop multiple accumulators	WPOP
POPB	Pop block and execute (return from XOP0)	WPOPB
POPJ	Pop PC and jump	WPOPJ
PSH	Push multiple accumulators	WPSH
PSHJ	Push jump	XPSHJ
PSHR	Push return address	XPEF
RSTR	Restore (return from VCT -- mode E)	WRSTR **
RTN	Return	WRTN *
SAVE	Save (used with JSR)	WSSVR*
		WSSVS *
SAVZ	Save without arguments (used with JSR)	WSSVR*
		WSSVS *
XOP0 ***	Extended operation (return with POPB)	WXOP ***

* *The* WRTN, WSSVS, *and* WSSVR *instructions modify the fixed-point overflow mask (OVK) in the processor status register, and use a return block of six doublewords.*

** *The* WRSTR *and* XVCT *instructions use the wide stack and are equivalent to the ECLIPSE instructions,* RSTR *and* VCT *(mode E).*

*** *The* XOP0 *and* WXOP *instructions are doubleword instructions.*

Program Flow

The program counter governs program flow management as described in the chapter, "Program Flow Management." The program counter contents are illustrated in the section, "ECLIPSE Registers," of this chapter.

All ECLIPSE 16-bit instructions should work correctly at any program counter (PC) value. The exceptions are the following instructions that should not be used above the lower 64 Kbytes of the segment: **LDI**, **LDIX**, **STI**, **STIX**, **EDIT**, and **LSN**. The ECLIPSE MV/Family equivalent instructions are respectively: **WLDI**, **WLDIX**, **WSTI**, **WSTIX**, **WEDIT**, and **WLSN**.

For any ECLIPSE program executing on an ECLIPSE MV/Family computer, when either the PC contains 77777_8 and increments to refer to the next instruction, or an instruction causes a skip over 77777_8, the PC does not wrap around to 0. The PC increments to the next value (such as 100000_8), and the processor executes the instruction at this location.

When using the ECLIPSE instructions, **BAM**, **BLM**, **CMP**, **CMT**, **CMV**, **CTR**, and **EDIT**, address wraparound may not occur at 77777_8. This means that an ECLIPSE program counter can possibly generate logical addresses larger than 64 Kbytes. In this situation, results are undefined. If any of these instructions move data with descending addresses and cross a segment boundary, a protection fault occurs, and AC1 will contain the protection code 4.

The ECLIPSE program flow instructions load bits 17 through 31 of the PC with the address generated by the program flow instruction. The segment bits (1 through 3) remain unchanged, and bits 4 through 16 are set to 0.

Fault Handling

ECLIPSE fault handling is identical to the handling of ECLIPSE MV/Family system nonprivileged faults as described in the chapter, "Program Flow Management." If the execution of an ECLIPSE instruction causes a fault, the processor uses the narrow stack.

In addition, the ECLIPSE MV/Family processor responds to floating-point traps upon completion of the floating-point instruction that caused the fault. In the ECLIPSE C/350 system, 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 program counter (FPPC) in the floating-point status register contains the address of the first floating-point instruction that caused a fault.

Note that an ECLIPSE commercial (decimal/ASCII) fault loads different information into AC0, AC2, and AC3 after the fault occurs. 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 ECLIPSE C/350 processor.

When the ECLIPSE instructions, **DIVS** or **DIVX**, produce a result of -32,768, the results will vary depending on which processor is executing these instructions.

- The ECLIPSE MV/Family processor sets Carry to 0 (meaning no overflow).
- The ECLIPSE C/350 processor sets Carry to 1 (indicating an overflow).

Note that ECLIPSE MV/Family wide divide instructions set overflow to 0 when returning a result of -32,768.

Refer to the section, "Decimal and ASCII Data Faults," in the chapter, "Program Flow Management," and the "Fault Codes" appendix for a listing of the error codes returned to AC1 when a decimal/ASCII fault occurs.

Reserved Memory

Reserved page zero memory locations differ between ECLIPSE 16-bit systems and ECLIPSE MV/Family 32-bit systems. For instance, ECLIPSE MV/Family computers do not implement ECLIPSE auto-increment and auto-decrement locations 20_8 through 37_8 (these locations are reserved to store certain system parameters). Refer to the chapter, "Memory and System Management," or the appendix, "Reserved Memory Locations," for the contents of page zero locations.

CPU Identification

The **ECLID** and **NCLID** instructions return central processor information.

The **NCLID** instruction loads the CPU identification into bits 16 through 31 of three accumulators (AC0, AC1, and AC2). The **NCLID** instruction can execute only with the LEF mode disabled. With the LEF bit enabled, this instruction becomes an LEF instruction.

Accumulator formats are listed in the *Instruction Dictionary* descriptions for each instruction and in the **"Register Fields"** appendix.

End of Chapter

A

Register Fields

This appendix describes the formats of registers that programmers can access on the ECLIPSE MV/Family computers.

The general information presented in this appendix applies to all ECLIPSE MV/Family computers. Refer to the appropriate supplement for machine-specific details.

Table A-1 summarizes the registers and their contents.

Table A-1 *Registers and contents*

Register	Contents
Segment Base	Information about logical address translation.
Program Counter	Logical address of currently executing instruction.
Processor Status	Information about fixed-point computations.
Floating-Point Status	Information about floating-point computations.
DCH/BMC Status	Information about data channel and burst multiplexor channel maps.
CPU Identification	Accumulators with information pertaining to the processor.

Segment Base Registers

The 32-bit segment base registers (SBRs) contain information for the logical address translation mechanism and for I/O protection. The format is diagrammed next and explained in Table A-2.

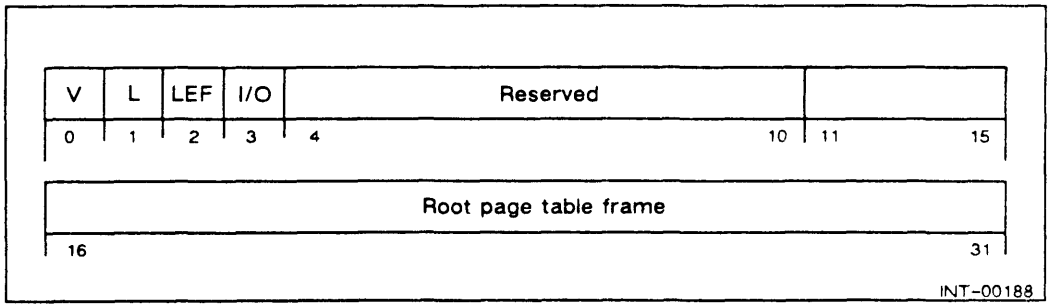


Table A-2 *Segment base register contents*

Bit	Name	Contents or Function
0	V	Segment-validity flag. Indicates ability of processor to refer to a segment. If 0, this is an invalid segment. If 1, this is a valid segment.
1	L	Translation-level flag. If 0, this is a one-level pagetable. If 1, this is a two-level pagetable.
2	LEF	Mode flag. If 1, the processor executes the instruction as an LEF instruction. If 0, the processor executes the instruction as an I/O instruction.
3	I/O	I/O validity flag. If 0, I/O operations are illegal from this segment. If 1, I/O operations are legal from this segment.
4-10	Reserved	Reserved for internal Data General use.
11-31	Root pagetable frame	Specifies the most significant bits of the physical address for the root pagetable page.

Program Counter

The 31-bit program counter (PC) contains the logical address of the currently executing instruction. PC formats are diagrammed and described below.

PC Format for Execution of ECLIPSE MV/Family Programs

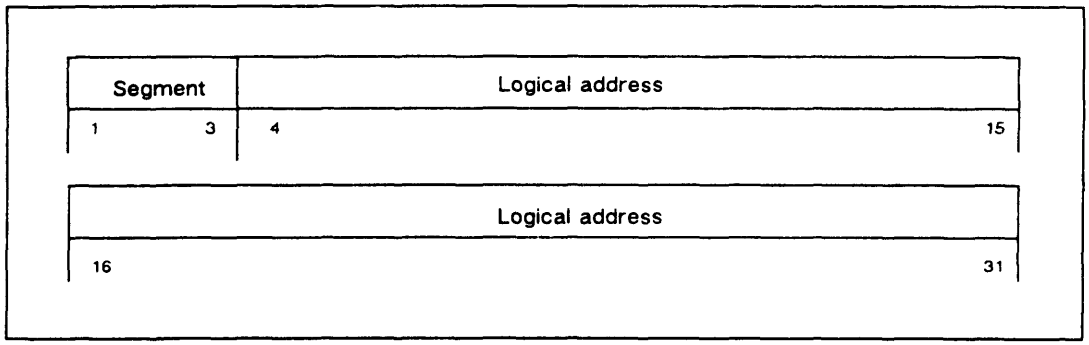


Table A-3 *Program counter format for ECLIPSE MV/Family programs*

Name	Bits	Meaning
Segment	1-3	Current segment of program execution.
Logical Address	4-31	Logical word address within the segment.

PC Format Altered by ECLIPSE 16-Bit Program Flow Instructions

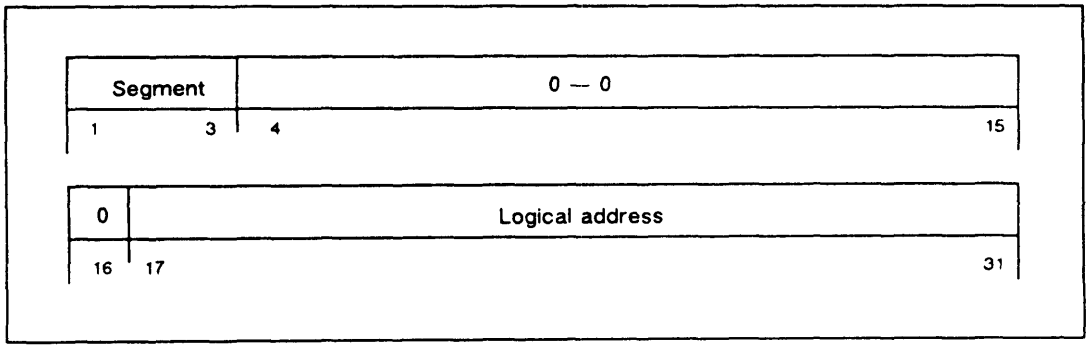


Table A-4 *Program counter modified by ECLIPSE 16-bit instructions*

Name	Bits	Meaning
Segment	1-3	Current segment of program execution.
0 — 0	4-16	Set to 0 by instruction.
Logical Address	17-31	Logical word address within the segment.

Processor Status Register

Only ECLIPSE MV/Family specific instructions affect the 16-bit processor status register (PSR). The format of the PSR is diagrammed below and described in Table A-5.

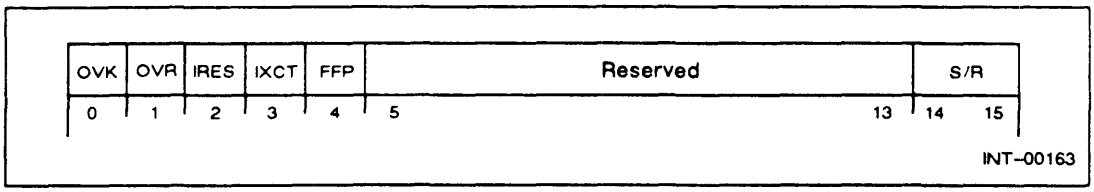


Table A-5 *Processor status register contents*

Bit	Mnemonic	Function
0	OVK	Overflow mask. If 0, no fixed-point overflow trap. If 1, trap on OVR sett to 1.
1	OVR	Fixed-point overflow flag. If 0, no fixed-point overflow. If 1, fixed-point overflow occurred.
2	IRES	Interrupt resume flag. If 0, interrupted instruction begins initial execution. If 1, interrupted instruction resumes execution.
3	IXCT	Interrupt-executed opcode flag. If 1, interrupted instruction was inserted into instruction stream.
4	FFP	Floating-point fault pending flag.
5-13	Reserved	Reserved for future use.
14-15	S/R	Software reserved in return block

NOTE: *Any instruction that loads OVK and OVR as part of its execution does not cause an overflow fault even if both bits are set to 1. For all ECLIPSE 16-bit instructions, overflow equals 0, leaving OVR unchanged.*

Floating-Point Status Register

Both ECLIPSE MV/Family-specific and ECLIPSE 16-bit instructions affect the 64-bit floating-point status register (FPSR). The format of the FPSR is diagrammed as shown and described in Table A-6.

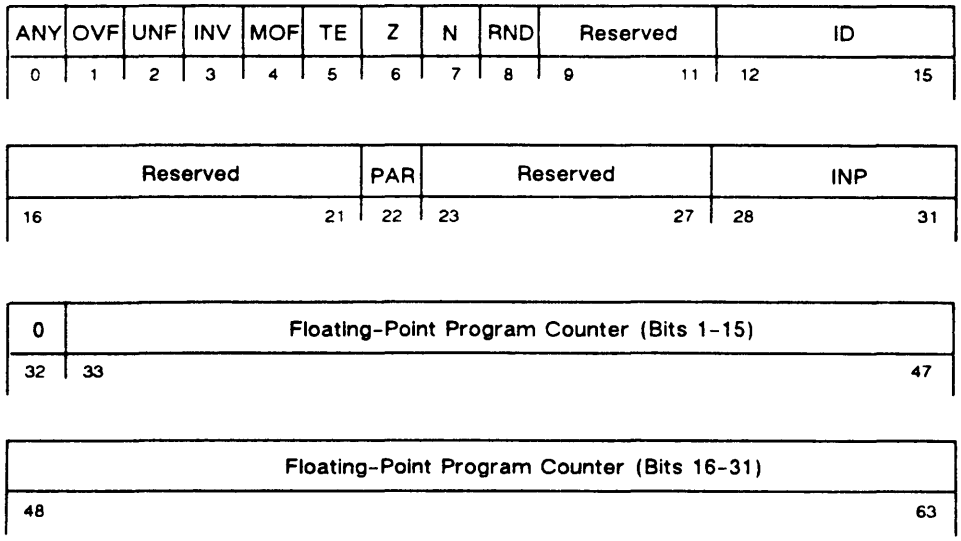


Table A-6 *Floating-point status register contents*

Bits	Name	Contents or Function
0	ANY	Error status flag — indicates any of bits 1 through 4 is set to 1.
1	OVF	Exponent overflow flag.
2	UNF	Exponent underflow flag.
3	INV	Invalid input argument error flag.
4	MOF	Mantissa overflow flag.
5	TE	Trap enable mask; if set to 1, setting any bit 1 through 4 results in floating-point fault.
6	Z	True zero flag.
7	N	Negative flag.
8	RND	Floating-point rounding flag.
9-11	Reserved	Reserved and returned as zeros.
12-15	ID	Floating-point identification code.
16-21	Reserved	Reserved and returned as zeros.
22	PAR	Floating-point operation flag (serial or parallel).
23-27	Reserved	Reserved and returned as zeros.
28-31	INP	Invalid input argument indicator.
32	0	Must be 0.
33-63	Floating-Point Program Counter	If floating-point fault occurs, contains address of first floating-point instruction that caused fault.

DCH/BMC Status Registers

This section describes three registers: I/O channel status register, I/O channel mask register, and I/O channel definition register.

I/O Channel Status Register

The read-only I/O channel status register (7700_8) provides I/O channel status information. The register format follows; Table A-7 describes the format.

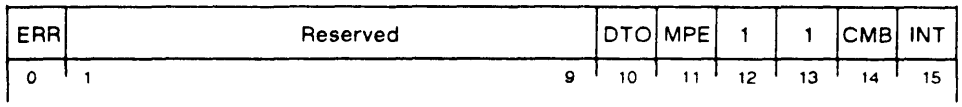


Table A-7 *I/O channel status register contents*

Bits	Name	Contents or Function
0	ERR	Error. If 1, the I/O channel has detected an error. This bit is set to 1 if any error-indicating bit in the IOC status register is set to 1.
1-9	Reserved	Reserved for future use and returned as zeros.
10	DTO	DCH time-out error. If 1, a DCH read-modify-write operation time-out error has occurred.
11	MPE	Map parity error. If 1, a map parity error has occurred.
12	1	Always set to 1, indicating extended DCH map slots and operations are supported.
13	1	Always set to 1.
14	CMB	Current state of the mask bit for this I/O channel (refer to the I/O channel mask register format description).
15	INT	Interrupt pending; if 1, the channel is attempting to interrupt the CPU.

I/O Channel Mask Register Format

The write-only I/O channel mask register (7701_8) specifies a mask flag for each channel. The register format follows; Table A-8 describes the format.

NOTE: *A command I/O instruction (***CIO, CIOI***) that reads the I/O channel mask register produces undefined results.*

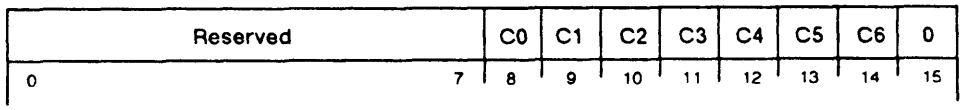


Table A-8 *I/O channel mask register contents*

Bits	Name	Contents or Function
0-7	Reserved	Reserved for future use; should be set to 0.
8	C0	I/O channel 0 mask *
9	C1	I/O channel 1 mask *
10	C2	I/O channel 2 mask *
11	C3	I/O channel 3 mask *
12	C4	I/O channel 4 mask *
13	C5	I/O channel 5 mask *
14	C6	I/O channel 6 mask *
15	0	Reserved and set to 0.

* *If 1, prevents all devices connected to the indicated I/O channel from interrupting the CPU. A system reset sets C0 to zero and C1 through C6 to ones.*

I/O Channel Definition Register Format

The I/O channel definition register (6000_8) provides status information. The format for this register is diagrammed below and described in Table A-9.

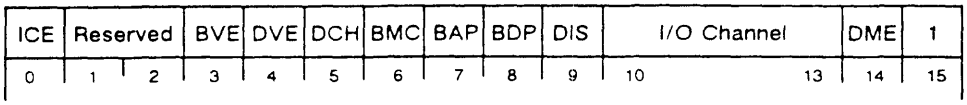


Table A-9 *I/O channel definition register contents*

Bits	Name	Contents or Function
0	ICE *	I/O channel error flag. 1 Error occurred on I/O channel . 0 Only when all other error bits are 0.
1, 2	Reserved	Reserved for future use and returned as 0.
3	BVE *†	BMC validity error flag· if 1, BMC address validity protect error has occurred.
4	DVE *†	DCH validity error flag; if 1, DCH address validity protect error has occurred.
5	DCH	DCH transfer flag; if 1, a DCH transaction is in progress (read-only bit).
6	BMC	BMC transfer flag; if 1, a BMC transfer is in progress (read-only bit).
7	BAP *†	BMC address error; if 1, the channel has detected an address parity error.
8	BDP *†	BMC data error; If 1, the channel has detected a data parity error.
9	DIS *	Disable block transfer; if 1, disables BMC block transfers to and from I/O memory port.
10-13	I/O channel	I/O channel number.
14	DME *	DCH mode; if 1, DCH mapping is enabled.
15	1	Always set to 1.

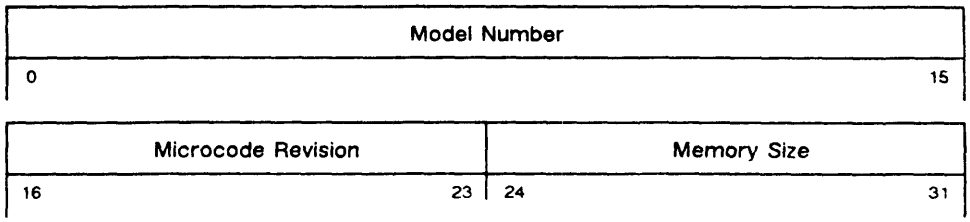
* *The* **IORST** *and* **PRTRST** *instructions clear these bits.*
† *Writing to these bits with a 1 complements them.*

CPU Identification

The three Load CPU Identification instructions — **LCPID**, **ECLID**, and **NCLID** — return processor information to specified accumulators. The accumulator formats and descriptions follow.

LCPID and ECLID Instructions

The **LCPID** and **ECLID** instructions load a doubleword into AC0.

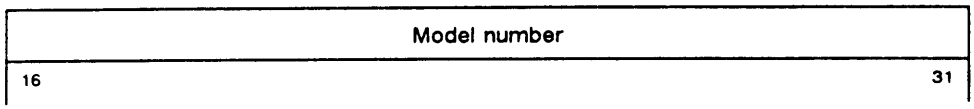


Bits	Name	Contents or Function
0-15	Model Number	Binary value of processor's allocated model number.
16-23	Microcode Revision	Current microcode revision.
24-31	Memory Size	Amount of physical memory available (measured in increments of 256-kilobyte modules with an origin of 0). Note that the actual memory size in bytes is equal to: (AC0[24-31] + 1) * 262144_{10} For example, 3_8 indicates 1 megabyte; 7_8 indicates 2 megabytes. NOTE: *Systems that contain 64 megabytes or more of physical memory return 377_8 to bits 24-31 of AC0.*

NCLID Instruction

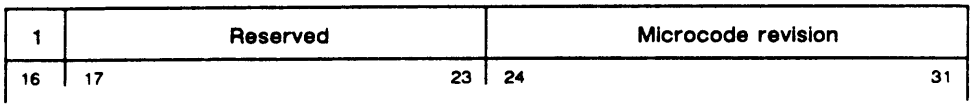
The NCLID instruction loads its result into the low-order 16 bits of accumulators AC0 through AC2. Bits 0 through 15 of each accumulator are undefined.

AC0



Bits	Name	Contents or Function
16-31	Model number	Binary value of processor's allocated model number.

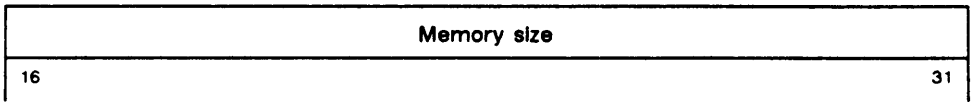
AC1



If AC1 contains 177777_8, load microcode.

Bits	Name	Contents or Function
16	1	Always set to 1.
17-23	Reserved	Reserved for future use and returned as 0s.
24-31	Microcode revision	Current microcode revision.

AC2



Bits	Name	Contents or Function
16-31	Memory size	Amount of physical memory available measured in 32-kilobyte modules with an origin of 0). Note that the actual memory size in bytes is equal to (AC0[16-31] + 1) * 32768_{10} For example, 32_8 indicates 1 megabyte; 64_8 indicates 2 megabytes. NOTE: *Systems that contain 2 gigabytes or more of physical memory return 177777_8 to bits 16-31 of AC0.*

B

Fault and Status Codes

This appendix presents general fault code and status information which applies to all ECLIPSE MV/Family computers.

Tables B-1 through B-6 explain the codes returned in AC1 for the following fault types:

- protection
- page
- stack
- universal power supply controller (UPSC)
- power supply controller (PSC)
- decimal/ASCII

Protection Faults

Table B-1 lists the meanings of codes returned in AC1 when a protection fault occurs.

Table B-1 *Protection fault codes*

Code (octal)	Meaning
0	Read violation
1	Write violation
2	Execute violation
3	Validity violation (SBR or PTE)
4	Inward address reference
5	Defer (indirect) violation
6	Illegal gate: out of bounds or gate bracket access violation
7	Outward call
10	Inward return
11	Privileged instruction violation
12	I/O protection violation
13	Reserved
14	Invalid micro-interrupt return block
15	Unimplemented instruction fault
16	Reserved
17	Invalid form ID (GIS)
20	Invalid attribute index (GIS)
21	Invalid CHARBLT source (GIS)

Page Faults

Table B-2 lists page fault codes that the processor stores in AC1.

Table B-2 *Page fault codes*

Code	Meaning
0	Reserved
1	Reserved
2	Pagetable page fault
3	Reserved
4	Normal object reference

Stack Faults

Table B-3 lists stack fault codes. The processor does not return an error code for a narrow stack fault.

Table B-3 *Stack fault codes*

Code (octal)	Meaning
000000	Overflow on every stack operation other than SAVE, WMSP, or segment crossing.
000001	Underflow or overflow would occur if the instruction were executed (pertains to the WMSP, WSSVR, WSSVS, WSAVR, and WSAVS instructions). (PC in return block refers to the instruction that caused the stack fault.)
000002	Too many arguments on a cross segment call.
000003	Stack underflow.
000004	Overflow due to a return block pushed as a result of a microinterrupt or fault.

UPSC Faults

Table B-4 lists the UPSC fault codes by fault category.

Table B-4 *UPSC fault codes*

Fault Code and Category (Bits 9-15) Octal	Hex	Meaning	Result
Category 0		**System off or no fault or UPSC fault**	
000	00	System off or no fault	—
170	78	Diagnostic mode timeout (computer failed to complete I/O)	Nonfatal
Category 1		**Environmental Fault**	
011	09	VNR undervoltage	Fatal > 300 msec
021	11	VNR overvoltage	Fatal
031	19	Power supply overtemperature	Fatal > 15 sec
041	21	Chassis overtemperature	Fatal > 15 sec
Category 2		**Fan failure**	
002	02	Blower or multiple fans	Fatal > 15 sec
012	0A	Fan 1	Fatal > 15 sec
022	12	Fan 2	Fatal > 15 sec
032	1A	Fan 3	Fatal > 15 sec
042	22	Fan 4	Fatal > 15 sec
052	2A	Fan 5	Fatal > 15 sec
062	32	Fan 6	Fatal > 15 sec
072	3A	Cannot set fan signals	Nonfatal

(continues)

Table B-4 *UPSC fault codes (concluded)*

Fault Code and Category (Bits 9-15) Octal	Hex	Meaning	Result
Category 3		**VNR fault**	
013	0B	Battery backup fault indicated	Nonfatal unless on batteries
Category 4		**Power supply fault (includes undervoltages)**	
004	04	+5V logic undervoltage	Fatal > 1 msec
014	0C	+5V logic current not sharing	Nonfatal
044	24	+5MEM undervoltage, PS1	Fatal > 1 msec
054	2C	+5MEM undervoltage, PS2	Fatal > 1 msec
064	34	+5MEM undervoltage, PS3	Fatal > 1 msec
074	3C	+12MEM or +12V undervoltage, PS1	Fatal > 1 msec
104	44	+12MEM or +12V undervoltage, PS2	Fatal > 1 msec
114	4C	+12MEM or +12V undervoltage, PS3	Fatal > 1 msec
124	54	-5MEM or -5V undervoltage, PS1	Fatal > 1 msec
134	5C	-5MEM or -5V undervoltage, PS2	Fatal > 1 msec
144	64	-5MEM or -5V undervoltage, PS3	Fatal > 1 msec
154	6C	Undervoltage PS1, voltage unknown	Fatal > 1 msec
164	74	Undervoltage PS2, voltage unknown	Fatal > 1 msec
174	7C	Undervoltage PS3, voltage unknown	Fatal > 1 msec
Category 5		**Overvoltage fault**	
005	05	+5V	Fatal
045	25	+5MEM, PS1	Fatal
055	2D	+5MEM, PS2	Fatal
065	35	+5MEM, PS3	Fatal
075	3D	+12V or +12MEM, PS1	Fatal
105	45	+12V or +12MEM, PS2	Fatal
115	4D	+12V or +12MEM, PS3	Fatal
125	55	-5V or -5MEM, PS1	Fatal
135	5D	-5V or -5MEM, PS2	Fatal
145	65	-5V or -5MEM, PS3	Fatal
155	6D	PS1, voltage unknown	Fatal
165	75	PS2, voltage unknown	Fatal
175	7D	PS3, voltage unknown	Fatal
Category 6		**Overcurrent fault**	
006	06	Reed switch sense low, +5V output, overcurrent to logic slots	Fatal > 1 msec
016	0E	+5V, PS1	Fatal > 1 msec
026	16	+5V, PS2	Fatal > 1 msec
036	1E	+5V, PS3	Fatal > 1 msec
046	26	+5MEM, PS1	Fatal > 1 msec
056	2E	+5MEM, PS2	Fatal > 1 msec
066	36	+5MEM, PS3	Fatal > 1 msec
076	3E	+12V or +12MEM, PS1	Fatal > 1 msec
106	46	+12V or +12MEM, PS2	Fatal > 1 msec
116	4E	+12V or +12MEM, PS3	Fatal > 1 msec
126	56	-5V or -5MEM, PS1	Fatal > 1 msec
136	5E	-5V or -5MEM, PS2	Fatal > 1 msec
146	66	-5V or -5MEM, PS3	Fatal > 1 msec
156	6E	PS1, voltage unknown	Fatal > 1 msec
166	76	PS2, voltage unknown	Fatal > 1 msec
176	7E	PS3, voltage unknown	Fatal > 1 msec
Category 7		**UPSC fault**	
007	07	Checksum error on PROM (blinks)	Fatal at powerup
177	7F	LED lamp test at powerup (short duration). If code is displayed for a long duration (> 5 seconds) without tangible powerup, this may indicate either insufficient ac voltage to complete a powerup, or a broken UPSC.	Nonfatal

NOTE: *Codes not listed are not used. The system shuts down when a fatal power system fault occurs.*

PSC Status and Faults

Table B-5 contains eight categories of code numbers for the power supply controller (PSC). The PSC status code numbers are listed under category 0, and the PSC fault code numbers are listed under categories 1 through 7. (Note that the least significant digit of each octal code number determines the code number's category.)

The leftmost column of Table B-5 lists the code number in hexadecimal, and the second column lists the code number in octal. When a fault occurs or status information is needed, the PSC displays the appropriate hexadecimal code number on the system's front panel and stores the equivalent octal code number in AC1.

Table B-5 *PSC status and fault codes*

Code Hex	Code Octal	Meaning	Result (see notes)
Category 0 - PSC status			
00	000	System up and OK	Status/Interrupt
08	010	Power system waiting for power on command from DRP before powering up	Status/interrupt
10	020	System powering up from power on command	Status
18	030	System up and OK; no heat and air testing	Status
20	040	Off command received	Status/Interrupt
28	050	Off switch detected	Status/Interrupt
30	060	Margining active	Status/Interrupt
38	070	BBU running	Status/Interrupt
40	100	ROM checksum OK	Status
48	110	System powering up from jumper	Status
50	120	VSR above low level during powerup	Status
58	130	VSR over or under shoot during powerup	Status
60	140	Checked VSR settled during powerup	Status
68	150	+18V AUX on but not checked	Status
70	160	All voltages on; no undervoltage checks	Status
80	200	BBU test running	Status/Interrupt
88	210	All voltages within tolerance	Status
Category 1 - Temperature/VSR voltage faults			
09	011	VSR undervoltage	Fatal fault/retry
11	021	VSR overvoltage	Fatal fault
21	041	Chassis over temperature	Warning (30 seconds)/fatal fault
29	051	Chassis under temperature	Status fault
31	061	Airflow sensor fault	Warning (1 second)/fatal fault
39	071	VSR undervoltage (no BBU installed)	Fatal fault
41	101	VSR undervoltage (BBU disabled by RNB command)	Fatal fault
Category 2 - Fan failure faults			
02	002	Blower failure	Warning (1 second)/fatal fault
Category 3 - VSR faults			
0B	013	Battery back-up fault indicated	Status fault
13	023	AC undervoltage detected (from VSR)	Status fault
1B	033	AC overvoltage detected (from VSR)	Status fault
23	043	VSR DC fault	Status fault
2B	053	BBU BATLOW (low charge)	Status fault
33	063	VSR fan fault	Status fault
3B	073	VSR over temperature	Warning (30 seconds)/fatal fault
43	103	BBU (battery out of charge)	Fatal fault/retry
4B	113	BBU test; pack 1 fault	Status fault
53	123	BBU test; pack 2 fault	Status fault
5B	133	BBU test; pack 1 and 2 fault	Status fault
63	143	BBU test; pack 3 fault	Status fault
6B	153	BBU test; pack 1 and 3 fault	Status fault
73	163	BBU test; pack 2 and 3 fault	Status fault
7B	173	BBU test; pack 1,2. and 3 fault	Status fault
83	203	BBU in high charge test delay	Status fault
8B	213	VSR not below 40V after 5 seconds	Status fault

(continues)

Table B-5 *PSC status and fault codes (concluded)*

Code Hex	Code Octal	Meaning	Result (see notes)
Category 4 - Power supply faults (includes undervoltage)			
04	004	+5V logic undervoltage	Fatal fault/retry
24	044	+5MEM undervoltage	Fatal fault/retry
3C	074	+12V undervoltage	Fatal fault/retry
54	124	-5V undervoltage	Fatal fault/retry
84	204	Power module +5V1	Status fault
8C	214	Power module +5V2	Status fault
94	224	Power module +5V3	Status fault
9C	234	Power module +5V4	Status fault
A4	244	Power module +5V5	Status fault
AC	254	Power module +5V6	Status fault
B4	264	Power module +5V7	Status fault
BC	274	Power module -5V7	Status fault
C4	304	Power module +5M4	Status fault
CC	314	Power module +5M3	Status fault
D4	324	Power module +12V1	Status fault
DC	334	Power module +12V2	Status fault
E4	344	Power module +12V5	Status fault
EC	354	Power module +12V7	Status fault
F4	364	Power module -5V6	Status fault
FC	374	Power boards in wrong slots	Fatal fault
Category 5 - Overvoltage faults			
05	005	+5V overvoltage	Fatal fault
25	045	+5MEM overvoltage	Fatal fault
3D	075	+12V overvoltage	Fatal fault
55	125	-5V overvoltage	Fatal fault
Category 6 - Overcurrent faults			
06	006	Reed switch sense low (+5V output overcurrent to logic slots)	Fatal fault/retry
Category 7 - Power supply controller faults			
07	007	Program checksum error	Fatal fault
0F	017	+12V AUX overvoltage	Fatal fault
17	027	-12V AUX overvoltage	Fatal fault
1F	037	+12V AUX undervoltage	Fatal fault/retry
27	047	-12V AUX undervoltage	Fatal fault/retry
2F	057	+18V AUX out of tolerance	Fatal fault
37	067	ASYNC/RNB fault	Status fault
3F	077	Checksum error on RNB	Status fault
47	107	Framing error on UART	Status fault/no interrupt
4F	117	Parity error on UART	Status fault/no interrupt
57	127	Overrun error on UART	Status fault/no interrupt
5F	137	DRP exhausted retries to PSC	DRP code
67	147	Break on UART	Status fault/no interrupt
6F	157	UART loopback fault	Status fault
77	167	UART interrupt fault	Status fault
7F	177	DRP retried RNB command	DRP code
8F	217	PWROK signal went away	DRP code
FF	377	PSC stuck; code did not run (but momentary LED lamp test at powerup is OK)	Fatal fault

NOTES: *Status fault — Interrupts the RNB but does not stop operation of the power system by itself.*

Status fault/no interrupt — Does not stop operation of the power system by itself and does not interrupt the RNB.

Fatal fault — Causes the system (except for PSC and DRP) to power down and stay down.

Fatal fault/retry — Causes the system to powerdown; PSC tries three times to power it up again before quitting.

Warning (xxxx)/fatal fault — Imminent system shutdown with prior warning interrupt to RNB (time until shutdown shown in parenthesis).

DRP code — Code generated by the DRP, not the PSC.

Decimal/ASCII Faults

Table B-6 describes decimal and ASCII fault codes. The first and second columns list codes that appear in AC1; the fourth and fifth columns list instructions and conditions that cause faults.

Table B-6 *Decimal/ASCII fault codes*

Code Returned in AC1		Return Block Type	Faulting Instruction	Condition
Narrow	Wide			
000000	100000	2	EDIT, WEDIT	An invalid digit or alphabetic character encountered during execution of one of the following subopcodes: DMVA, DMVF, DMVN, DMVO, DMVS.
000001	100001	1	LDIX, STIX	Invalid data type (6 or 7).
		3	WEDIT, WLDIX, WSTIX, WDMOV, WDDEC, WDINC, WDCMP, EDIT	Invalid data type (6 or 7).
000002	100002	2	EDIT, WEDIT	DMVA or DMVC subopcode with source data type 5; AC2 contains the data size and precision.
000003	100003	2	EDIT, WEDIT	An invalid opcode; AC2 contains the data size and precision.
000004	100004	1	LDI, STI, STIX, WLDI, WSTI, WSTIX, WLDIX	Number too large to convert to specified data type. |number|> (10^{16}) - 1 Number too large to convert to specified data type. Number > (10^{32}) - 1
000005	—	3	EDIT, LDI, LDIX, STI, STIX	Invalid microinterrupt return block. (Applies only to ECLIPSE interrupt-resumable instructions.)
000006	100006	1	WLSN, WLDI, LSN, LDI, LDIX, WLDIX	Sign code is invalid for this data type *
		3	EDIT, WEDIT,WDINC, WDMOV, WDCMP, WDDEC	
000007	100007	1	WLSN, WLDI. WLDIX, LSN, LDI, LDIX	Invalid digit *
		3	WDMOV, WDCMP, WDINC, WDDEC	

* *A value containing both an invalid sign and one or more invalid digits produces a decimal/ASCII fault which may indicate either type of error.*

End of Appendix

C

Reserved Memory Locations

The information in this appendix applies to all ECLIPSE MV/Family computers. Details on the ECLIPSE MV/8000 computer's C/350 MAP are given in the appropriate machine-specific supplement.

The processor reserves memory locations 0 through 47_8 of page zero (locations 0 through 377_8) of each segment for storing certain parameters and the starting addresses of the fault handlers. The processor interprets page zero locations for segment 0 differently from page zero locations for segments 1 through 7. For example, segment 0 contains pointers to privileged fault handlers, and segments 1 through 7 reserve these locations. Segment 0 locations are listed in Table C-1; segments 1 through 7 locations are listed in Table C-2.

Specified addresses for the fault handlers are not indirectable unless otherwise specified. Some pointers are 16 bits long; they can only refer to locations in the first 64 Kbytes of the segment containing the pointer. If the pointer is indirect, all pointers in the indirect chain will only refer to the first 64 Kbytes of the segment. With the address translator enabled, the processor interprets all locations in page zero as logical addresses. With the address translator disabled, only the contents of page zero in segment 0 are valid; the processor interprets page zero addresses as physical ones.

Table C-1 *Page zero location for segment 0*

Location (octal)	Name	Contents or Function
0	Interrupt level	Level of interrupt processing: 0 base-level processing nonzero intermediate-level processing
1	I/O handler	Address of I/O interrupt handler (indirectable).
2-3	I/O return address	Address of I/O interrupt return. Location 2 contains the high-order bits; location 3 contains the low-order bits.
4	Vector stack pointer	Low-order 16 bits of vector stack pointer, base, and frame pointer (high-order bits = 0).
5	Current 16-bit narrow mask	Current 16-bit narrow interrupt priority mask.
6	Vector stack limit	Low-order 16 bits of vector stack limit.
7	Vector stack fault address	Address of vector stack fault handler (indirectable).
10-11	Breakpoint address	Address of breakpoint handler (indirectable).
12-13	WXOP origin address	Address of beginning of extended operations table — see the WXOP instruction description.
14	Wide stack fault handler	Address of wide stack fault address handler (indirectable).
15-17	Reserved	Reserved.
20-21	WFP	Wide frame pointer.
22-23	WSP	Wide stack pointer.
24-25	WSL	Wide stack limit.
26-27	WSB	Wide stack base.
30-31	Page fault handler	Address of wide page fault handler.
32-33	Context block pointer	Address of base of context block save area.
34-35	WGP	Gate pointer; address of the gate array.
36	Protection fault handler address	Address of protection fault handler (indirectable).
37	Fixed-point fault handler address	Address of fixed-point fault handler (indirectable).
40	Stack pointer	Address of top of 16-bit narrow stack.
41	Frame pointer	Address of start of current narrow frame minus 1.
42	Stack limit	Address of last normally usable location in narrow stack.
43	Narrow stack fault handler	Address of ECLIPSE 16-bit narrow stack fault handler (indirectable).
44	XOP0 origin address	Address of beginning of narrow extended operations table. See the XOP0 instruction description.
45	Floating-point fault address	Address of floating-point fault handler (indirectable).
46	Decimal/ASCII fault handler	Address of decimal/ASCII fault handler (indirectable).
47	DERR error handler	Address of DERR instruction error/trap handler. See the DERR instruction description.

Table C-2 *Page zero locations for segments 1 through 7*

Location (octal)	Name	Contents or Function
0-7	Reserved	Reserved.
10-11	Breakpoint address	Address of breakpoint handler (indirectable).
12-13	WXOP origin address	Address of beginning of extended operations table — see the WXOP instruction description.
14	Wide stack fault handler	Address of wide stack fault address handler (indirectable).
15-17	Reserved	Reserved.
20-21	WFP	Wide frame pointer.
22-23	WSP	Wide stack pointer.
24-25	WSL	Wide stack limit.
26-27	WSB	Wide stack base.
30-33	Reserved	Reserved.
34-35	WGP	Gate pointer; address of the gate array.
36	Reserved	Reserved (refer to the "User Protection Fault Handler" section in the "Memory and System Management" chapter).
37	Fixed-point fault handler address	Address of fixed-point fault handler (indirectable).
40	Stack pointer	Address of top of 16-bit narrow stack.
41	Frame pointer	Address of start of current narrow frame minus 1.
42	Stack limit	Address of last normally usable location in narrow stack.
43	Narrow stack fault handler	Address of ECLIPSE 16-bit narrow stack fault handler (indirectable).
44	XOP0 origin address	Address of beginning of narrow extended operations table. See the XOP0 instruction description.
45	Floating-point fault address	Address of floating-point fault handler (indirectable).
46	Decimal/ASCII fault handler	Address of decimal/ASCII fault handler (indirectable).
47	DERR error handler	Address of DERR instruction error/trap handler. See the DERR instruction description.

End of Appendix

D

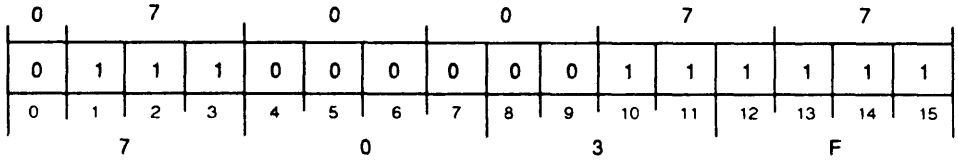
Load Control Store Instruction

This appendix describes the Load Control Store instruction and its associated microcode file.

CAUTION: *The Load Control Store instruction changes various parts of the machine's internal state. This instruction is intended for diagnostic and special system applications.*

Load Control Store LCS

LCS
(error return)
(normal return)

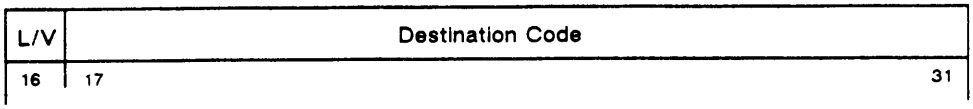


The Load Control Store instruction loads and verifies the soft internal states of the machine (such as, the microstore, decode rams, and scratchpad). In conjunction with bits 16 through 31 of the four accumulators, the **LCS** instruction loads and verifies, or verifies only, using the contents of a microcode file. The assembler recognizes **LCS** to be equivalent to **NIO,CPU**.

The **LCS** instruction loads a certain number of words per instruction (generally 16K words). Depending on the machine it may be necessary to issue the instruction many times. This instruction is noninterruptible. Note that some ECLIPSE MV/Family systems ensure that microcode blocks will be no greater than 1K words in length.

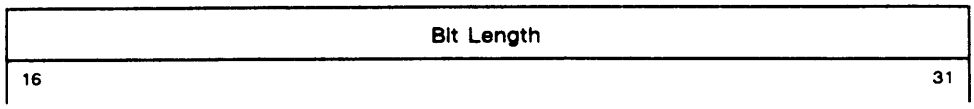
The formats of the accumulators are diagrammed and described next (bits 0 through 15 of each accumulator are undefined and unused).

AC0



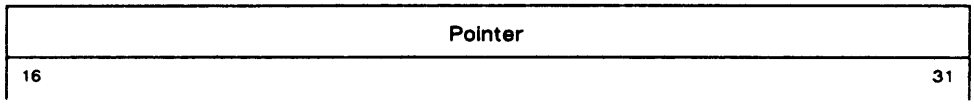
Bits	Name	Contents or Function
16	L/V	Load/verify option 0 Load and verify 1 Verify only
17-31	Destination Code	Indicates where data is to be loaded

AC1



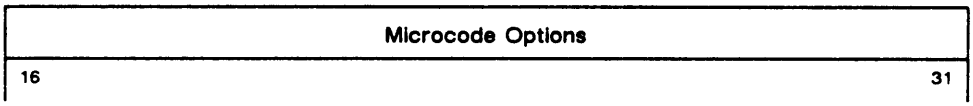
Bits	Name	Contents or Function
16-31	Bit Length	Bit length of code data

AC2



Bits	Name	Contents or Function
16-31	Pointer	Pointer to first block of data (indirectable)

AC3 (Optional)



NOTE: *ECLIPSE MV/Family systems which do not support microcode options ignore the contents of AC3.*

Bits	Name	Contents or Function		
16-31	Microcode Options	Specifies which microcode options to load: If bits 24-31 are 0, perform a normal load (options will be defaulted).		
		Bit	**Mnemonic**	**Option (if set to 1)**
		16-29	Reserved	Unused
		30	ARCH	Load architectural clock microcode. (This bit should be used on processors that support either architectural clocks or the PIT/RTC combination.)
		31	FPU	Do NOT load microcode support for a hardware floating-point unit (FPU). This setting should be used in the case where there is an FPU, but you do not want to use it, such as during a diagnostic function.

Follow these steps to load and verify the microcode:

1. Parse microcode file blocks.
   a. Load Code blocks.
   b. Fill Fill blocks.
   c. Ignore Revision blocks.
   d. Print Comment blocks.
2. Repeat the sequence listed above until an End block is encountered. (The LCS instruction is complete when an End block is encountered.)
3. Verify Code blocks that were loaded in step 1; ignore Fill, Comment, and Revision blocks. To only verify, perform step 3.

Microcode File and Block Format

The microcode file format contains data used in various parts of the machine's state. Figure D-1 shows the general format for each microcode file. The microcode format is block-oriented, that is, arranged in packets or blocks. Each block contains a description of its size and the type of data it contains. As Figure D-1 shows

- Each microcode file must begin with a Title block and conclude with an End block. Optionally, Revision and/or Comment blocks may precede the Title block and Comment blocks may follow the End block.
- Fill and Code blocks must be placed between the Title/End block pair.
- The Revision block, if any, precedes the first Title block.
- Comment blocks can appear anywhere within the microcode file.

Table D-1 summarizes the contents and functions of each block type.

Table D-1 *Microcode file format blocks*

Block Type	Contents or Function
Title	Data about code word's bit length, and destination code. The program issuing the LCS instruction places this data in AC0 and AC1.
End	Data needed to continue or terminate LCS instruction.
Code	Code words and starting location for storing each code word. Code blocks must appear between Title/End block pair.
Fill	Code words used as background filler and to specify locations to receive this data. Fill blocks must appear between Title/End block pair.
Comment	Data that can be output to system console or ignored. Comment blocks can appear anywhere within the microcode file structure; placement determines where data is output. If Comment block is internal (appears within Title/End block pair), data is output to system console. If Comment block is external (appears outside Title/End block pair), program issuing LCS decides whether to output or ignore the data.
Revision	Target CPU model number. Major and minor revision numbers for microcode. Revision blocks are optional, but if used, they should appear as first block of the microcode file. Program issuing LCS instruction determines whether Revision blocks are ignored or output to system console.

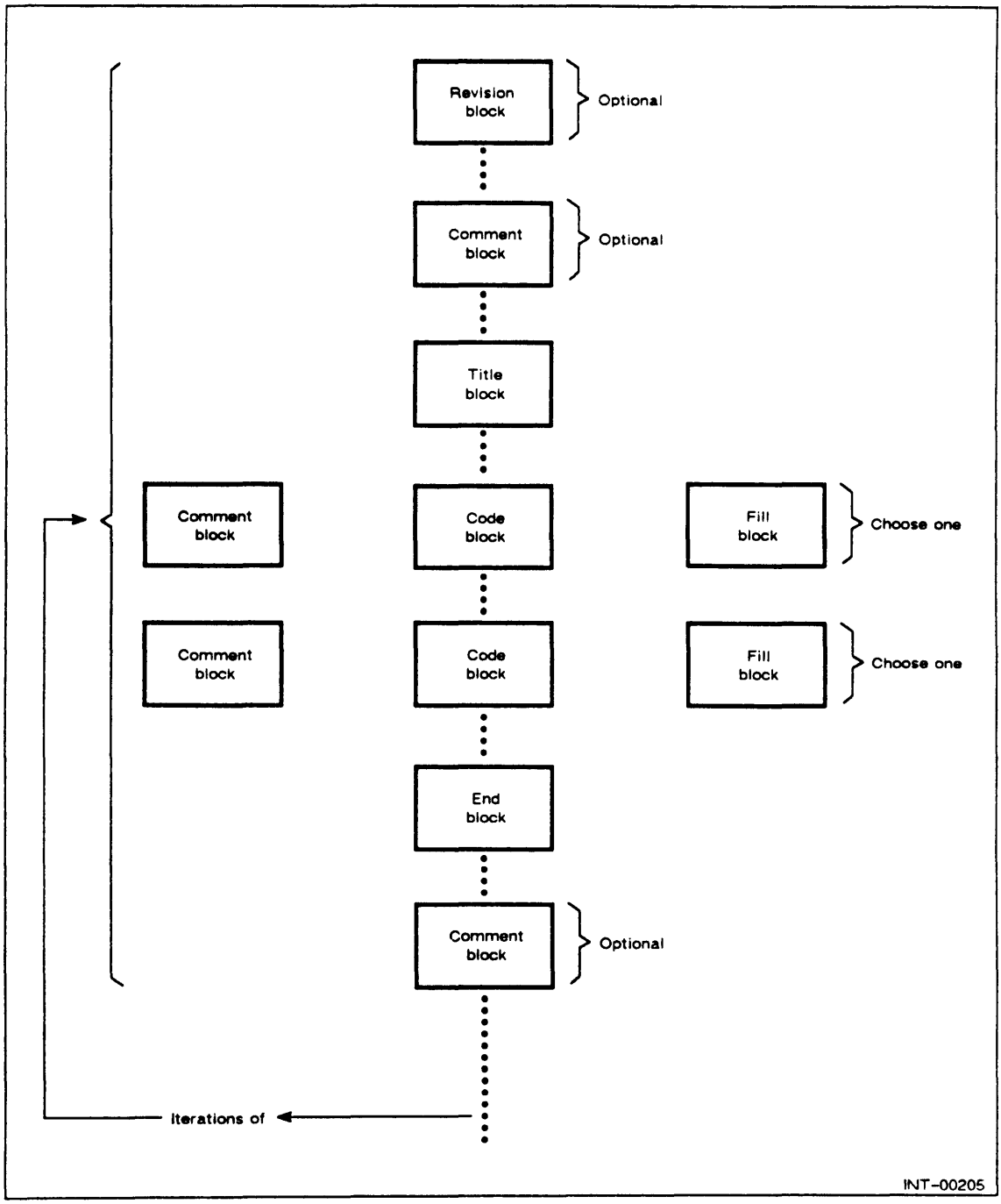


Figure D–1 *Microcode file format*

LCS Implementation

The following describes the effect of the LCS instruction on Code, Comment, Fill, and End blocks. The program issuing the LCS instruction must parse and organize the information from the Title and Revision blocks and from any external Comment blocks.

The LCS instruction

- Recognizes Code blocks and loads the data into the proper destination addresses;
- Recognizes internal Comment blocks and prints the text string on the system console;
- Recognizes Fill blocks and performs a fill operation of the proper destination;
- Recognizes End blocks and performs a verify operation upon the previously loaded data;
- Recognizes an error condition (see the section, "Error Returns,") and returns the proper error code to AC0.

Microcode Blocks

Figure D-2 shows the general format of each microcode block; Table D-2 summarizes the contents of the format. Tables D-3, D-4, and D-6 through D-9 explain the individual blocks in more detail.

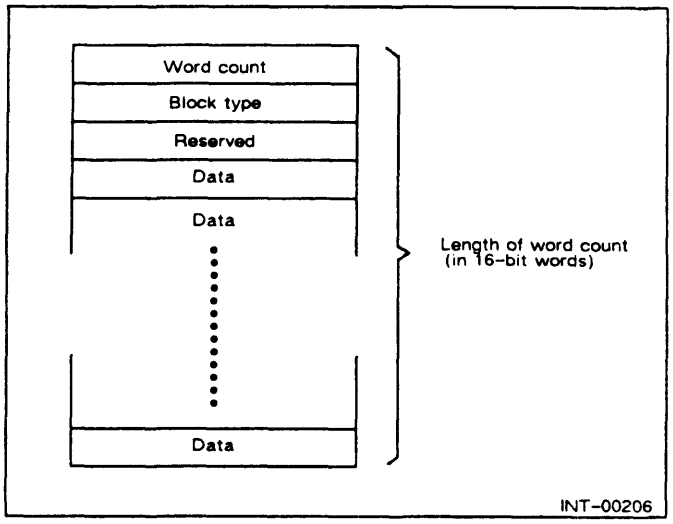


Figure D-2 *Microcode block format*

Table D-2 *Words used in the microcode block format*

Word	Name	Contents or Function
1	Word count	Number of 16-bit words in microcode block.
2	Block type	Type of data contained in block.
3	Reserved	Reserved for future use.
4 + n	Data	Data pertaining to block type.

Table D-3 *Title block format*

Word	Contents or Function
Word Count	7
Block Type	0
Reserved	Reserved for future use.
Data word 1	Code word's bit length.
Data word 2	Reserved for future use.
Data word 3	Reserved for future use.
Data word 4	Destination code indicating where data is to be loaded. (The processor accepts only positive nonzero 16-bit integers in the range 1 through 77777_8.)

The data from the first Title block is used by the program issuing the **LCS** instruction. For example:

Data word 4 (destination) is placed into ACO.
Data word 1 (bit length of code word) is placed into AC1.

Table D-4 *End block format*

Word	Contents or Function
Word Count	5
Block Type	1
Reserved	Reserved for future use.
Data word 1	Control word. **Bits Meaning** 0-12 Reserved. 13 Destination completion indicator. 0 More code of this destination may follow. 1 No more code. 14 Switch from PROM to RAM Control Store. 0 Stay in current mode. 1 Switch to RAM. 15 Start designator. 0 Start host (and continue SCP). 1 Start SCP; data word 2 must be an address (see Table D-5).
Data word 2	Address to be started (Table D-5). If this is -1 (177777_8), continue execution with LCS normal/error return.

Table D-5 summarizes the combined actions of Data word 1 (bit 15) and Data word 2 of an End block.

Table D-5 *Combined action of End block data words 1 and 2*

Data Word 2 Contains	Data Word 1 (Bit 15) Contains 0	1
-1	Continue host at LCS normal/error return.	Illegal
Address	Start host at this address; continue SCP.	Start SCP at this address; host remains halted.

Table D-6 *Code block format*

Word	Contents or Function
Word Count	Variable
Block Type	2
Reserved	Reserved for future use.
Data word 1	Location for storing first code word in this block.
Data word 2 to n+1(1)	First code word of block.
Data word n+2(1) to 2n+1	Code word for next sequential address.
Data word 2n+2(1) to 3n+1	Code word for next sequential address until end of block.

NOTE: *Code data is in a word-aligned format: n is the number of 16-bit words that contain one code word [n = (word-bit-length + 15)/16].*

The Fill block enables background filling of certain destinations within the machine. For example, if an uninitialized location is erroneously entered during execution, it is possible to zero-fill the control store to induce parity errors.

The function of the Fill block can also be accomplished with Code blocks if they contain the appropriate data.

Table D-7 *Fill block format*

Word	Contents or Function
Word Count	n+5 [n=(word-bit-length + 15)/16]
Block Type	3
Reserved	Reserved for future use.
Data word 1	Starting location for storing code word.
Data word 2	Ending location for storing code word.
Data word 3 to n+2	Code word to be used as background filler.

Table D-8 *Comment block format*

Word	Contents or Function
Word Count	Variable
Block Type	4
Reserved	Reserved for future use.
Data word 1	Length of ASCII string, not counting terminating null[s]. Odd string length indicates one terminating null; even string length indicates two terminating nulls.
Data word 2 to x+2	ASCII string (packed right to left) terminated by null. [x = (String length + 1)/2]

Table D-9 *Revision block format*

Word	Contents or Function
Word Count	6
Block Type	5
Reserved	Reserved for future use.
Data word 1	Target CPU model number.
Data word 2	Microcode major revision number.
Data word 3	Microcode minor revision number.

Error Return

When the processor encounters an error, three accumulators (AC0, AC1, AC2) contain information indicating the source of the problem. Bits 0 through 15 of each accumulator are undefined and unused; bits 16 through 31 contain the following:

- AC0 contains the code indicating the type of error (refer to Table D-10 for an explanation of the error codes returned).
- AC1 contains information dependent upon the error code returned in AC0 (refer to Table D-10 for the contents of AC1).
- AC2 contains a 16-bit pointer to the erring block. If initial information in AC0 or AC1 caused error, then AC2 is unchanged.

The error codes returned to AC0 are listed in Table D-10.

Table D-10 *Error codes returned to AC0*

AC0 Code	Meaning	Definition (AC1 Contents) [Possible Cause]
1	Verify error	Data not received properly by destination. (AC1 will contain the code word location in error.) [Hardware problem]
2	Illegal code word length	Disagreement between code word bit length and length of code data specified by destination word in same Title block. (AC1 unchanged.) [Attempt to load wrong model microcode.]
3	Unexpected block type	Block type other than Code, Fill, End, Revision or Comment. (AC1 unchanged.) [Missing block or out of sequence.] NOTE: *If any Title blocks are encountered between the Title/End block pair, the error "unexpected block type" will occur.*
4	Illegal block length	Block length error. (AC1 unchanged.) [Block length less than four; code block did not contain an integral number of code words — such as, if code word bit length is 80, then length of all code blocks must be 4+N*(80+15)/16. N = number of code words per code block. 16 = number of bits per word. 4 = number of words at the beginning of each code block. In this example, all code blocks must be of length 4+5*N].
5	Unknown destination	Unknown location for loading of code word. (AC1 unchanged.) [Attempt to load incorrect model machine microcode file.]
6	Illegal option	Microcode option specified in AC3 is undefined. (AC1 will contain error code 6.) [Attempt to use microcode option presently undefined.]

Kernel Functions

The kernel is the minimum set of microcode necessary for the machine to function properly. The processor can read target microcode from an I/O device (using the kernel I/O instructions) and then load this microcode into the control store with the kernel instruction set (including the LCS instruction).

Since the LCS instruction must return to the host after completion, the kernel instruction set must exist and be working after each execution of the LCS instruction.

The amount of data that can be loaded with a single LCS instruction is processor-specific (generally, 16K words). Therefore, several iterations of accessing the I/O device and executing the LCS instruction may be necessary to completely change the machine from the kernel to the target.

End of Appendix

Glossary

ADDRESS TRANSLATOR. Mechanism used in demand paging to translate the specified logical address to its physical equivalent.

ATOMIC INSTRUCTIONS. Any instruction which will perform its entire specified operation without being interrupted (such as a read-modify-write operation).

ATTRIBUTE BLOCK. Graphics block consisting of unsigned 32-bit integers that is created when a form descriptor is created. Initially, the attribute block is filled with a set of default values which can be examined with the graphics instruction, Read Attribute, and modified with the Write Attribute instruction. (*See* FORM DESCRIPTOR.)

BIT IDENTIFIER. With the word pointer, forms a bit pointer. The bit identifier is located in the least significant bits of the ACD accumulator. (*See* BIT POINTER, WORD POINTER.)

BIT POINTER. Formed from the contents of two accumulators, the bit pointer contains a word pointer and a bit identifier. The ECLIPSE memory reference instructions (**BTO**, **BTZ**, **SNB**, **SZB**, and **SZBO**) use the bit pointer to refer to a bit.

BYTE. Eight consecutive bits.

BYTE POINTER. Formed from the contents of an accumulator or from the contents of the index field and the 16- or 32-bit displacement.

BOUNDING RECTANGLE. Specifies the usable range of values for X and Y coordinates in a form. (*See* FORM.)

COMBINATION RULE. Specifies how pixels are to be combined for any instruction that writes to a form.

CURRENT RING OF EXECUTION (CRE). The segment in which the program is presently executing.

CURSOR. A pattern that is drawn on the bitmap screen to represent the position of a pointing device.

CURSOR DESCRIPTOR. Permits the cursor to be managed by the operating system, even though it is drawn over the user's picture in the form.

DATA CHANNEL MAPS. A set of address translation registers that the user-specified map defines for the memory references of a data channel used by a particular device. These maps translate logical addresses to physical addresses when data channel devices access memory.

DATA ELEMENT. An entry in a queue.

DEMAND PAGING. A page-swapping mechanism controlled by the page fault handler which moves pages referred to by an instruction or routine from secondary storage (such as, a disk) to main physical memory as they are needed.

DEQUEUEING. The process of removing a data element from a queue.

DOUBLEWORD. Two consecutive words of memory (4 bytes or 64 bits).

ENQUEUEING. The process of adding a data element to a queue.

FIXED-POINT COMPUTATION. Fixed-point binary arithmetic operations on signed and unsigned 16- and 32-bit numbers.

FLOATING-POINT COMPUTATION. Floating-point binary arithmetic operations on signed, single-precision (32-bit) and double-precision (64-bit) numbers.

FONT. A set of shapes for letters, numbers, and punctuation marks, also known as a character set.

FORM. The basic unit of pixel space on which a picture is drwan. All GIS operations are performed on the form.

FORM DESCRIPTOR. Describes the form and points to related databases such as cursor descriptors and attributes. The form descriptor block is a doubleword table of 32-bit integers.

FORM MASK. Used to implement palette sharing, a technique that helps programs to share a display without destroying each other's data. The form mask in the form descriptor is a value that specifies which bits in a pixel can be accessed by drawing operations.

GATE ARRAY. A series of locations specifying entry points (or gates) to a segment. The processor accesses a gate array through an indirect pointer in page zero of the destination segment.

GBYTE. Gigabyte (2^{30} bytes).

GUARD DIGIT. One hex digit (four bits) that initially contains zero. To increase the accuracy of floating-point arithmetic, the processor appends one or two guard digits to the operands of both mantissas before performing arithmetic calculation.

HEAD. The beginning or first element of a queue.

IMPURE ZERO. *See* NORMALIZED FORMAT and TRUE ZERO.

INSTRUCTION CACHE. The major component of the instruction processor. The instruction cache provides input to the instruction decoder.

INSTRUCTION PROCESSOR. Decodes instructions for execution.

INSTRUCTION SET. The instruction set has two subsets: the 16-bit instruction set and the 32-bit instruction set. The 16-bit instructions are supported by the ECLIPSE MV/Family 32-bit processors and also supported by ECLIPSE 16-bit computers (such as the ECLIPSE C/350 computer). The 16-bit instructions are also referred to as ECLIPSE 16-bit instructions; the 32-bit instructions are also called ECLIPSE MV/Family instructions.

KERNEL. The minimum set of microcode necessary for the machine to function properly.

LINK. An address used to link together the data elements in a queue. Two links are required: the forward link contains the effective word address of the following data elements in a queue; the backward link contains the address of the preceding data elements in a queue.

LOGICAL ADDRESS. Specifies a segment number and a logical address. The computer uses 31-bit word addresses and 32-bit byte addresses, which can refer to all 4 Gbytes of the logical address space.

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

Mantissa times 16**y (*y* is the true value of the exponent).

MANTISSA. A fraction representing part of the value of a floating-point number. *See* MAGNITUDE.

MAP. For *memory allocation* and *protection* unit. The MAP's primary function is address translation. The MAP divides each user's primary logical address space into pages and associates each logical page with a physical page. By doing so, the MAP allows several users access to the same section of physical memory.

MEMORY REFERENCE INSTRUCTION. An instruction that accesses memory for data or for another instruction. A memory reference instruction contains the information for determining the effective address of an operand or determining the effective address of the next nonsequential instruction.

NARROW STACK. A contiguous set of single words that supports ECLIPSE 16-bit program development and upward program compatibility.

NONINTERRUPTIBLE INSTRUCTIONS. Any instruction which executes in such a relatively short time that it will not be interrupted.

NO-OP INSTRUCTION. An instruction that performs no work and does not afect any data (including the resolution of indirect addresses).

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

NOVA-TYPE INSTRUCTIONS. Any instruction originally part of the 16-bit NOVA® instruction set.

OPERATION MASK. Specifies which bits in a pixel can be modified by drawing operations. A zero means do nothing with this bit; a one means operate on this bit using the combination rule.

PAGE. A 2-Kbyte block of contiguous logical addresses in virtual memory.

PAGE ADDRESS. A page number with 10 zeros following it; the logical address denoting the first logical address in a page.

PAGE FAULT. The condition caused by referring to a page that is not resident in main physical memory.

PAGE FRAME. A 2-Kbyte block of contiguous physical memory locations (addresses).

PAGE PROTOCOL. Determines the validity of the reference made when a memory reference instruction addresses the current segment. The page protocols are valid page, read access, write access, and execute access.

PAGETABLE. One or more pagetable pages that completely specify the logical to physical address translation for a segment. A pagetable may be one- or two-level. A one-level pagetable consists of a single pagetable page, and a two-level pagetable consists of one pagetable page whose PTEs point to pagetable pages.

PAGETABLE ENTRY (PTE). A doubleword used by the processor in translating logical addresses to physical addresses. A PTE contains a page number and bits defining the page protocol.

PAGETABLE PAGE. A page consisting of 512 PTEs.

PAGE ZERO. Denotes the first 2K bytes of a segment, but sometimes denotes the first 256 words of a segment (locations $0-377_8$).

PIXEL. An addressable picture element.

PROTECTION. The system uses a hardware-implemented hierarchical protection system that allows programs different levels of privileges. Each segment has a different level, or ring, of protection associated with it. Each ring governs its associated segment with a different degree of privilege. Ring 0 has the highest degree of protection; thus, the kernel of the operating system resides in segment 0.

QUEUE. A variable-length list of linked entries typically used by an operating system to track the processes it must perform, such as printing files on a line printer.

QUEUE DESCRIPTOR. Two 32-bit words indicating the current tail and head of the queue.

QUEUE MANAGEMENT. The process of inserting, deleting, and searching for elements in a queue.

RECTANGLE DESCRIPTOR. Describes one of the set of rectangles that make up a form. (*See* FORM.)

RECTANGLE LIST. A structure used to keep track of which bitmap is used for various parts of the form. The list consists of one or more rectangle descriptors.

RESTARTABLE INSTRUCTIONS. Any instruction which, after being interrupted, restarts its specified operation using updated values.

RESUMABLE INSTRUCTIONS. Any instruction which, after being interrupted, continues its specified operation at the same point where it was interrupted using its original values.

RING. Protection mechanism that safeguards the contents of a segment.

SEGMENT. A portion of memory that contains data and programs and can be logically addressed. There are eight segments; each is a complete address space of 512 Mbytes.

SEGMENT BASE REGISTER (SBR). A processor register used in translating logical to physical addresses. There is one SBR for each of the eight segments. An SBR contains the address of the pagetable for the segment and various protection and privilege bits for the segment.

SNIFFING. A process that checks for memory errors, verifies all memory locations, and corrects all single-bit errors. Sniffing prevents single-bit errors from collecting in unused areas of memory and also prevents intermittent single-bit errors from developing into multiple-bit errors.

STACK. A series of consecutive locations in memory. A program uses the stack to pass arguments between subroutine calls and to save the program's state when the processor services a fault. Stack instructions add items to the stack in sequential order (push) and retrieve them from the stack in reverse order (pop). Although a program can access many stack areas, it can use only one area at a time.

SYSTEM CACHE. A look-ahead/look-behind buffer for the system, the system cache reduces the time the CPU and the I/O systems need to access memory.

SUPERVISOR. The part of the operating system that controls system functions, for instance, selecting unused pages for a new user and prioritizing users' requests.

TAIL. The end or last element of a queue.

TRUE 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.

UNDEFINED. Refers to a state that may or may not have been altered by the execution of an instruction. (Identical initial conditions may not always produce the same results when executing on different processors.)

USER MAPS. A set of address translation functions which the MAP defines for a particular user. When the processor encounters a memory reference instruction in a user's program, the user maps translates logical addresses to physical address. *(See* MAP.)

VIRTUAL MEMORY. A 4-Gbyte portion of memory that consists of eight segments and eight rings and facilitates memory management.

WIDE FRAME POINTER (WFP). Defines a reference point in the wide stack. The wide frame pointer is unchanged by push and pop operations (unless specified otherwise).

WIDE INSTRUCTIONS. Instructions which manipulate data with lengths of 8, 16, or 32 bits. The mnemonics of the instructions indicate the size of the data fields referenced. A mnemonic preceded by the letter N manipulates 16-bit (narrow) data; a mnemonic preceded by the letter W manipulates 32-bit (wide) data. No special prefix precedes those mnemonics that manipulate 8-bit data.

Other mnemonic prefixes indicate the addressing range of the instruction. X indicates that the instruction has a 512-Mbyte (extended) offset addressing range: L indicates a 4-Gbyte (long) addressing range.

WIDE STACK. A contiguous set of doublewords that supports 32-bit processor programs.

WIDE STACK BASE (WSB). Defines the lower limit of the wide stack.

WIDE STACK LIMIT (WSL). Defines the upper limit of the wide stack.

WIDE STACK POINTER (WSP). Addresses the top location of the wide stack, which is either the location of the last word placed onto the stack or the next word available from the stack.

WORD. Two bytes or 16 bits of memory. The basic unit of addressing in the ECLIPSE MV/Family instructions is one or more words in length, and most instructions manipulate one or more words of data.

WORD ADDRESS. Identifies a 16-bit word in the memory segment.

WORD POINTER. Consists of an effective address (in the source accumulator) and a word offset (in the destination accumulator).

End of Glossary

Index

Within the index, the page number refers to the first page for an entry (even if the subject spans multiple pages). Instruction mnemonics are printed in boldface type (such as **INTEN**); instruction names are printed with initial capital letters (such as I/O Interrupt Enable).

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

TIPS ORDERING PROCEDURES

TO ORDER

1. An order can be placed with the TIPS group in two ways:
   a) MAIL ORDER - Use the order form on the opposite page and fill in all requested information. Be sure to include shipping charges and local sales tax. If applicable, write in your tax exempt number in the space provided on the order form.

      Send your order form with payment to:
      Data General Corporation
      ATTN: Educational Services/TIPS G155
      4400 Computer Drive
      Westboro, MA 01581-9973

   b) TELEPHONE - Call TIPS at (508) 870-1600 for all orders that will be charged by credit card or paid for by purchase orders over $50.00. Operators are available from 8:30 AM to 5:00 PM EST.

METHOD OF PAYMENT

2. As a customer, you have several payment options:
   a) Purchase Order - Minimum of $50. If ordering by mail, a hard copy of the purchase order must accompany order.
   b) Check or Money Order - Make payable to Data General Corporation.
   c) Credit Card - A minimum order of $20 is required for Mastercard or Visa orders.

SHIPPING

3. To determine the charge for UPS shipping and handling, check the total quantity of units in your order and refer to the following chart:

Total Quantity	Shipping & Handling Charge
1-4 Units	$5.00
5-10 Units	$8.00
11-40 Units	$10.00
41-200 Units	$30.00
Over 200 Units	$100.00

   If overnight or second day shipment is desired, this information should be indicated on the order form. A separate charge will be determined at time of shipment and added to your bill.

VOLUME DISCOUNTS

4. The TIPS discount schedule is based upon the total value of the order.

Order Amount	Discount
$1-$149.99	0%
$150-$499.99	10%
Over $500	20%

TERMS AND CONDITIONS

5. Read the TIPS terms and conditions on the reverse side of the order form carefully. These must be adhered to at all times.

DELIVERY

6. Allow at least two weeks for delivery.

RETURNS

7. Items ordered through the TIPS catalog may not be returned for credit.
8. Order discrepancies must be reported within 15 days of shipment date. Contact your TIPS Administrator at (508) 870-1600 to notify the TIPS department of any problems.

INTERNATIONAL ORDERS

9. Customers outside of the United States must obtain documentation from their local Data General Subsidiary or Representative. Any TIPS orders received by Data General U.S. Headquarters will be forwarded to the appropriate DG Subsidiary or Representative for processing.

TIPS ORDER FORM

Mail To: Data General Corporation
Attn: Educational Services/TIPS G155
4400 Computer Drive
Westboro, MA 01581 - 9973

BILL TO:	SHIP TO: (No P.O. Boxes - Complete Only If Different Address)
COMPANY NAME _____	COMPANY NAME _____
ATTN: _____	ATTN: _____
ADDRESS _____	ADDRESS (NO PO BOXES) _____
CITY _____	CITY _____
STATE _____ ZIP _____	STATE _____ ZIP _____

Priority Code _____ (See label on back of catalog)

_____ _____ _____ _____ _____
Authorized Signature of Buyer (Agrees to terms & conditions on reverse side) | Title | Date | Phone (Area Code) | Ext.

ORDER #	QTY	DESCRIPTION	UNIT PRICE	TOTAL PRICE

A SHIPPING & HANDLING

☐ UPS	ADD
1-4 Items	$ 5.00
5-10 Items	$ 8.00
11-40 Items	$ 10.00
41-200 Items	$ 30.00
200+ Items	$100.00

Check for faster delivery

Additional charge to be determined at time of shipment and added to your bill.

☐ UPS Blue Label (2 day shipping)
☐ Red Label (overnight shipping)

B VOLUME DISCOUNTS

Order Amount	Save
$0 - $149.99	0%
$150 - $499.99	10%
Over $500.00	20%

Tax Exempt # or Sales Tax (if applicable) _____

ORDER TOTAL	
Less Discount See B	-
SUB TOTAL	
Your local* sales tax	+
Shipping and handling - See A	+
TOTAL - See C	

C PAYMENT METHOD

☐ Purchase Order Attached ($50 minimum)
P.O. number is _____. (Include hardcopy P.O.)
☐ Check or Money Order Enclosed
☐ Visa ☐ MasterCard ($20 minimum on credit cards)

Account Number _____ Expiration Date _____

_____
Authorized Signature
(Credit card orders without signature and expiration date cannot be processed.)

THANK YOU FOR YOUR ORDER

PRICES SUBJECT TO CHANGE WITHOUT PRIOR NOTICE.
PLEASE ALLOW 2 WEEKS FOR DELIVERY.
NO REFUNDS NO RETURNS.

* Data General is required by law to collect applicable sales or use tax on all purchases shipped to states where DG maintains a place of business, which covers all 50 states. Please include your local taxes when determining the total value of your order. If you are uncertain about the correct tax amount, please call 508-870-1600.

Form 702
Rev. 8/87

DATA GENERAL CORPORATION
TECHNICAL INFORMATION AND PUBLICATIONS SERVICE
TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form. These terms and conditions apply to all orders, telephone, telex, or mail. By accepting these products the Customer accepts and agrees to be bound by these terms and conditions.

1. CUSTOMER CERTIFICATION
Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

2. TAXES
Customer shall be responsible for all taxes, including taxes paid or payable by DGC for products or services supplied under this Agreement, exclusive of taxes based on DGC's net income, unless Customer provides written proof of exemption.

3. DATA AND PROPRIETARY RIGHTS
Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

4. LIMITED MEDIA WARRANTY
DGC warrants the CLI Macros media, provided by DGC to the Customer under this Agreement, against physical defects for a period of ninety (90) days from the date of shipment by DGC. DGC will replace defective media at no charge to you, provided it is returned postage prepaid to DGC within the ninety (90) day warranty period. This shall be your exclusive remedy and DGC's sole obligation and liability for defective media. This limited media warranty does not apply if the media has been damaged by accident, abuse or misuse.

5. DISCLAIMER OF WARRANTY
EXCEPT FOR THE LIMITED MEDIA WARRANTY NOTED ABOVE, DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS, CLI MACROS OR MATERIALS SUPPLIED HEREUNDER.

6. LIMITATION OF LIABILITY
A. CUSTOMER AGREES THAT DGC'S LIABILITY, IF ANY, FOR DAMAGES, INCLUDING BUT NOT LIMITED TO LIABILITY ARISING OUT OF CONTRACT, NEGLIGENCE, STRICT LIABILITY IN TORT OR WARRANTY SHALL NOT EXCEED THE CHARGES PAID BY CUSTOMER FOR THE PARTICULAR PUBLICATION OR CLI MACRO INVOLVED. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO CLAIMS FOR PERSONAL INJURY CAUSED SOLELY BY DGC'S NEGLIGENCE. OTHER THAN THE CHARGES REFERENCED HEREIN, IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOST PROFITS AND DAMAGES RESULTING FROM LOSS OF USE, OR LOST DATA, OR DELIVERY DELAYS, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY THEREOF; OR FOR ANY CLAIM BY ANY THIRD PARTY.

B. ANY ACTION AGAINST DGC MUST BE COMMENCED WITHIN ONE (1) YEAR AFTER THE CAUSE OF ACTION ACCRUES.

7. GENERAL
A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts, excluding its conflict of law rules. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer. DGC hereby rejects all such different, conflicting, or additional terms.

8. IMPORTANT NOTICE REGARDING AOS/VS INTERNALS SERIES (ORDER #1865 & #1875)
Customer understands that information and material presented in the AOS/VS Internals Series documents may be specific to a particular revision of the product. Consequently user programs or systems based on this information and material may be revision-locked and may not function properly with prior or future revisions of the product. Therefore, Data General makes no representations as to the utility of this information and material beyond the current revision level which is the subject of the manual. Any use thereof by you or your company is at your own risk. Data General disclaims any liability arising from any such use and I and my company (Customer) hold Data General completely harmless therefrom.

ECLIPSE®
MV/Family
(32-Bit)
Systems
Principles of
Operation
014-001371-01

Cut here and insert in binder spine pocket

Data General
Data General Corporation, Westboro, Massachusetts 01580

014-001371-01