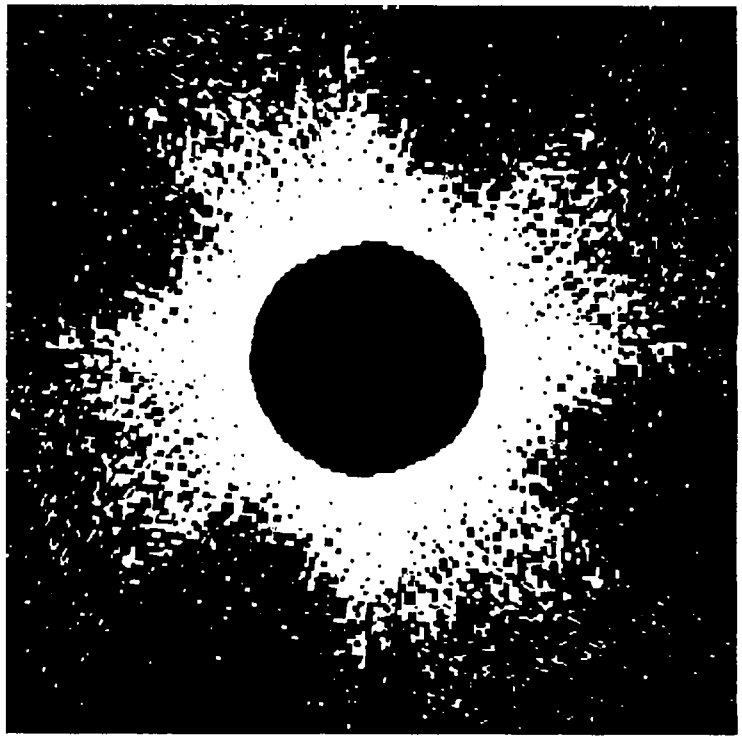
DataGeneral



ECLIPSE MV/Family 32-Bit Systems Principles of Operation

Notice

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

DGC reserves the right to make changes in specifications and other information contained in this document without prior notice, and the reader should in all cases consult DGC to determine whether any such changes have been made.

The terms and conditions governing the sale of DGC hardware products and the licensing of DGC software consist solely of those set forth in the written contracts between DGC and its customers. No representation or other affirmation of fact contained in this document including but not limited to statements regarding capacity, response-time performance, suitability for use or performance of products described herein shall be deemed to be a warranty by DGC for any purpose, or give rise to any liability of DGC whatsoever.

In no event shall DGC be liable for any incidental, indirect, special or consequential damages whatsoever (including but not limited to lost profits) arising out of or related to this document or the information contained in it, even if DGC has been advised, knew or should have known of the possibility of such damages.

Ordering No. 014-000704

Printed in the United States of America
Rev. 05, September 1986

Revision History:

Original Release – December 1981
First Revision – March 1982
Second Revision – February 1983
Third Revision – August 1984
Fourth Revision – January 1986
Fifth Revision – September 1986

A vertical bar or an asterisk in the margin of a page indicates substantive change or deletion, respectively, from the previous revision.

Preface

The *ECLIPSE MV/Family 32-Bit Systems Principles of Operation* manual explains the processor independent concepts, functions, and instruction set to an assembly language programmer. Processor dependent information, available in machine-specific supplements, complements this global manual.

A related manual, the *ECLIPSE MV/Family Instruction Reference Booklet*, provides a brief summary of the instruction set and related information. The reference booklet lists each instruction by assembler-recognizable mnemonic with a shorthand description of its function.

Manual Organization

This manual contains 11 chapters. Chapter 1 gives an overview of the 32-bit ECLIPSE MV/Family system of computers as well as a hardware summary via the appropriate machine-specific supplement.

Chapters 2 through 10 present, in a functional framework, processor independent (and machine-specific dependent in machine-specific supplements) concepts, functions, and instruction set. 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

Chapter 11 provides the Instruction Dictionary.

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

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

Machine-specific supplements include Appendixes E through G that 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.

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	or **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	or *italic* characters indicate a variable argument in an Assembler statement. When you include the *italic* argument with an Assembler statement, substitute a literal value for the variable argument.
*	An asterisk indicates multiplication. For instance, 2*3 means 2 multiplied by 3.
ac	The ac abbreviation indicates a fixed-point accumulator.
acs	The acs abbreviation indicates a fixed-point accumulator called a *source accumulator*.
acd	The acd abbreviation indicates a fixed-point accumulator called a *destination accumulator*.
fac	The fac abbreviation indicates a floating-point accumulator.
facs	The facs abbreviation indicates a floating-point accumulator called a *source accumulator*.
facd	The facd abbreviation indicates a floating-point accumulator called a *destination accumulator*.

Coordinating Machine–Specific Supplements

This revision of the *ECLIPSE MV/Family 32–Bit Systems Principles of Operation* manual supersedes all previous revisions and contains the most up–to–date information for the ECLIPSE MV/Family computers. As such, certain revisions of the machine–specific supplements contain information which may be disregarded when creating a machine–specific manual.

The following table lists the various supplement revisions and the pages within those supplements to be discarded. Note that the supplement revision number may be found on the manual's *Notice* page. (If your particular machine's supplement or *Functional Characteristics* manual is not listed, then that manual contains no pages to discard.)

Manual	Ordering Number	Revision Number	Discard Pages
ECLIPSE MV/2000 DC and DS/7500 Series Systems Principles of Operation	014–001203	00	7–9 through 7–12 7–21 through 7–24 11–511/11–512 11–521/11–522 11–529/11–530
		01	None
ECLIPSE MV/7800 Series Systems Principles of Operation	014–001180	00	Chapter 7 11–49/11–50 11–507 through 11–532
ECLIPSE MV/8000 II System Principles of Operation	014–001227	00	7–9 through 7–12 7–21 through 7–24 11–511/11–512 11–521/11–522 11–529/11–530
ECLIPSE MV/10000 Class Systems Principles of Operation	014–001228	00	7–9 through 7–12 7–21 through 7–24 11–511/11–512 11–521/11–522 11–529/11–530
ECLIPSE MV/20000 Series Systems Principles of Operation	014–001169	00	D–1 through D–4 D–9/D–10

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

9 Memory and System Management

10 ECLIPSE 16-Bit Programming

11 Instruction Dictionary

A Register Fields

B Fault Codes

C Reserved Memory Locations

D Load Control Store Instruction

E Standard I/O Device Codes

F Context Block Formats

G Instruction Execution Times

Glossary

Index

Figures

Tables

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

Functional Capabilities

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

Registers

All ECLIPSE MV/Family computers implement the following registers:

- Four 64-bit floating-point accumulators
- Four 32-bit fixed-point accumulators
- One 16-bit processor status register
- 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

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: four 32-bit fixed-point accumulators (AC0-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 double-word (32 bits). (See Figure 1-2.)

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.

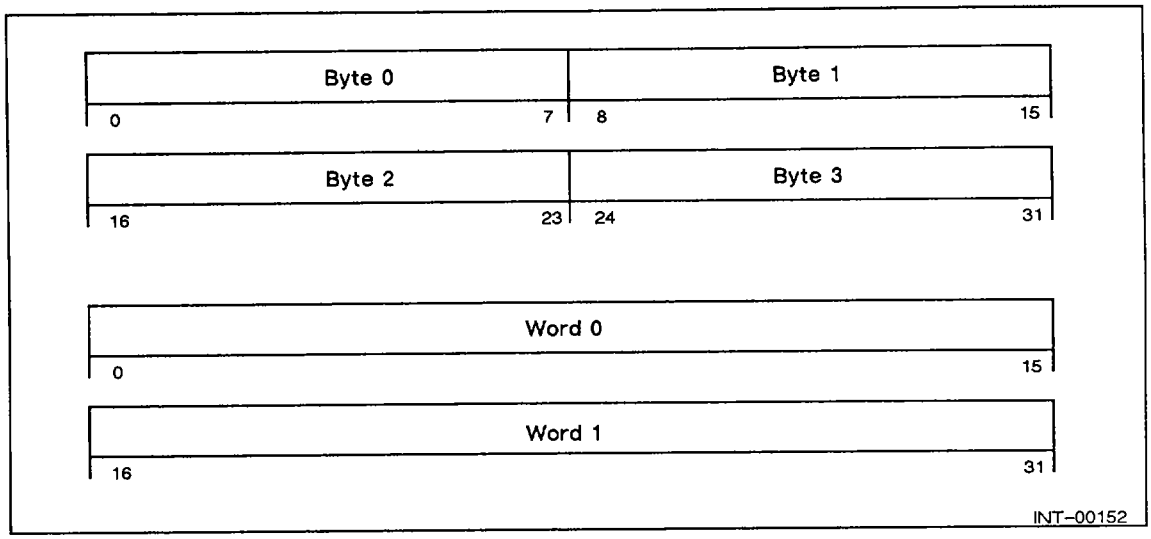


Figure 1-2 *Fixed-point accumulator*

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: four 64-bit floating-point accumulators (FPAC0-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. (See Figure 1-3.)

A single-precision number requires a double word (two consecutive words), while a double-precision number requires two double words (four consecutive words).

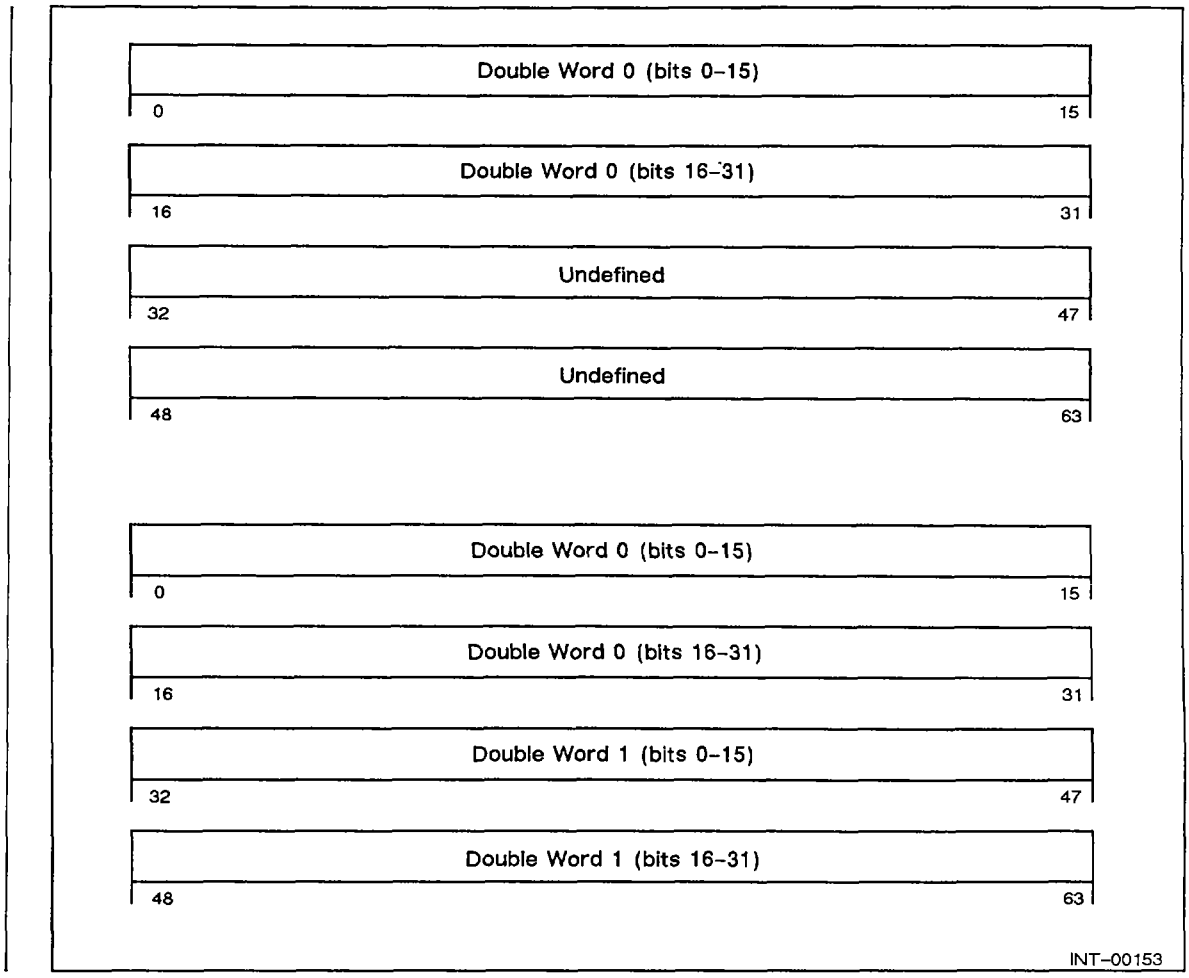


Figure 1-3 *Floating-point accumulator*

Floating-Point Status Register

The floating-point status register contains 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 a register 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 double words that support the 32-bit processor 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 "Memory Management" section 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.

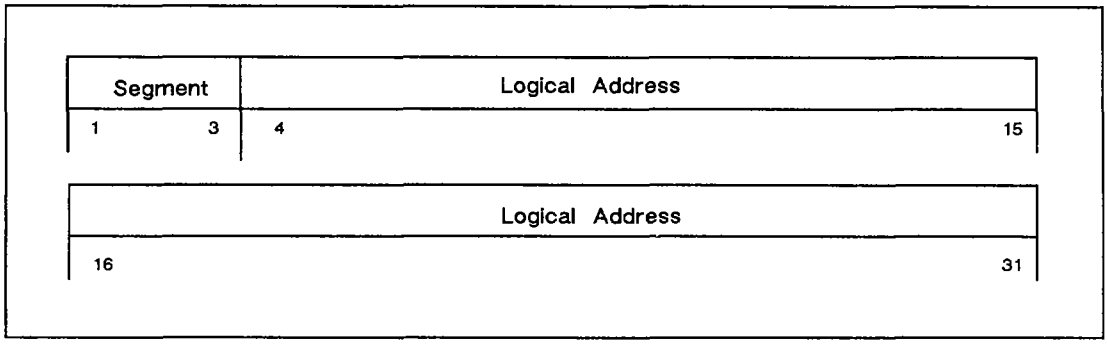


Figure 1-4 *Program counter format*

In Figure 1-4,

Segment	Bits 1-3 specify the current segment. The processor provides specific procedures for modifying the current segment field.
Logical Address	Bits 4-31 specify a 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 "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 that contain

- 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).

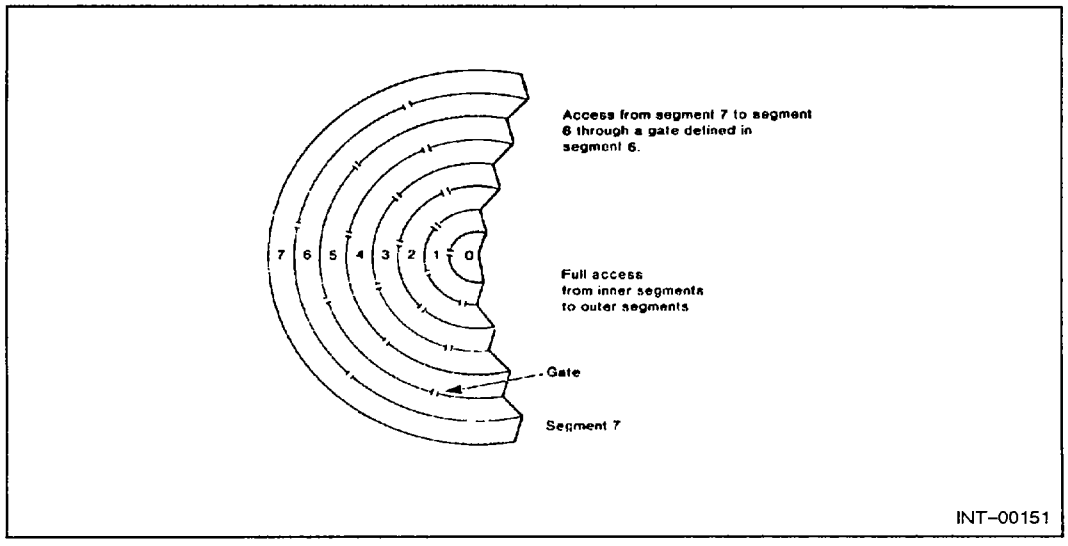


Figure 1-5 *Logical address space*

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-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. Refer to the chapter "ECLIPSE 16-Bit Programming" for additional information.

This manual references ECLIPSE 16-bit instructions as ECLIPSE instructions, and ECLIPSE MV 32-bit instructions as MV 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 double-word boundaries.*

The instruction that the processor accesses can be a single word or multiple words. The operand can be a bit, byte, word, double word, or multiple words (e.g., **BLM**). A memory reference instruction specifies the address of the instruction or data.

A *memory reference instruction* refers to a class of instructions that accesses 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 protection violation fault. Refer to the chapter "Memory and System Management" for further details on page access and the chapter "Program Flow Management" for more information on 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 the chapter "Program Flow Management" for more information on protection violation faults.

Memory Reference Instructions

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 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 to 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 specify 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.

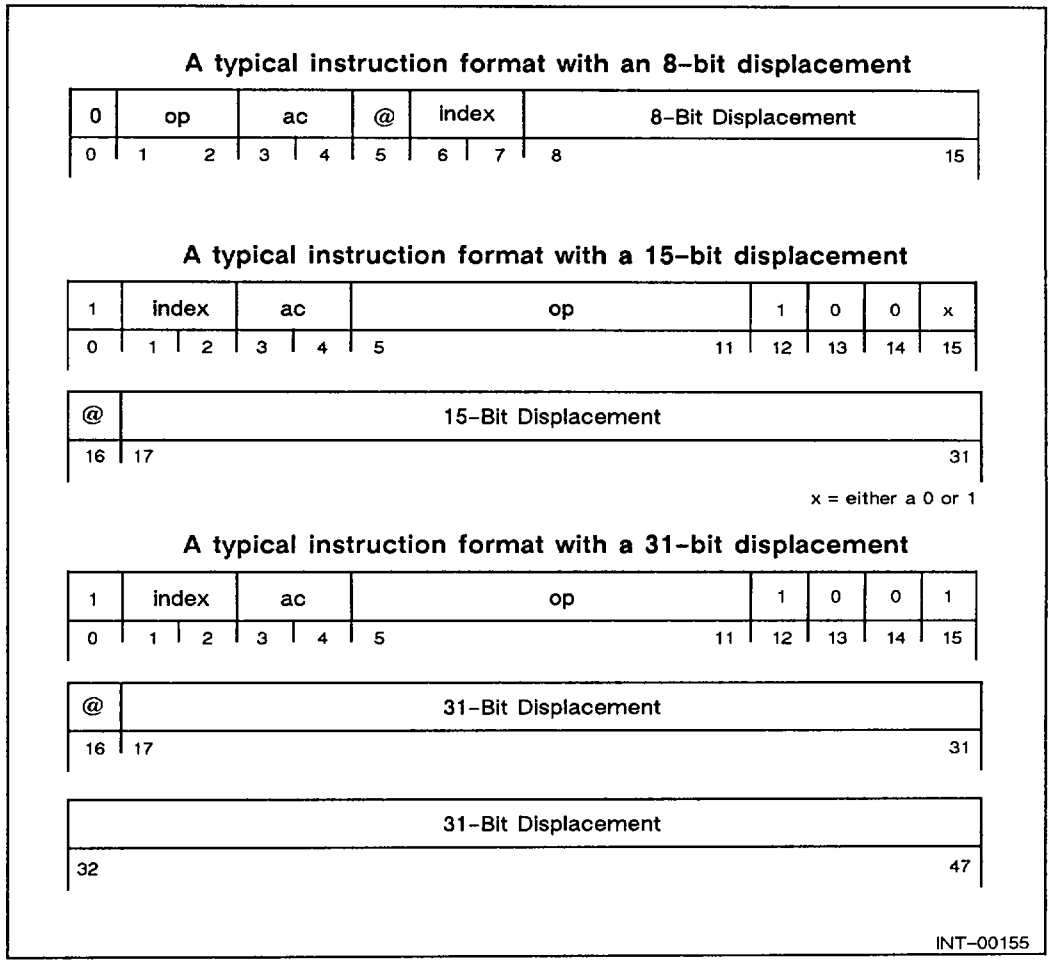


Figure 1-6 *Memory reference instruction word addressing formats*

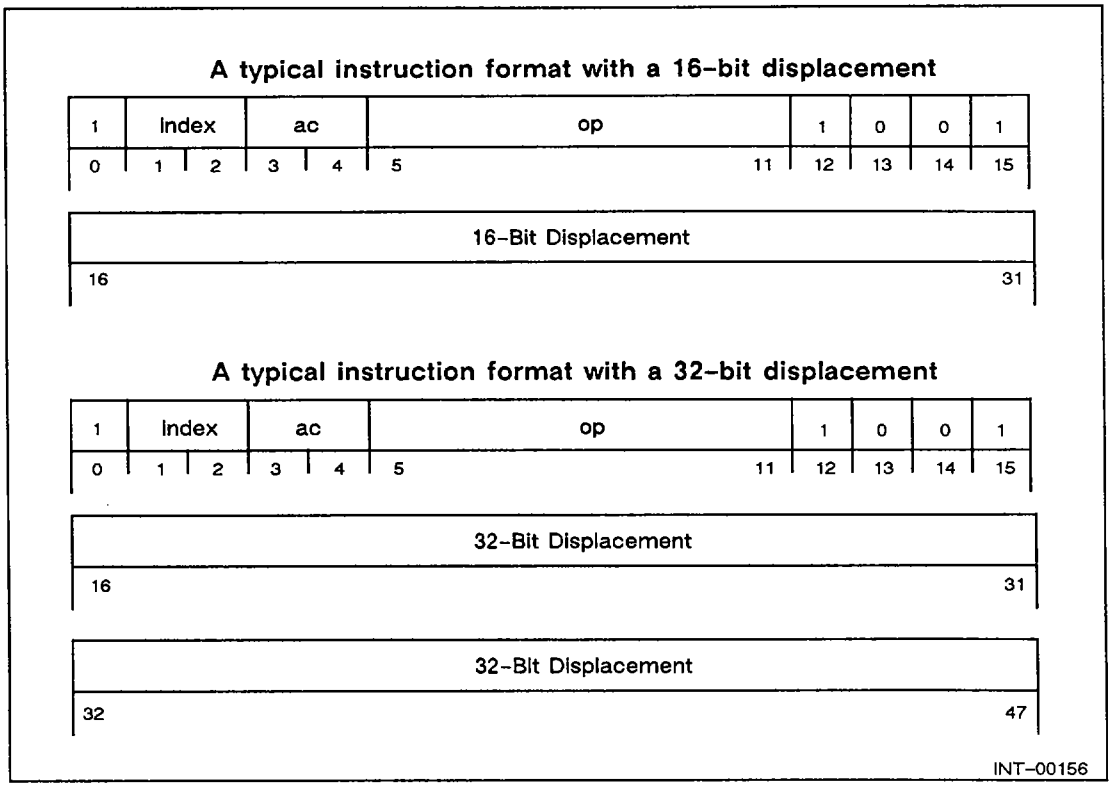


Figure 1-7 *Memory reference instruction byte addressing formats*

Address Modes

Using the index field (Table 1-1), 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 1-1.

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 1-1) 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 double word.*

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.

  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 must be equal to or less than 15.

NOTE: *If the processor attempts to resolve more than 15 indirect addresses, a protection violation occurs.*

Table 1-1 *Effective addressing*

Address Mode	Intermediate Index Bits	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.*

** *X or L corresponds to the prefix of an instruction mnemonic, which identifies an instruction containing an 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 (for fixed- or floating-point computation), the processor first resolves an effective address.

The processor accesses an operand as a bit, byte, several bytes, word, double word, or several double words. Tables 1-2 and 1-3 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 double words, it must first access a word.)

Table 1-2 *Word-oriented data*

Instruction	Address Width (Bits)	Displacement Width (Bits)
LDA	16	8
ELDA	16	16
XNLDA	32	16
XWLDA	32	16
LNLDA	32	32
LWLDA	32	32

Table 1-3 *Byte data*

Instruction	Address Width (Bits)	Displacement Width (Bits)	Index =
STB	16	0	Byte address in any ac
ELDB	16	16	Word address (absolute, PC-relative, AC-relative)
WSTB	32	0	Byte address in any ac
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 fixed-point computation. A fixed-point instruction mnemonic with a prefix of N (such as **NADD**) indicates a narrow or one word operand. An instruction that requests a word (such as **XNLDA**) supplies the effective address parameters to the processor. The processor then resolves the effective address.

Double Word

The processor accesses a double word operand for fixed-point or floating-point computation. 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 requires one double word, while a double-precision instruction requires two double words.

An instruction that requests a double word (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 double-word 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 word 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.

- 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.

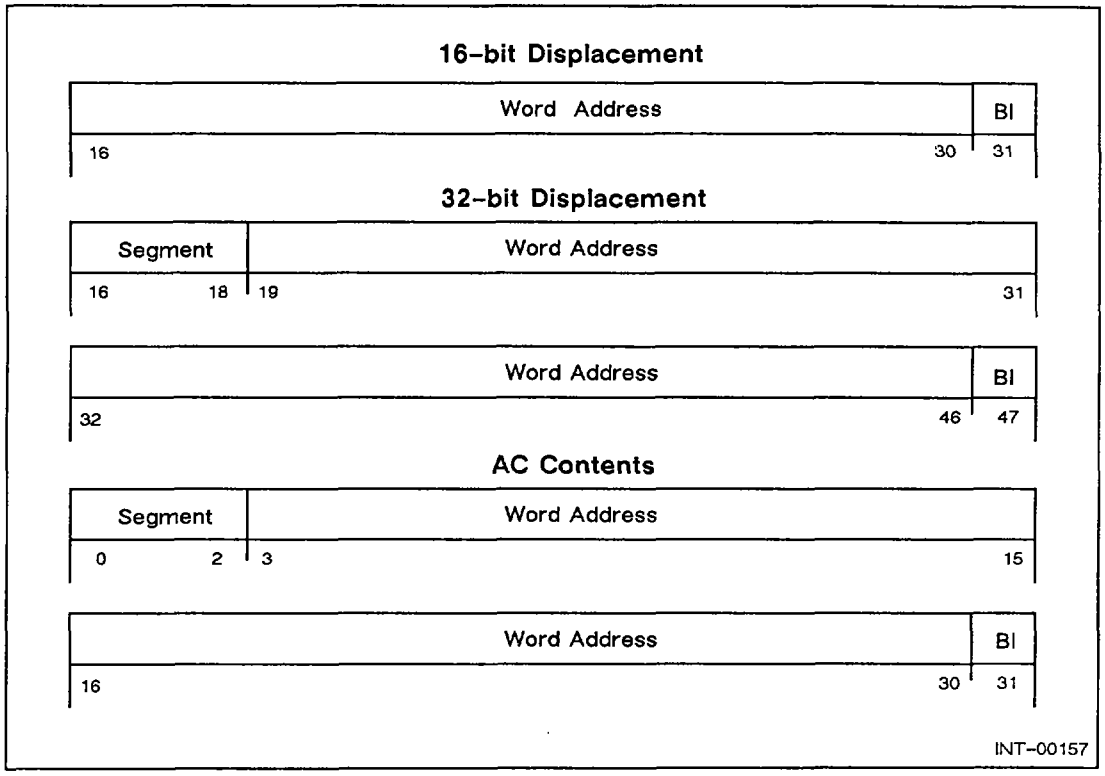


Figure 1-8 *Byte pointer format*

In Figure 1-8,

Segment	The segment field identifies the current or an outward memory segment.
Word Address	The word address field identifies a 16-bit word in the memory segment.
BI	The BI field identifies the byte. When the BI field equals zero, the processor accesses the most significant byte (bits 0-7). When the BI field equals one, 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.

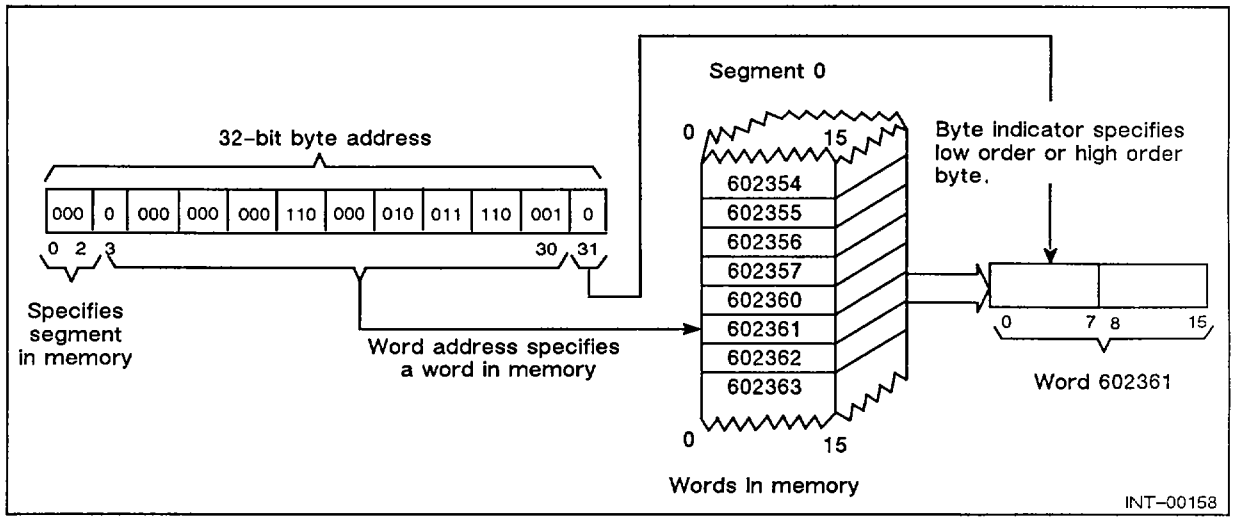


Figure 1-9 *Byte addressing*

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.

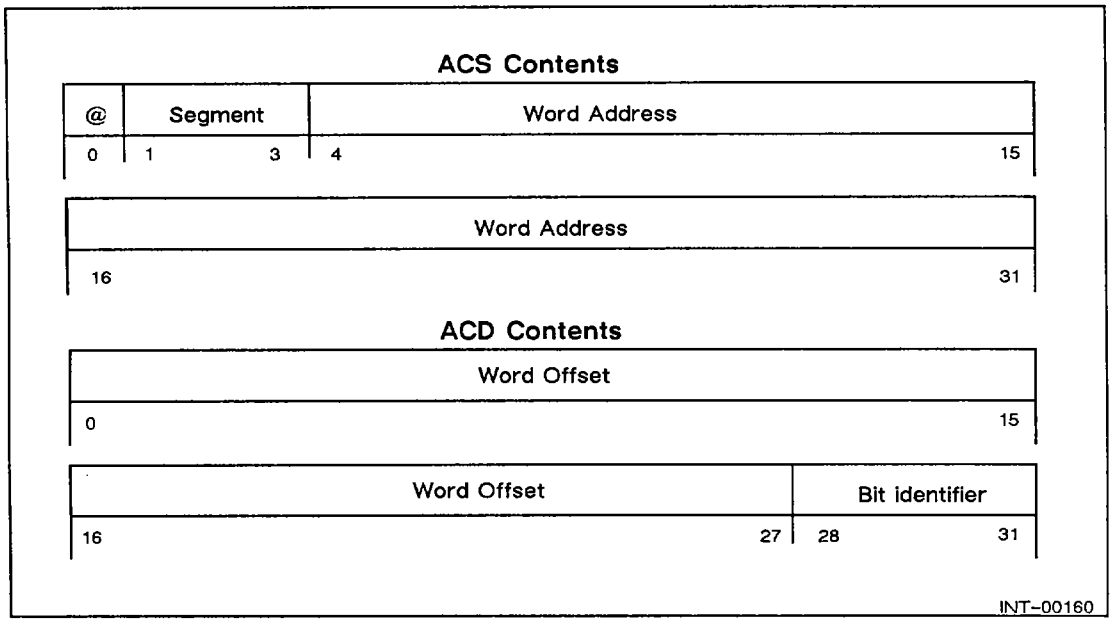


Figure 1-10 *Bit pointer format*

In Figure 1-10,

@	When the @ field equals one, it identifies an indirect address. When the @ field equals zero, it identifies a direct address.
Segment	The segment field identifies the current or an outward memory segment.
Base Word Address	The word address field identifies a 16-bit word in the memory segment.
Base Word Offset	The processor adds the word offset bits, an unsigned integer, to the effective address and arrives at a final word address (Figure 1-11).
Bit Identifier	The bit identifier field specifies the bit position (0-15) in the final word.

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.

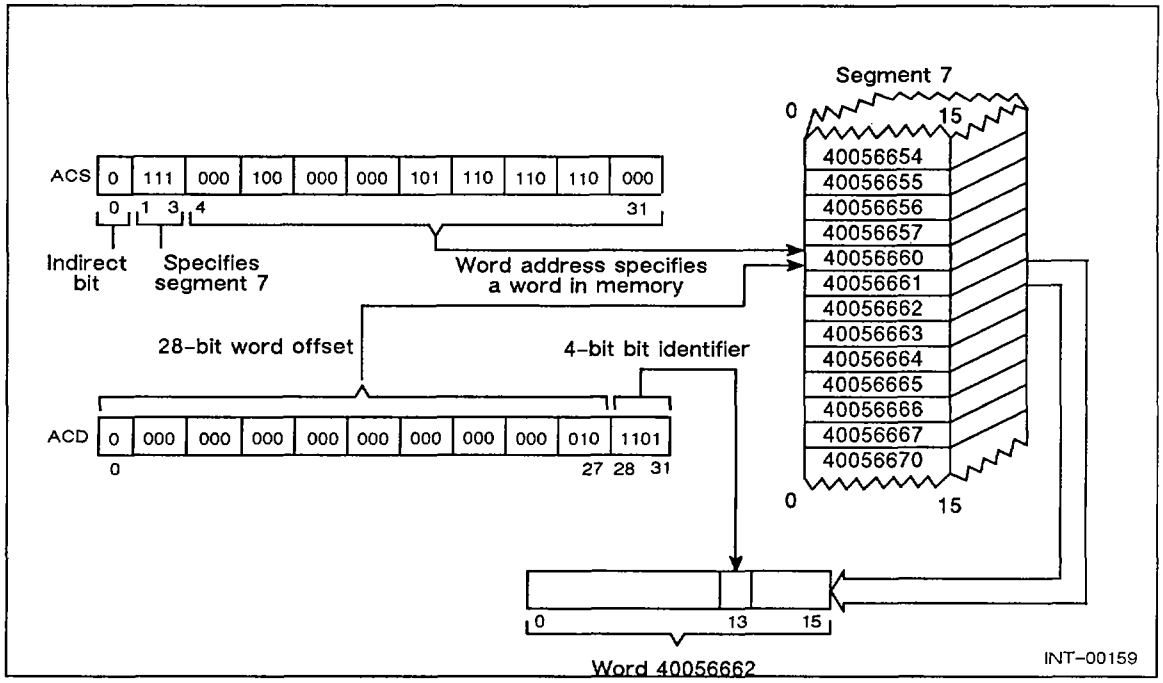


Figure 1-11 *Bit addressing*

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 1-4 lists the validity checks (or faults).

Table 1-4 *Faults*

Fault	Type
Nonresident page	Privileged
Protection violation	Nonprivileged
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 input/output (I/O) access on a per ring basis, and it 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 a specific segment or segments.

Accessing and changing a protection mechanism requires a privileged instruction (executable only in segment 0) or data access. Either one is 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.

ECLIPSE MV/20000 Series Hardware Summary

This summary describes the ECLIPSE MV/20000™ series computer systems and their initial processor conditions.

The ECLIPSE MV/20000 series computers include the ECLIPSE MV/20000 Model 1 computer and the ECLIPSE MV/20000 Model 2 computer. When we refer to the ECLIPSE MV/20000 series computer system within this manual, we include both systems. We will differentiate between the systems as necessary.

- The ECLIPSE MV/20000 Model 1 computer is a single-CPU system supporting from one to three input/output controllers (the third IOC supports only a data channel), an optional hardware floating-point unit (FPU), and up to 64 megabytes of physical memory.
- The ECLIPSE MV/20000 Model 2 computer is identical to the ECLIPSE MV/20000 Model 1 computer with an additional CPU and optional FPUs. (References in this manual to multiple-processor systems denote only the ECLIPSE MV/20000 Model 2 computer.)

The following summarizes what each ECLIPSE MV/20000 series computer supports:

Components/ Subsystems	ECLIPSE MV/20000 Model 1	ECLIPSE MV/20000 Model 2
CPUs	1	2
FPUs	0 or 1	0 or 2*
IOCs	1, 2, or 3	1, 2, or 3
Memory	4-64 Mbytes	4-64 Mbytes

**If a customer chooses the hardware floating-point option, the Model 2 requires two hardware floating-point units, one for each CPU.*

The ECLIPSE MV/20000 series computers are 32-bit data processing systems that retain substantial hardware and software compatibility with 16-bit ECLIPSE® systems. (Kernel 16-bit operating system instructions such as SYC, VCT, and LMP are not supported.)

Main Systems

All ECLIPSE MV/20000 series systems incorporate five major elements as follows:

- One or two central processing units
- A memory system
- An input/output system
- A power system
- A diagnostic remote processor

Figure 1-12 shows the major elements of the ECLIPSE MV/20000 series systems.

Central Processing Unit

The ECLIPSE MV/20000 series 32-bit central processing units (CPUs) execute instructions, generate logical addresses, translate logical addresses to physical addresses, and perform arithmetic and logical data manipulation. Refer to the chapter, "Device Management," for CPU programming information.

Each microcode-controlled central processing unit consists of the following:

- An instruction processor for decoding and executing instructions
- A microsequencer that manages the writable control store, which contains the microinstructions
- An address generator for logical address generation
- An address translation unit for logical-to-physical address translation
- An arithmetic logic unit for data manipulation
- A data cache for high-speed, temporary data storage
- An optional floating-point unit for executing floating-point, integer multiply and divide, and some decimal instructions

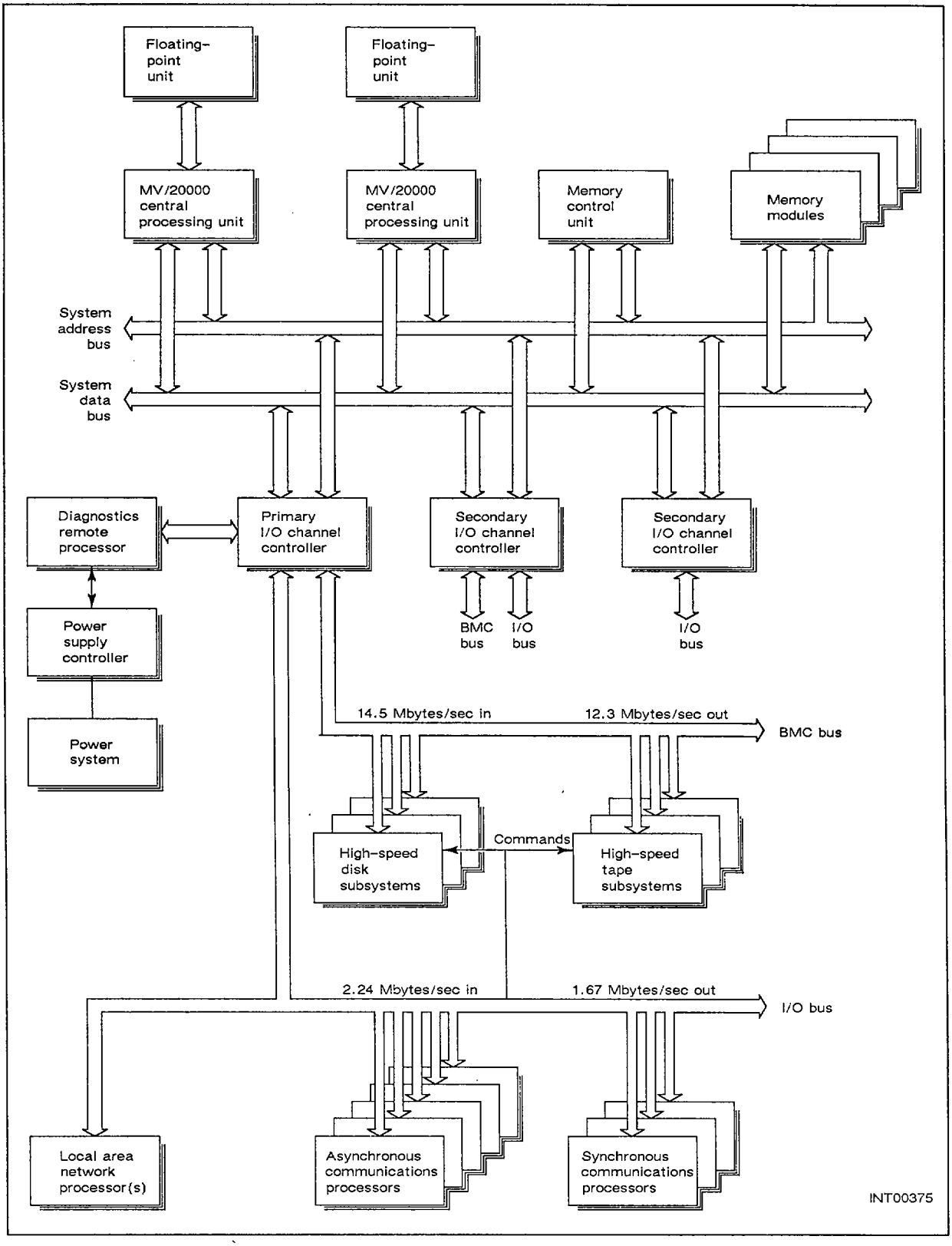


Figure 1-12 *Major elements of the ECLIPSE MV/20000 series system*

Like all ECLIPSE MV/Family processors, the CPU's subsystems work together to form a pipeline for the instruction stream that speeds up execution performance.

The pipeline allows instructions to be decoded and executed simultaneously. Before an instruction is executed, the appropriate starting microcode address must be found. And typically, a logical address needs to be resolved or immediate data needs to be extracted. Certain CPU subsystems are designated to perform these initial tasks; these subsystems form the pipeline. Instructions move through the pipeline in various stages of decoding. At the last stage of the pipeline, the preprocessing done by previous stages lets the CPU execute the instructions in the smallest number of CPU cycles.

The ECLIPSE MV/20000 series computer system uses a four-stage instruction pipeline as shown in Figure 1-13. As the first instruction executes, the second fetches its starting microinstruction, a third decodes its opcode, and finally, a fourth comes into the instruction queue.

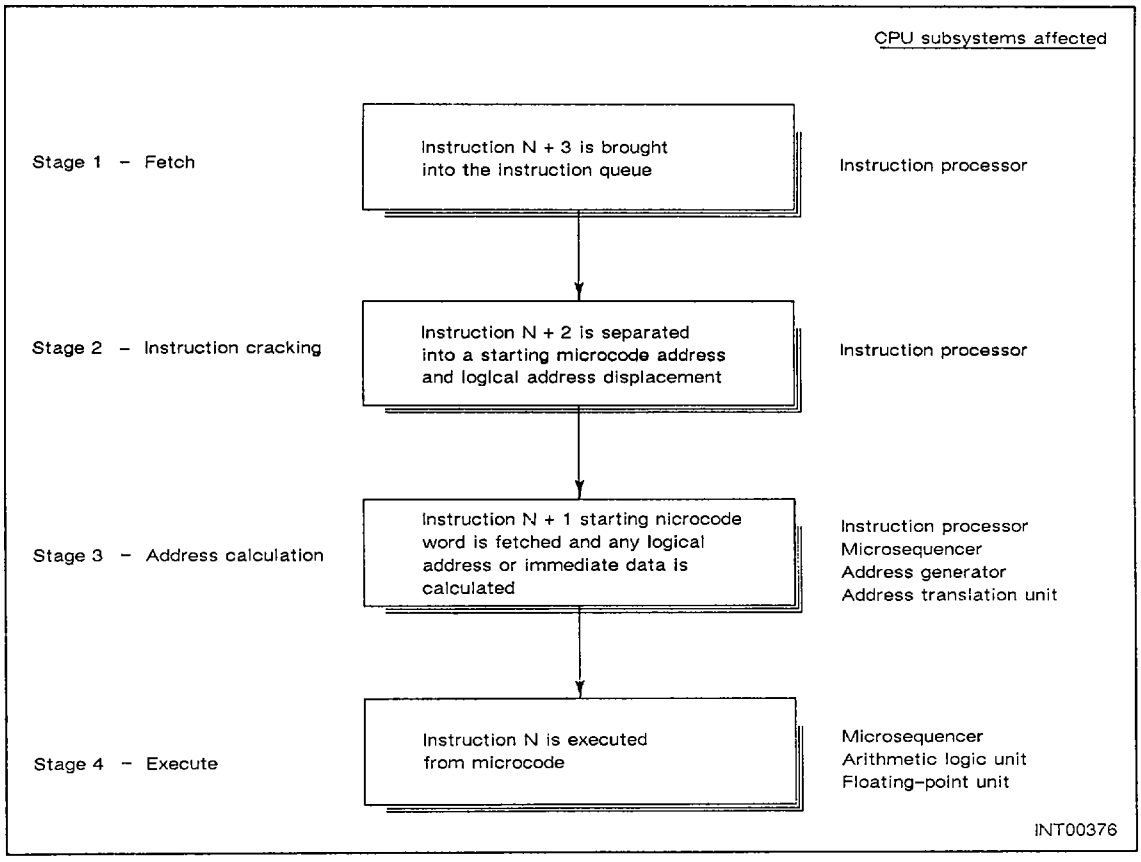


Figure 1-13 *Four-stage instruction pipeline*

Instruction Processor

The instruction processor consists of two main components: the instruction cache and the instruction queue. Besides providing a high-speed buffer for the CPU, the 1-kilobyte-by-32-bit instruction cache implements a look-ahead and look-behind facility. This cache accelerates the execution of applications that contain program loops and can reside within its storage boundaries.

As the entrance to the pipeline, the instruction cache prefetches the instruction stream and feeds each instruction (in double words) through the decode logic of the instruction queue. The decode logic supplies a starting address to the appropriate microroutines located in the microsequencer's writable control stores. As one instruction passes to the microsequencer for the beginning of the execution phase, others are in various stages of decoding. The instruction queue maintains a queue of the next six sequential words after the current instruction.

The instruction cache monitors parity. One parity bit is maintained for each double word in the cache. If a parity error is detected, the double word containing the error is brought in again from main memory. If the error persists, the instruction cache can be physically disabled.

Microsequencer

The microsequencer contains the writable control store, including the microinstructions that drive the CPU. A portion of the writable control store holds the system microcode; the remainder can hold any special microcode written by a user.

The writable control store is RAM-based and stores 6K microwords, 72-bits wide. At system boot time, the writable control store's random access memory (RAM) is loaded from a peripheral. Loading from a peripheral simplifies microcode changes and updates.

The CPU operates at a microinstruction cycle time of 85 nanoseconds. As each microinstruction executes, the output of its control fields drives the remaining CPU subsystems to execute the originally fetched, assembly-level instruction and to keep the pipeline operating.

Address Generator

When memory reference instructions are in the pipeline, the address generator calculates their effective logical addresses. In turn, it passes the logical addresses to the address translation unit, which converts logical addresses to physical addresses.

When the CPU accesses memory, the address generator provides an additional level of pipelining. By computing the logical addresses, the address generator permits the CPU to overlap operand fetches with instruction execution, which further accelerates the execution process.

Address Translation Unit

The address translation unit (ATU) performs the logical-to-physical address conversion process. To efficiently implement this process, the subsystem contains its own cache together with protection and pagetable-access logic. The ATU performs all hardware checks required by the protection system. Access, page, and ring-crossing validations are among the types of hardware checks. If any of these checks fail, the address translator initiates a protection fault to the operating system.

The ATU cache stores the most recently used address translations along with the page-referenced bits used by an operating system for virtual memory management. The cache contains 256 address translations for segments 0 through 3 and 256 entries for segments 4 through 7, in addition to associated protection information.

As each conversion process transpires, the ATU traps the running microinstruction if the needed translation information is not present in its cache. When this occurs, the subsystem uses a pagetable look-up mechanism to access the physical address. Then it loads the address into the ATU cache, and instruction execution proceeds.

The ATU cache monitors parity. One parity bit is maintained for each address in the cache. If a parity error is detected, the ATU subsystem retranslates and reloads the corresponding address. If the error persists, the ATU cache can be physically disabled.

Arithmetic Logic Unit

The arithmetic logic unit (ALU) performs the data manipulation required to execute most arithmetic- and logic-class instructions that deal with fixed-point data, as well as those required to translate and validate commercial data. It also supplies memory and I/O data to executing programs via its accumulators.

The ALU itself is supported by three other units: the shifter, the decoder, and the scratchpad. The shifter performs operations such as shift-and-rotate, byte swap, and sign extend. The decoder interprets commercial data formats. The scratchpad is a high-speed RAM used by the ALU to store constants and certain floating-point operations. It is part of the ATU cache RAM and is accessed through the ATU.

The ALU contains the four 32-bit, fixed-point accumulators; the processor status register; the stack registers; and several general registers.

For floating-point calculations, the ECLIPSE MV/20000 series CPU either uses the optional FPU or performs these calculations with special microcode routines. In either case, the four 64-bit, floating-point accumulators and the floating-point status register are available for use and are located on the FPU or the ALU (without the FPU option). The execution speed differs between the FPU and microcode implementations.

Data Cache

The data cache provides a high-speed, temporary storage buffer for the CPU. Data is read into the cache from main memory as it is called for by the program. The CPU then executes programs directly using the cached data. Data can be read from the cache within the 85-nanosecond CPU cycle time.

Within the data cache, memory is organized into 1,024 16-byte blocks. Each block directly maps to a block in main memory, overlapping every 16 Kbytes. Two tag stores contain the corresponding main memory address for each block. Each tag store contains a validity flag indicating whether or not the data within the cache block is valid. One tag store monitors access to cached data from within the CPU. The other tag store monitors the system bus for memory access requests from other devices. Other devices can include another CPU within the ECLIPSE MV/20000 series computer system.

The data cache uses a technique called write through. Whenever data in the cache is modified by the CPU, the corresponding main memory location is also updated. This procedure insures that other devices on the system bus will read only current data from main memory. If another device updates a main memory location that is also in the data

cache, the tag store monitoring the system bus will invalidate the corresponding cache block. When the CPU accesses data within that block, it will first read in the updated block from main memory.

Once data is brought into the cache, it remains there until invalidated or overwritten by another block. This allows the same look-ahead and look-behind program acceleration as that used by the instruction cache.

The data cache monitors parity. One parity bit is maintained for each byte of data in the cache. If a parity error is detected, the cache block containing the error is brought in again from main memory. If the error persists, the data cache can be physically disabled.

Floating-Point Unit

The floating-point unit (FPU) is a separate processor that interfaces directly with the CPU. This processor executes single- and double-precision, floating-point instructions; integer multiply and divide; intrinsic instructions; and some decimal instructions.

The FPU operates asynchronously with subsystems of the CPU. It is brought on line by the microsequencer subsystem when an appropriate instruction appears in the pipeline. Thus, this parallel processor frees the CPU to perform other tasks, providing substantial throughput acceleration in compute- and data-intensive environments.

The FPU contains four 64-bit, floating-point accumulators, along with the floating-point status register. To increase the accuracy of floating-point operations, the FPU uses hex guard digits while executing floating-point instructions. Then it truncates or rounds the results, depending upon the state of a program-controlled flag.

Multiprocessor Operation

One major performance feature of the ECLIPSE MV/20000 Model 2 system that distinguishes it from other ECLIPSE MV/Family systems is that the Model 2 has two central processing units. The system architecture of the Model 2 was designed so that the second CPU uses the minimum amount of overhead.

Both CPUs connect to the high-speed system bus. They have identical architecture, features, options, and function, which make them independent of one another. Each has its own set of caches and other hardware and software accelerators, including optional FPUs. Because of the write-through ability of the data caches, each processor always contains current data, regardless of the action of the other CPU.

At system startup, one CPU is designated as the initial CPU, performing diagnostic routines and initialization functions first. When the first CPU has passed all of its tests and has loaded its own writable control store, it passes control to the other CPU, which performs a similar initialization procedure. Once running, each CPU operates independently, with its actions determined entirely by program control.

To support the dual-processor configuration, only a few new instructions were added to the ECLIPSE MV/Family instruction set. These instructions allow for individual control of the CPUs, status checks, and cross interrupts. Refer to the chapter, "Device Management," for multiple-processor programming information.

Memory System

The main memory system of the ECLIPSE MV/20000 series computer system consists of a memory control unit (MCU) and up to eight memory modules. Together, these elements provide high-speed access to system memory for the CPUs and I/O system. Further, using the error-detection bits of system memory and their supporting buses, these elements continually check the integrity of all information entering, residing in, and leaving system memory.

Memory Control Unit

The memory control unit detects, corrects, and logs errors, and it also conducts memory integrity checks. The MCU continually monitors accesses to the memory modules that make up main memory. It appends error-correction bits to memory data as it is written; checks and corrects all single-bit and many double-bit errors as memory data is read; and selectively performs an integrity check on the contents of memory during memory-refresh operations. This latter process, called sniffing, prevents single-bit and many double-bit errors from accumulating and destroying correctable data. If a memory error occurs, the MCU stores both the address and a code indicating the type of error.

The MCU also monitors parity on the system address and system data buses. If a parity error occurs on one of these, the CPU is informed through an instruction trap.

Memory Modules

The memory modules that compose the system's main memory are available in increments of 4 or 8 megabytes. The modules use 256-Kbit dynamic RAM elements.

*

The ECLIPSE MV/20000 Model 1 and 2 memory systems support up to eight memory modules with a maximum memory capacity of 64 megabytes.

Input/Output System

The input/output (I/O) system for the ECLIPSE MV/20000 series computer system handles all communications with the system peripherals. These communications take place over either the standard ECLIPSE I/O bus or the burst multiplexor channel (BMC) bus.

External devices connect to the I/O buses; internal devices are found on various CPU boards. External devices include communications and network processors, a variety of peripheral devices, and the power supply controller. Internal devices include the asynchronous interface for the system console, the real-time clock, and the programmable interval timer.

The I/O system is program compatible and electrically compatible with ECLIPSE 16-bit computers and the MV/Family of ECLIPSE 32-bit processors. This means that the ECLIPSE MV/20000 series computers support all standard Data General peripherals. Figure 1-14 shows the organization of the I/O system.

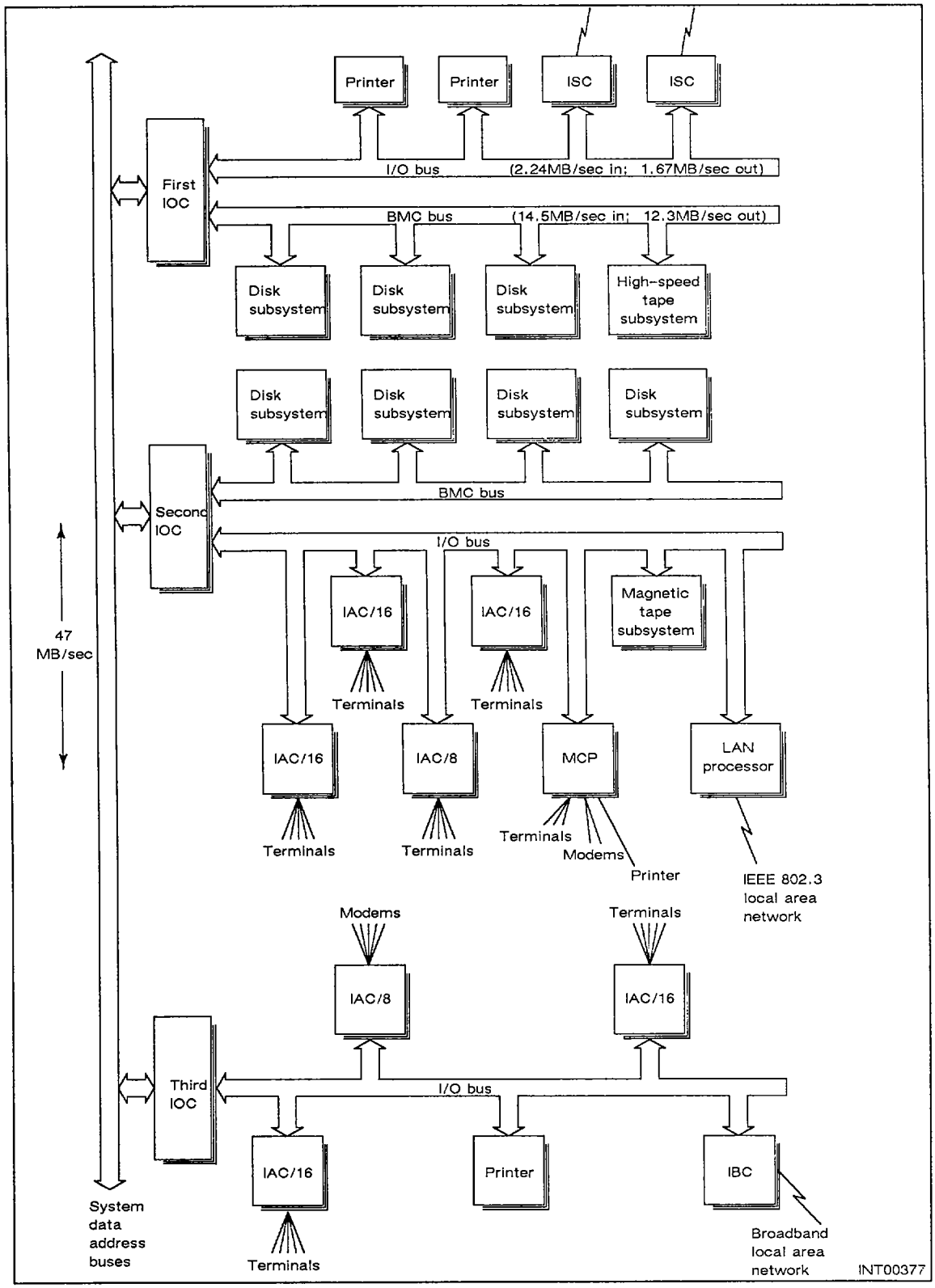


Figure 1-14 *I/O system*

One or, optionally, two or three I/O channel controllers (IOCs) provide the gateways between peripherals and the system buses of the ECLIPSE MV/20000 series computers.

The ECLIPSE MV/Family I/O instructions allow communications with peripherals on the I/O channels—either individually on one I/O channel or simultaneously on all I/O channels. For further information, refer to the chapter, "Device Management."

Within the I/O system, there are two types of buses around which peripherals are organized according to speed and priority: the burst multiplexor channel (BMC) and the I/O bus. Each of the three I/O channel controllers generates its own I/O bus. However, only two (the primary and one optional) I/O channel controllers have the BMC facility.

The burst multiplexor channel supports high-speed devices, such as disks and tapes, with direct-memory-access transfers of block-oriented data. The BMC operates at a bandwidth of 14.5 megabytes per second for device input and 12.3 megabytes per second for device output.

The I/O bus supports medium- and low-speed devices. It provides a data channel facility to support medium-speed devices with direct-memory-access transfers of single words or variable-length blocks of data. The data channel facility of the I/O bus operates at two different bandwidths: normal and extended. The normal bandwidth data channel operates at 2.24 megabytes per second for input and 1.67 megabytes per second for output. The extended bandwidth operates at 1.47 megabytes per second for input and 1.0 megabytes per second for output. Data channel controller boards inserted into main chassis slots designated I/O-only operate at normal bandwidth. Data channel controller circuit boards inserted into main chassis slots designated as either I/O or bus repeater as well as boards inserted into any slots of an expansion chassis operate at the extended bandwidth.

The I/O bus provides a programmed I/O facility for low-speed transfers of single-word information (instructions, commands, status, bytes, or characters) between a device and an accumulator in the CPU. These transfers are instrumental in setting up the parameters of the transfers for the higher speed channels. (Devices on the BMC bus also interface to the I/O bus for CPU control of device setup and data transfer initiation.) The ECLIPSE MV/20000 series computers execute all ECLIPSE 16-bit programmed I/O instructions. They execute them exactly as ECLIPSE 16-bit systems do.

For further information, refer to the chapter, "Device Management."

I/O Channel Controller

I/O channel controllers function much like a central processor, communicating with the CPUs and memory via the system buses. The IOCs support data channel and BMC devices with internal and external (to the I/O system) bus protocol transformations. They maintain the maps that change data channel and BMC addresses to physical memory addresses. IOCs allow intelligent device controllers to load their own memory maps while maintaining protection bits that prevent errant devices from gaining access to protected memory areas.

Communications Controllers

The ECLIPSE MV/20000 series computers support a wide range of intelligent communications and network controllers that function as full I/O processors. These independent processors include the following:

- Intelligent asynchronous controllers
- Intelligent synchronous controllers
- Multi-communications processors
- Intelligent LAN controllers
- Intelligent computer-to-PBX (private branch exchange) interfaces

Each controller is microprocessor based with local memory and industry-standard line interfaces. I/O processors connect to the I/O bus of the input/output system. These processors use a cross-interrupt facility for direct communication with the ECLIPSE MV/20000 series CPU. Data transfers take place to and from main memory via the data channel facility. Refer to the appropriate communications controller manual for detailed information.

Power System

The ECLIPSE MV/20000 series power system includes power supplies that provide regulated dc voltage to the system, a battery back-up unit (BBU), and a microprocessor-based controller. The ECLIPSE MV/20000 Models 1 and 2 intelligent power supply controller (PSC) allow the CPUs to communicate with their power systems through the diagnostic remote processor (DRP). *

Each power supply controller brings up power to its system in a specified sequence and then monitors and manages power parameters. Further, to prevent system failures resulting from random ac-line dropout conditions, they supply a powerfail/auto-restart facility that allows the system to recover gracefully. During power outages, the battery back-up units supply dc power to the CPU chassis and the expansion chassis for up to two minutes, giving the operating system time to prepare for an orderly shutdown.

During operations, the power supply controllers also check for blower failures and excessive power supply and cabinet temperatures, and they can interrupt the CPU to report critical failure conditions.

Refer to the chapter, "Device Management," for further information.

Diagnostic Remote Processor

The diagnostic remote processor (DRP) is a dedicated processor that serves as the front end to the ECLIPSE MV/20000 series computer systems. The DRP provides the line interface to the system console. Further, the DRP connects to the major components of the ECLIPSE MV/20000 series computer through the primary I/O channel controller. The DRP performs the following major tasks:

- Supports system powerup and initialization
- Executes part of the system control program

- Logs system errors
- Supports the system console
- Interfaces to the front panel switches and light-emitting diodes (LEDs)
- Interfaces directly to the power supply controller
- Provides the system clocks
- Supports diagnostic troubleshooting
- Provides ports for remote diagnostic communications

Refer to the chapter, "Device Management," for further information.

Powerup and Initialization

At powerup, the DRP first checks its own hardware, and then resets the CPUs and other system elements. Next, each CPU performs diagnostic routines that check for minimal internal hardware integrity as well as I/O and microcode load paths. Upon successful completion of this testing, the CPUs begin initialization. Full system initialization includes loading the ECLIPSE MV/20000 series microcode and the ECLIPSE MV/20000 series operating system.

The DRP monitors each step in the testing and initialization sequence. In dual-processor configurations, the DRP synchronizes the CPUs. If an error occurs, the DRP responds by displaying an error code indicating the faulty element.

The DRP includes two facilities for more maintenance-free initialization: a battery backed-up time-of-day clock and a nonvolatile configuration RAM. The time-of-day clock comes preset. It keeps the time of day for the operating system and for time stamping messages in the error log. The configuration RAM contains the CPU, IOC, and memory configurations for system configuration and testing. It also contains serial numbers of parts and service contract numbers to aid field service personnel.

System Control Program

The system control program (SCP) is a ROM-based operating system that runs partly on the DRP and partly on the main CPUs. During full operation, the SCP allows the system operator to load, examine, and modify the main memory or the writable control store and to step through the instructions of a program.

The portion of the SCP that runs on the ECLIPSE MV/20000 series CPUs is microcoded and resides in read only memory (ROM) on the CPU boards. The remainder of the SCP is macrocoded and is executed by the DRP directly from on-board ROM.

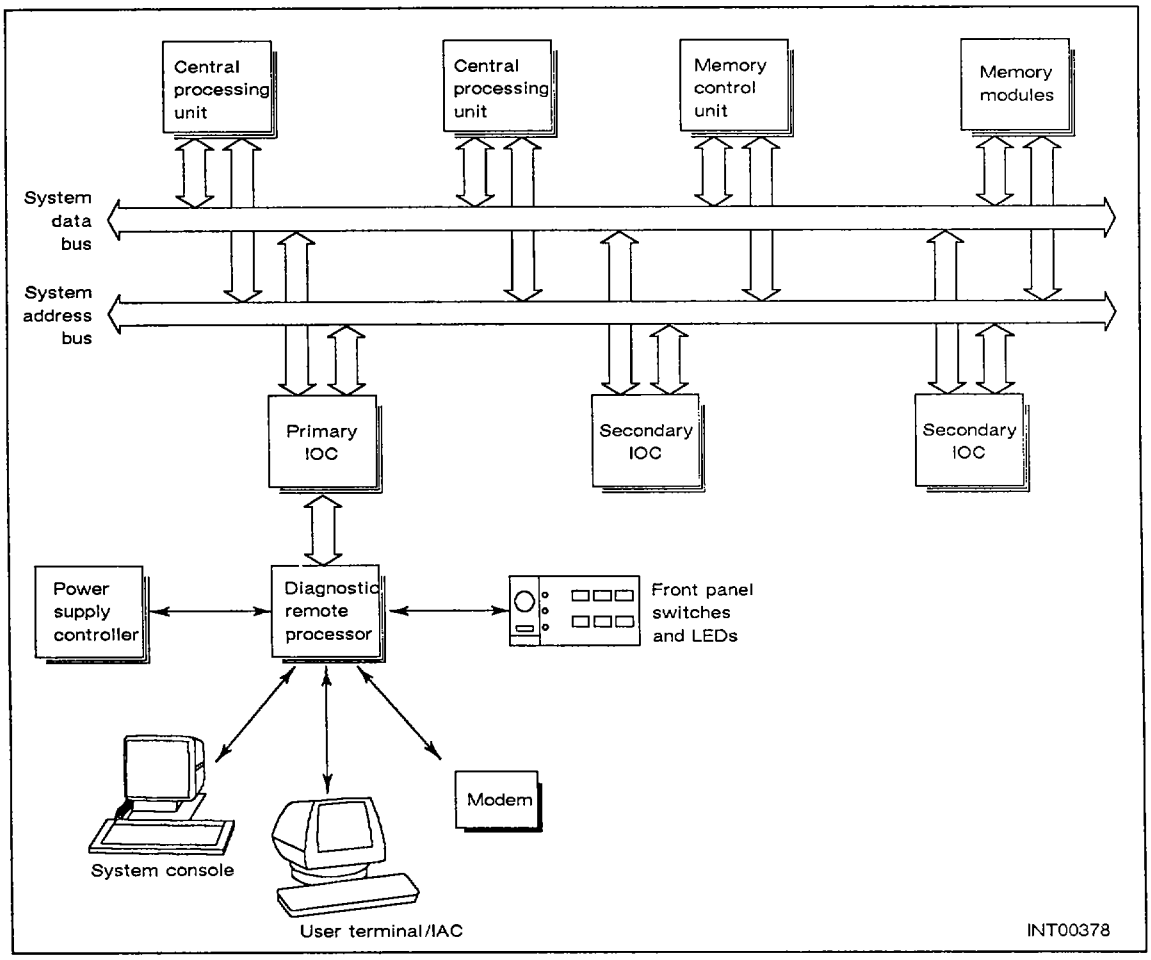


Figure 1-15 *Diagnostic remote processor*

Before the SCP can become fully functional, the various system elements must be tested, including the system buses and each CPU. Once the CPUs have verified that they can load their writable control stores, the SCP operating system (SCPOS) microcode is executed. However, if a system element fails a test before verification, a kernel system identifies the faulty element.

Error Detection and Logging

Most errors are detected by the ECLIPSE MV/20000 series CPU. The CPU then passes the error information to the DRP through programmed I/O commands. For correctable errors, the CPU notifies the DRP and continues operation. For uncorrectable errors, the CPU notifies the DRP, suspends normal operations, and enters into the SCP mode. The SCP then attempts to display a message indicating the nature of the error.

When the DRP receives an error message, it records the message in battery backed-up RAM. Each recorded error is time stamped from the time-of-day clock as it is logged. In addition, the DRP keeps track of the frequency of certain correctable memory errors. This logging procedure allows the ECLIPSE MV/20000 series operating system to track correctable errors. It also provides troubleshooters with important information about both correctable and uncorrectable errors.

The error logging capability of the DRP takes full advantage of the integrity checks built into the system hardware, including byte parity checks on all major system data paths.

System Console

The DRP provides the ECLIPSE MV/20000 series computer system with a direct-line interface for the system console terminal through which the operator can control the activities of the system. This asynchronous port is independent of any other asynchronous communications controllers that may be present in the system.

Front Panel

The DRP directly interfaces with the front panel of the ECLIPSE MV/20000 series computer. The DRP continually monitors the status of the switches on the front panel and controls the state of the indicator LEDs and seven-segment displays.

Power Supply Interface

The intelligent power supply controller is connected directly to the DRP. The CPU can communicate with the controller through the DRP using standard I/O commands.

System Clock

The DRP supplies a free-running, 42.5-nanosecond clock to the ECLIPSE MV/20000 series computer. All clocks for the rest of the system (except the DRP's time-of-day clock) are derived from this system clock. The CPU, for example, uses the system clock to generate an 85-nanosecond clock that provides its microinstruction cycle time. For diagnostic purposes, the system clock can be margined 5 percent faster to 40.5 nanoseconds.

Diagnostic Troubleshooting

The DRP, in combination with diagnostic software, supplies system engineers and troubleshooters with diagnostic capabilities. Since the DRP has its own separate processor, it can perform certain diagnostic functions independent of the status of the CPUs.

During diagnostic testing, troubleshooters can margin the main power supply and system clock frequency. These procedures stress the machine to isolate intermittent random failures that are otherwise difficult to diagnose.

Remote Diagnostic Communications

The DRP supports remote diagnostics that allow Data General's staff of system engineers to diagnose hardware and software system problems off site.

In addition to the system console facilities, the DRP provides two other asynchronous ports for diagnostic support. One of these is a modem interface, allowing remote access to the DRP. The full set of system console functions is available over the modem, as well as the ability to load each CPU's writable control store.

The other asynchronous port can connect to both a user terminal and an intelligent asynchronous controller, such as an IAC/16. This port enables the modem to communicate with the CPU through a standard IAC connection. It also allows individuals using the modem and the user terminal ports to send messages to each other.

ECLIPSE 16-Bit Compatibility

The ECLIPSE MV/20000 series computers support the instruction mnemonics and binary opcodes of most instructions implemented on ECLIPSE systems. This means that most programs that execute on ECLIPSE 16-bit computers also execute on the ECLIPSE MV/20000 series computers without program recompiling or reassembling.

Refer to the chapter, "ECLIPSE 16-Bit Programming," for further information.

Initialization

The processor assumes the physical mode upon powerup or system reset.

When the processor first powers up (and the system microcode loads) or after a system resets, the processor does the following:

- Disables the logical address translation, making logical and physical addresses equal
- Disables the logical address translation protection system
- Sets indeterminate values for referenced and modified bits
- Sets the processor status register and bits 0 through 9 of the floating-point status register to 0
- Disables error reporting via the diagnostic remote processor
- Sets undefined contents for data channel maps
- Selects channel 0 as the default I/O channel
- Clears bits 3, 4, 7, 8, and 14 of the I/O channel definition register

When in physical mode, effective address translation works as if logical address translation were enabled. As the logical address space exceeds the physical address space, the processor truncates a number of the 31 most significant bits in the logical address before referring to memory. The number of bits truncated depends upon the amount of physical memory available. The maximum length of the word address formed from this procedure is 25 bits for 64 megabytes of physical memory.

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– 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.

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.

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.

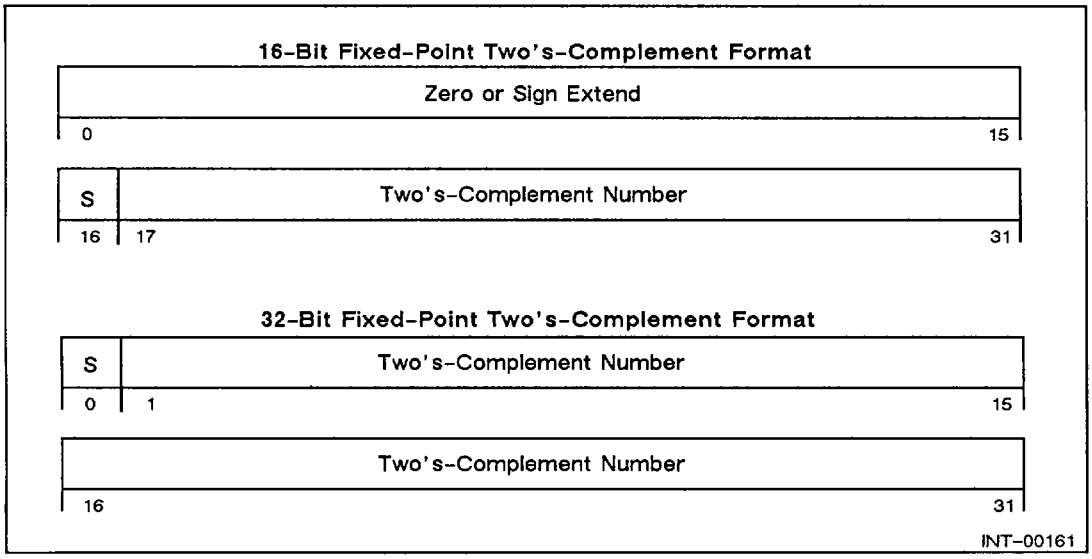


Figure 2-1 *Fixed-point two's-complement data formats*

In Figure 2-1,

Zero or Sign Extend	The zero or sign-extend bits contain 16 zeros or 16 ones. For moving and computing narrow data (depending on the instruction), the processor sign-extends narrow data when loading it into an accumulator. The processor sign-extends narrow data before or after narrow data operations, when converting it to wide data.
S	The S bit equals 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 2-1 shows the precision of 16- and 32-bit two's complement (signed) binary numbers.

Table 2-1 *Range of 16- and 32-bit fixed-point numbers*

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 2-2 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 arithmetic logic unit sign extends the result.

Table 2-2 *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

Table 2-3 lists the load and store accumulator instructions.

The wide block move instruction (**WBLM**) requires an effective address in an accumulator. Use a load effective address instruction to calculate and load the effective address into an accumulator.

Table 2-3 *Fixed-point data movement instructions*

Instruction	Operation
BAM * (**)	Block add and move
BLM * (**)	Block move
ELDA *	Extended load accumulator
ELEF *	Extended load effective address
ESTA *	Extended store accumulator
LDA *	Load accumulator
LDATS	Load accumulator with double word addressed by WSP
LEF *	Load effective address
LLEF	Load effective address (long displacement)
LNLDA	Narrow load accumulator (long displacement)
LNSTA	Narrow store accumulator (long displacement)

Table 2-3 *Fixed-point data movement instructions (contd)*

Instruction	Operation
LWLDA	Wide load accumulator
LWSTA	Wide store accumulator
MOV *	Move and skip
NLDAI	Narrow load immediate
STA *	Store accumulator
STATS	Store accumulator into double word addressed by WSP
WBLM **	Wide block move
WLDAI	Wide load with wide immediate
WMOV	Wide move
WXCH	Wide exchange accumulators
XCH *	Exchange accumulators
XLEF	Load effective address
XNLDA	Narrow load accumulator
XNSTA	Narrow store accumulator
XPEF	Push effective address
XWLDA	Wide load accumulator
XWSTA	Wide store accumulator
LPEF	Push address (long displacement)

* *ECLIPSE compatible Instruction*

** *requires an effective address in an accumulator*

Arithmetic Instructions

Tables 2-4 through 2-8 list the arithmetic instructions.

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

Table 2-4 *Fixed-point addition instructions*

Instruction	Operation	Operand Value		Result
		Source	Destination	
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

Table 2-4 *Fixed-point addition instructions (contd)*

Instruction	Operation	Operand Value Source	Destination	Result
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 double-word	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 2-5 *Fixed-point subtraction instructions*

Instruction	Operation	Operand Value Source	Destination	Result
Accumulator to 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	U16	U32
SBI *	Subtract immediate	2I	U16	U16
WSBI	Wide subtract immediate	2I	U32	U32
Immediate from Memory				
LNSBI	Narrow subtract immediate	2I	S16	S16
LWSBI	Wide subtract immediate	2I	S32	S32
XNSBI	Narrow subtract immediate	2I	U16	U16
XWSBI	Wide subtract immediate	2I	U32	U32
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 2-6 *Fixed-point multiplication instructions*

Instruction	Operation	Operand Value		
		Source	Destination	Result
Accumulator to 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 2-7 *Fixed-point division instructions*

Instruction	Operation	Operand Value		
		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 double-word	S32	S32	S32/--

S16 = Signed 16-bit integer
U16 = Unsigned 16-bit integer
S32 = Signed 32-bit integer
U32 = Unsigned 32-bit integer
* *ECLIPSE compatible instruction*

Table 2-8 *Fixed-point increment or decrement value and skip instructions*

Instruction	Operation
DSZ *	Decrement and skip if zero
DSZTS	Decrement the double word 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 double word 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 double-word boundary) with the exception of **DSZTS** *and* **ISZTS**. *Note that in a multiple-CPU configuration, a performance increase may be realized if instructions which are not atomic are used in place of atomic ones, such as* **LWADI** *(Wide Add Immediate) instead of* **LWISZ**.

* *ECLIPSE compatible instruction*

Carry Operations

For fixed-point arithmetic operations, the processor maintains a carry flag (CARRY). CARRY contains a value of zero or one. For instance, if an instruction adds 16-bit data, the carry occurs from bit 16. If an instruction adds 32-bit data, the carry occurs from bit 0.

The value of CARRY can be initialized before a binary operation by executing an explicit carry instruction. Table 2-9 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 MV/Family arithmetic instruction or an ECLIPSE compatible fixed-point instruction. For an MV/Family arithmetic instruction, the processor loads the result of carry into CARRY; it is not relative to its former value (as it is with an ECLIPSE compatible instruction). For an ECLIPSE compatible instruction, the processor complements the CARRY flag during

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

Table 2-9 *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

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.

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

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

- With an 8-bit number equal to zero, no shifting occurs.
- With an 8-bit negative number, the processor shifts from 0 to 31 bits to the right and sign-extends the vacated bit positions. The processor drops the bits shifted from the least significant bit position.

  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 ECLIPSE compatible arithmetic 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 significantbyte.

  The processor preserves the state of the CARRY flag.

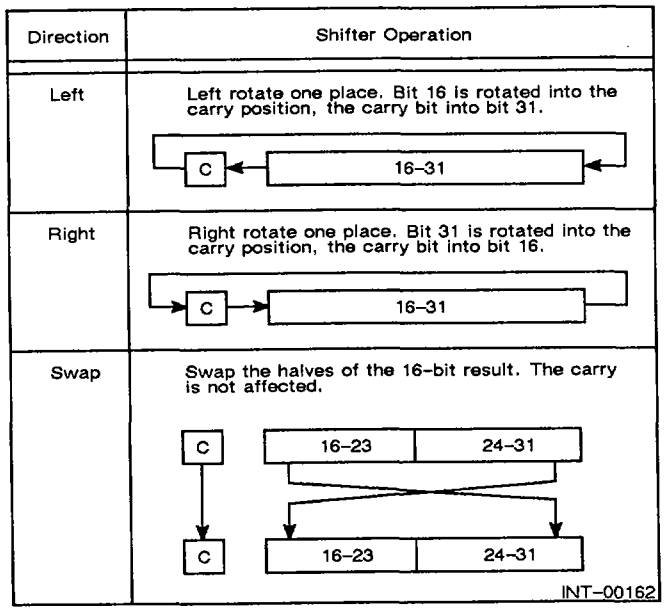


Figure 2-2 *ECLIPSE compatible shift operations*

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 **ADD**), 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**).

Table 2-10 lists the fixed-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.

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

Table 2-10 *Fixed-point skip on condition instructions*

Instruction	Operation
ADC *	Add complement with optional skip
ADD *	Add with optional skip
AND *	AND with optional CARRY initialization
CLM *	Compare to limits
COM *	One's complement with optional CARRY initialization
INC *	Increment with optional skip
MOV *	Move with optional skip
NEG *	Negate with optional CARRY initialization
NSALA	Narrow skip on all bits set in accumulator
NSALM	Narrow skip on all bits set in memory location
NSANA	Narrow skip on any bit set in accumulator
NSANM	Narrow skip on any bit set in memory location
SGE *	Skip if ACS greater than or equal to ACD
SGT *	Skip if ACS greater than ACD
SNB *	Skip on nonzero bit
SNOVR	Skip on OVR reset
SUB *	Subtract with optional skip
SZB *	Skip on zero bit
SZBO *	Skip on zero bit and set to one
WCLM	Wide compare to limits and skip
WSALA	Wide skip on all bits set in accumulator
WSALM	Wide skip on all bits set in double word memory location
WSANA	Wide skip on any bit set in accumulator
WSANM	Wide skip on any bit set in double word memory location
WSEQ	Wide skip if ACS equal to ACD
WSEQI	Wide skip if equal to immediate
WSGE	Wide signed skip if ACS greater than or equal to ACD
WSGT	Wide signed skip if ACS greater than ACD
WSGTI	Wide skip if AC greater than immediate
WSKBO	Wide skip on AC bit set to one
WSKBZ	Wide skip on AC bit set to zero
WSLE	Wide signed skip if ACS less than or equal to ACD
WSLEI	Wide skip if AC less than or equal to immediate
WSLT	Wide signed skip if ACS less than ACD
WSNB	Wide skip on nonzero bit
WSNE	Wide skip if ACS not equal to ACD
WSNEI	Wide skip if AC not equal to immediate
WSZB	Wide skip on zero bit
WSZBO	Wide skip on zero bit and set bit to one
WUGTI	Wide unsigned skip if AC greater than immediate
WULEI	Wide unsigned skip if AC less than or equal to immediate
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 overflow flag (OVR) to one. OVR remains set until cleared by another instruction. The processor status register contains the OVR flag. 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 section for a list of the instructions that test the register contents. Table 2-11 lists the instructions that manipulate the register contents.

Table 2-11 *PSR manipulation instructions*

Instruction	Operation
BKPT	Breakpoint
FXTD	Disable fixed-point trap (resets OVK and disables trap)
FXTE	Enable fixed-point trap (sets OVK and enables trap)
LCALL	Call subroutine
LPSR	Load PSR into AC0
PBX	Pop block and execute
SNOVR	Skip on OVR reset
SPSR	Store PSR from AC0
WPOPB	Wide pop block
WRSTR	Wide restore
WDPOP	Wide pop context block
WRTN	Wide return
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
XVCT	Vector I/O interrupt

Figure 2-3 shows the format of the processor status register.

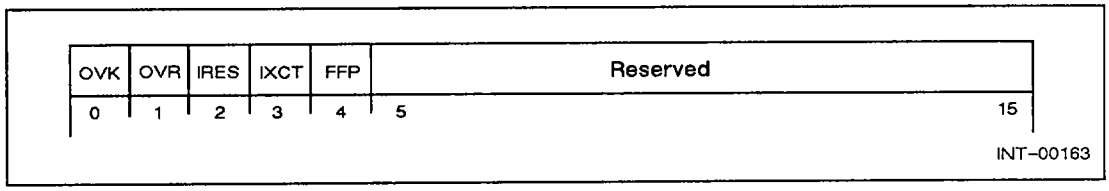


Figure 2-3 *Processor status register format*

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

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

In Figure 2-3,

OVK — The OVK bit is an overflow mask. To enable fixed-point overflow detection and servicing, set the OVK mask to one. You can set the OVK mask to one with the **FXTE**, **SPSR**, **WSAVS**, and **WSSVS** instructions (Table 2-11).

The processor saves or restores the status of the OVK mask when going to or returning from a subroutine or fault handler. For the processor to detect and service an overflow fault, the OVK mask must be set to one before the processor sets the OVR flag to one.

OVR

The OVR bit is an *overflow* flag.
The processor sets the OVR flag to one 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 the OVR flag before completing the instruction.
The OVR flag remains set to one until any of the following events occur:

- I/O interrupt request acknowledged

  Refer to the chapter "Device Management" for additional details.

- Fault detection and servicing

  Refer to the chapter "Program Flow" for additional details.

- Power up, I/O reset, or system reset

- Processor executes an instruction listed in Table 2-11

IRES

The IRES bit is an interrupt resume flag.
The processor sets the IRES flag 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 the IRES flag and saves the microstate on the user stack.
When a resumable instruction begins execution, it first tests the IRES flag. If the flag is 0, the instruction begins an initial execution. If the flag is 1, the instruction restores the state, resets the IRES flag to 0, and resumes execution.

NOTE: *Although the processor can interrupt some instructions, implementations may choose to run them through completion. Refer to a machine-specific supplement for additional information.*

IXCT

The IXCT bit 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 the IXCT flag in the PSR and pushes the opcode of the saved instruction onto 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.

FFP — The FFP bit 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. The FFP bit is applicable to systems with parallel floating-point units. The FFP bit 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 (FPU) 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 bit 4 (FFP) of the PSR to reflect the current value of the TE bit in the FPSR. The processor then clears the TE bit of FPSR (if the FFP bit in the PSR 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 bit and clears the PSR FFP bit. 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.

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**) logically OR's two 16-bit binary values; a wide inclusive OR instruction (**WIOR**) logically OR's two 32-bit binary values. Figure 2-4 shows the 16- and 32-bit formats.

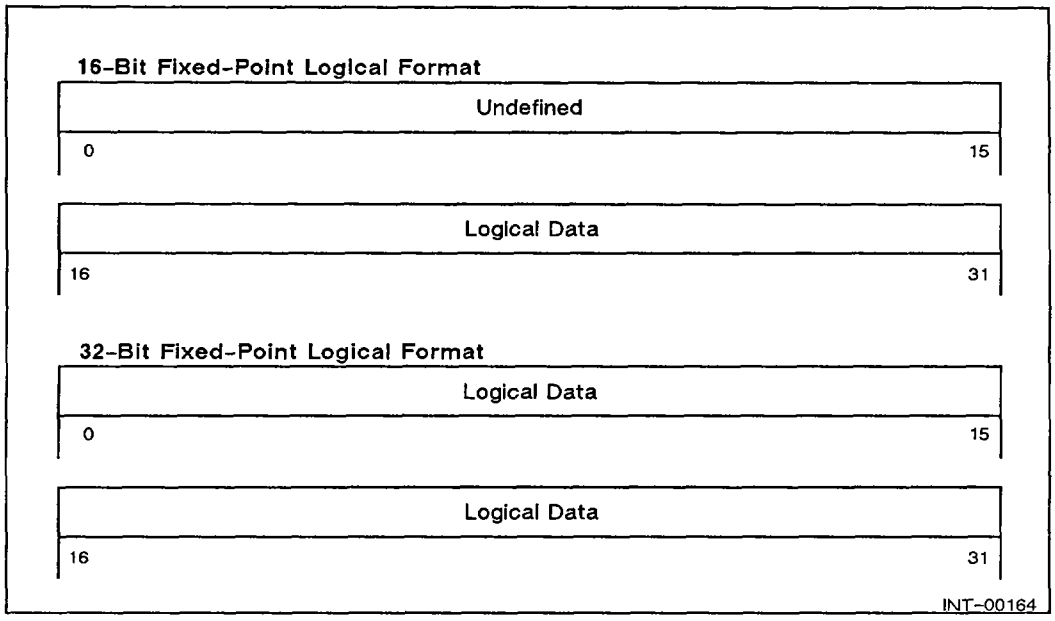


Figure 2-4 *Fixed-point logical data formats*

Logic Instructions

Table 2-12 lists the logical instructions. 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 2-12 *Logical Instructions*

Instruction	Operation
ANC *	AND with one's complemented source
AND *	AND
ANDI *	AND immediate
COB *	Count bits
COM *	Complement (one's complement)
IOR *	Inclusive OR
IORI *	Inclusive OR immediate
LOB *	Locate lead bit
LRB *	Locate and reset lead bit
WANC	Wide AND with one's complemented source
WAND	Wide AND
WANDI	Wide AND immediate
WBTO **	Wide set bit to one
WBTZ **	Wide set bit to zero
WCOB	Wide count bits
WCOM	Wide complement (one's-complement)
WIOR	Wide inclusive OR
WIORI	Wide inclusive OR immediate
WLOB	Wide locate lead bit
WLRB	Wide locate and reset lead bit
WXOR	Wide exclusive OR
WXORI	Wide exclusive OR immediate
XOR *	Exclusive OR
XORI	Exclusive OR immediate

* *ECLIPSE compatible instruction*

** *requires an effective address in an accumulator*

Shift Instructions

Table 2-13 lists the logical shift instructions. The processor can shift an intermediate result as explained for the **ADC, ADD, INC, MOV, NEG** and **SUB** instructions.

Table 2-13 *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
MOV *	Move
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 2-14 lists the logical 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.

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

Table 2-14 *Fixed-point logical skip instructions*

Instruction	Operation
ADC *	Add complement with optional skip
AND *	AND with optional skip
COM *	One's complement with optional skip
NEG *	Negate with optional skip
SNB *	Skip on nonzero bit
SZB *	Skip on zero bit
SZBO *	Skip on zero bit and set to one
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*

A wide skip on bit instruction (**WSNB, WSZB,** and **WSZBO**) requires an effective address in an accumulator. A load effective address instruction may be used (**LLEF** or **XLEF**) to calculate and to load the effective address into an accumulator.

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.

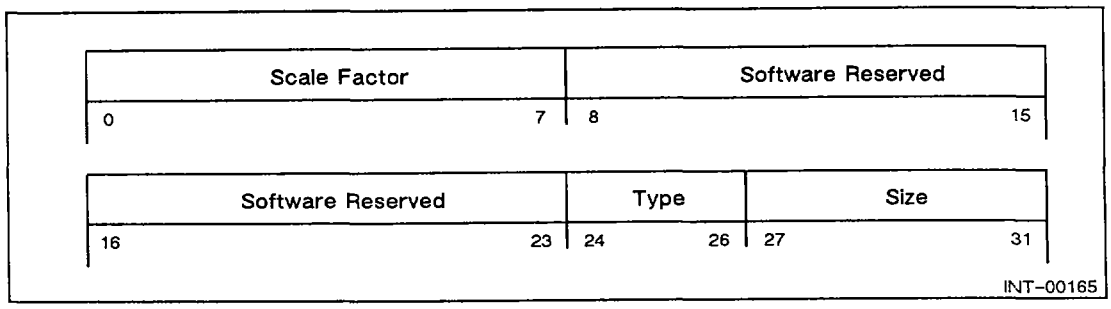


Figure 2-5 *Explicit data type indicator*

In Figure 2-5,

Reserved — The reserved field indicates that DGC reserves bits 8-23 for future use.

Scale Factor — The scale factor determines how the decimal integer in memory will be interpreted by some instructions. Several decimal instructions ignore this field.

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;
.00932,   if sf =  5.
```

Type — The type field identifies the type of data, as listed in Table 2-15.

Size — 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 commercial 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 2-15 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 2-17 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 2-16 for a list of the sign-positioned ASCII characters. Table 2-17 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 (or D_{16}) for negative (-).

Table 2-15 on the following page gives examples of eight commercial data types.

Table 2-15 *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

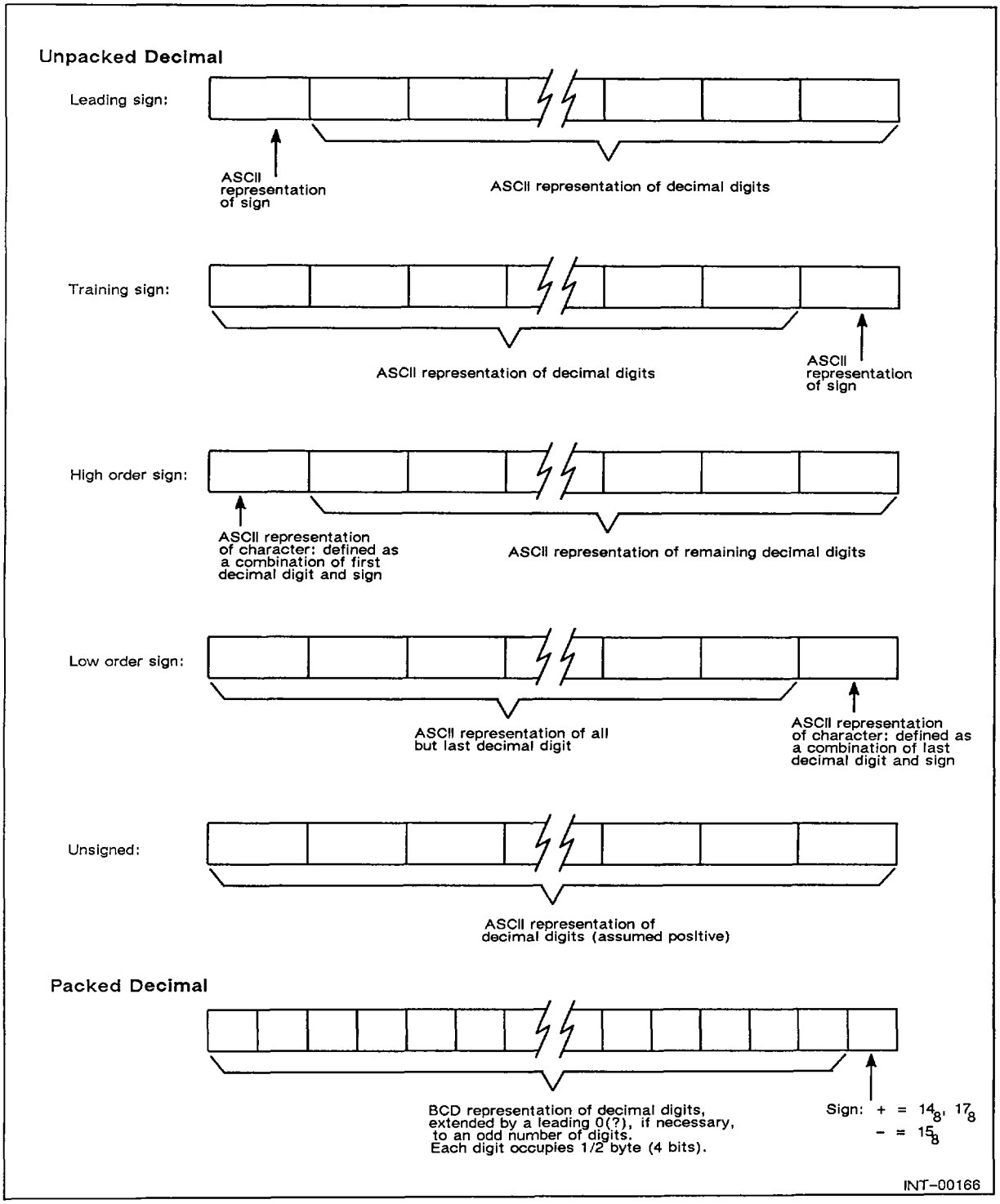


Figure 2-6 *Packed and unpacked decimal data*

Table 2-16 *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 2-17 *Nonsign-positioned numbers for unpacked decimal*

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

Move Instructions

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

Table 2-18 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 addressed byte, but the other byte in the memory word remains intact.

Table 2-18 *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

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

Table 2-19 lists the move and convert decimal/floating-point instructions.

Table 2-19 *Fixed-point to floating-point conversion and store instructions*

Instruction	Operation
LDI, WLDI	Convert a decimal and load into FPAC
LDIX, WLDIX	Convert a decimal, extend 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

Move instructions require an effective word address and/or an effective byte address. Table 2-23 lists the instructions that calculate the address and store it in a fixed-point accumulator.

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 2-20 lists the edit subprogram instructions.

Table 2-20 *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 *character-a* or *character-b* depending on sign flag
DINT	Insert *character-a* or *character-b* depending on trigger
DMVA	Move *j* alphabetical characters
DMVC	Move *j* characters
DMVF	Move *j* float
DMVN	Move *j* 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 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 2–21 lists the arithmetic instructions.

Table 2-21 *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 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 2–22 list the hex shift instructions.

Table 2–22 *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 2–23) 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 2–23 *Load effective byte address instructions*

Instruction	Operation
ELEF *	Extended load effective address
LEF *	Load effective address
LLEF	Load effective address (long displacement)
LLEFB	Load effective byte address
LPEF	Push address
LPEFB	Push byte address
WMOVR	Wide move right (convert byte pointer to word pointer)
XLEF	Load effective address
XLEFB	Load effective byte address
XPEF	Push effective address
XPEFB	Push effective byte address

* *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.

CAUTION: *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 does service the fault before executing another instruction.

Table 2-24 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 2-24 *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,	Number too large to convert to specified data type. |number|> (10^{16}) - 1
			WSTIX, WLDIX	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	

Decimal Arithmetic Example

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

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

          .LOC  CONSOLE+?IRCL
          -1
          180.
          .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                        ;INIT FOR DATA TYPE 4
          XNLDA  3.CONSOLE+?IBAD               ;GET BYTEPTR FROM CONSOLE PKT
          WLDI   0
          FAS    0,0                           ;SINGLE PRECISION FLOATING-POINT ADD
          XNLDA  1.FCON
          XNLDA  3.CONSOLE+?IBAD
  AGAIN:  WSTI   0
          INC    1,1,SZC                       ;INC 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*

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 (move, arithmetic, and skip), the Intrinsic Instruction Set, and the floating–point status register (FPSR).

Data Formats

Floating-point arithmetic and intrinsic instructions require normalized, sign magnitude numbers. You can use the Floating-point Normalize (FNOM) instruction to normalize raw floating-point data, which may or may not be normalized.

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

Single-precision numbers are most efficient when used with even-word boundaries, and double-precision numbers are most efficient when used with double-word 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.

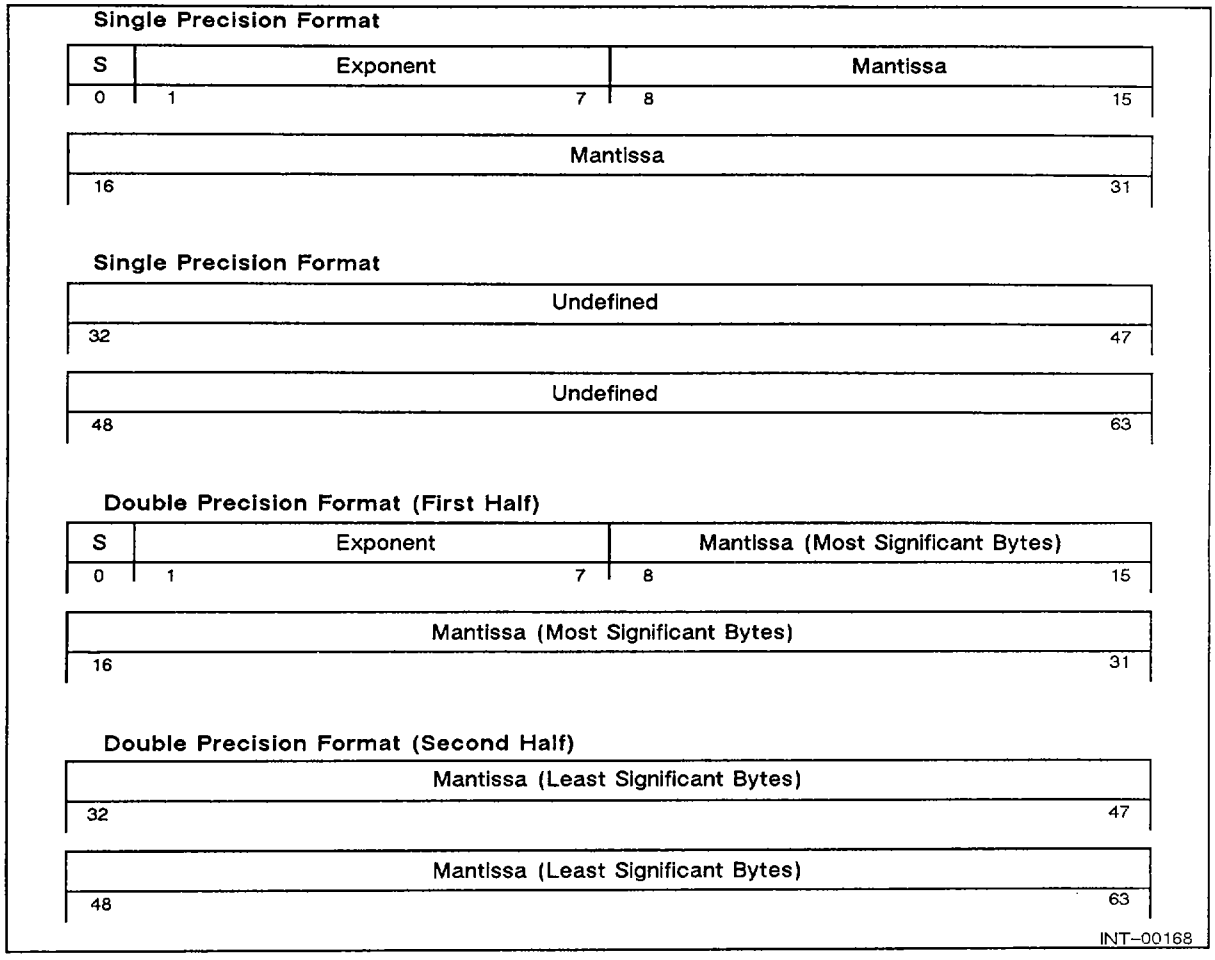


Figure 3-1 *Floating-point data formats*

In Figure 3-1,

S — The S bit equals the sign bit of the mantissa. The sign bit equals a zero for a positive number, and equals a one 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

$$1/16 \text{ to } 1-(2^{-24})$$

the range for double-precision is

$$1/16 \text{ to } 1-(2^{-56})$$

Table 3-1 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 3-1 *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)
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 3-2 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 3-2 *Floating-point decimal conversion instructions*

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

Move Instructions

All single-precision operations that specify an accumulator fetch the most significant 32 bits of the floating-point accumulator and ignore the least significant 32 bits. Upon completion of the specified operation, the processor returns the result to the most significant portion of the floating-point accumulator. The processor loads the least significant 32 bits of the floating-point accumulator with zeros. Table 3-3 lists the load and store floating-point accumulator instructions.

Table 3-3 *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
LFLDS	Load floating-point single
LFSTD	Store floating-point double
LFSTS	Store floating-point single
WFPOP	Wide floating-point pop
WFPSH	Wide floating-point push
XFLDD	Load floating-point double
XFLDS	Load floating-point single
XFSTD	Store floating-point double
XFSTS	Store floating-point single

* *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.

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

Appending Guard Digits

The processor appends the one or two guard digits to the least significant hex digit of both mantissas, 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.

NOTE: *The floating-point conversion and single-precision store instructions (*FINT, FSCAL, LFSTS, WFFAD, WFLAD, *and* XFSTS*) ignore the RND flag. Refer to the individual instruction description in the chapter "Instruction Dictionary" for further information.*

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.

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 bitof 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_{16} 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 one 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 out of 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 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 3-4 lists the floating-point add instructions.

Table 3-4 *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)
LFAMS	Add single (memory to FPAC)
XFAMD	Add double (memory to FPAC)
XFAMS	Add single (memory to FPAC)

* *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 is different from the destination accumulator (facs, facd). Table 3-5 lists the floating-point subtract instructions.

Table 3-5 *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)
LFSMS	Subtract single (memory from FPAC)
XFSMD	Subtract double (memory from FPAC)
XFSMS	Subtract single (memory from FPAC)

* *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 3-6 lists the floating-point multiplication instructions.

Table 3-6 *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)
LFMMS	Multiply single (FPAC by memory)
XFMMD	Multiply double (FPAC by memory)
XFMMS	Multiply single (FPAC by memory)

* *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 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 one 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 3-7 lists the floating-point divide instructions.

Table 3-7 *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)
LFDMS	Divide single (FPAC by memory)
XFDMD	Divide double (FPAC by memory)
XFDMS	Divide single (FPAC by memory)

* *ECLIPSE compatible instruction*

Skip Instructions

A skip instruction tests the result of an operation for a specific condition and (except for FCMP) directs the processor 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 reflecting the relationship. You can then use the FSGT, FSEQ, and FSLT skip instructions to test the status flags.

Table 3-8 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.

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

Table 3-8 *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 and Z = 1)
FSLT *	Skip on less than zero (N = 1)
FSND *	Skip on no zero divide (DVZ = 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 DVZ = 0)
FSNU *	Skip on no underflow (UNF = 0)
FSNUD *	Skip on no underflow and no zero divide (UNF and DVZ = 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 argument to be operated on is in FPAC0 and the answer will be returned to FPAC0. For two-argument instructions (such as WFATN2 and WFPWR), FPAC1 is used as 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.

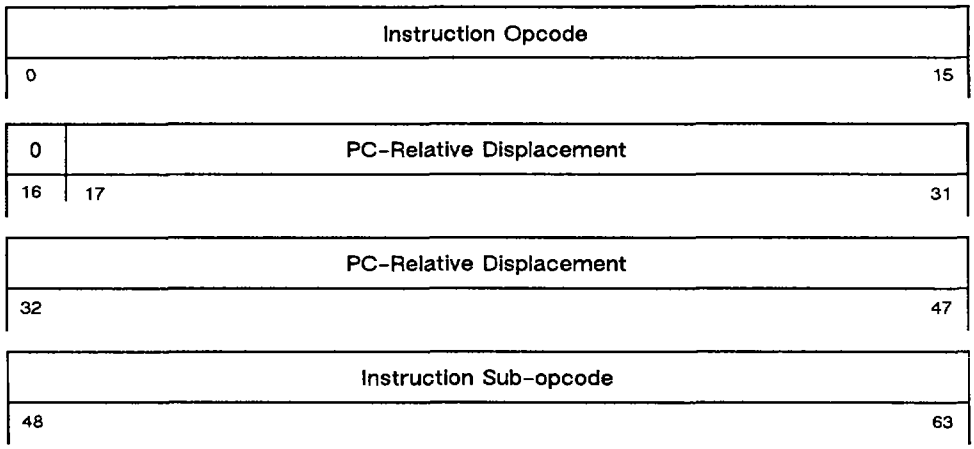
IIS instructions always update the Z and N flags of the FPSR so that they reflect the result -- either zero or negative.

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

- 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 erroroccurred. For example, a negative value input to a square root (WFSQR) instruction causes an invalid input argument trap with the error code two returned to the INP bits. The FPSR description in the section "Faults and Status" of this chapter explains the various codes and their meanings.

- 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 IIS instructions have a displacement coded with each instruction. The format for these four-word instructions is:



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, non-indirectable address of a routine in an optional run time library in the current ring that does emulate the function of the instruction if hardware support does not exist.

All IIS instructions are interruptible. Table 3-9 lists the intrinsic instructions.

Table 3-9 *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 and for the mantissa status after executing a floating-point instruction. The processor stores the result in a 64-bit floating-point status register. When the processor detects a floating-point fault, it sets the appropriate floating-point status register bits. FPSR bits are accumulative 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 floating-point status register). If the TE mask equals zero, the processor continues normal program execution with the next sequential instruction. Program flow remains unchanged. If the TE mask equals one, 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 fault handling.

NOTE: **FPSH, FSST, LFSST** *and* **WFPSH** *instructions store the FPSR, however, they will not store a value with any combination of bit 5 and bits 1 to 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 3-10 lists the instructions that initialize the register and that store or load the register contents.

Table 3-10 *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 faults (ANY, OVF, UNF, INV, MOF, and TE), mantissa status (Z and N), rounding (RND), floating-point identification (ID), floating-point operation (PAR), invalid input argument indicator (INP), and the floating-point program counter (FPPC). Figure 3-2 shows the format of the floating-point status register.

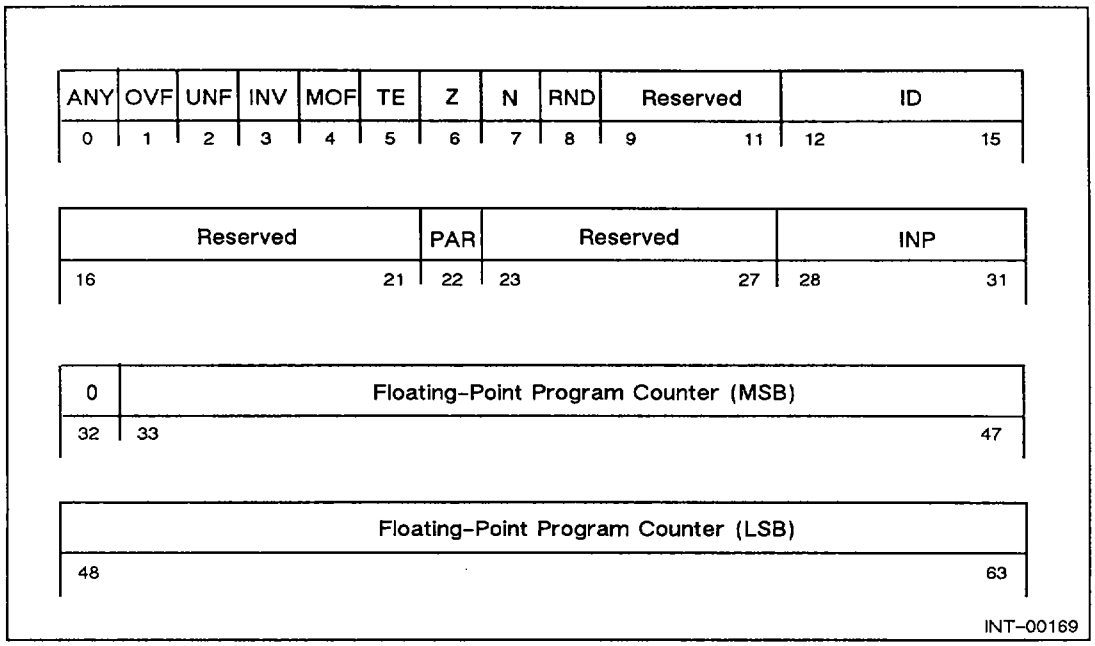


Figure 3-2 *Floating-point status register format*

In Figure 3-2,

Bit	Name	Contents or Function
0	ANY	Error status flag. The processor sets the ANY flag to one when it sets either the OVF, UNF, INV, or MOF flag to one. ANY is never set from memory, but is always recomputed from these bits.
1	OVF	Exponent overflow flag. The processor sets the OVF flag while executing a floating-point instruction, and an exponent overflow occurs. The result is correct except the exponent is 128 too small.
2	UNF	Exponent underflow flag. The processor sets the UNF flag while executing a floating-point instruction, and an exponent underflow occurs. The result is correct except the exponent is 128 too large.
3	INV	Input argument error flag. The processor sets the INV flag while attempting to execute an instruction with an invalid argument as input. If the INV bit is one, the INP bits further define 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.*
4	MOF	Mantissa overflow flag. The processor sets the MOF flag 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 a **FFAS**, **FFMD**, or **WFFAD** instruction, the result is too large and the processor truncates the result before storing it.

5 TE Trap enable mask.

The processor or you enable floating-point fault detection and servicing by setting the TE mask to one. The TE mask can be set to one with the **FTE** instruction.

Unless your system runs with a parallel floating-point unit, the processor does not save or restore the status of the TE mask 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 the TE mask is set to one before the processor sets the ANY flag to one. If TE is set to one, a one in any of bits 1 through 4 results in a floating-point trap, except where noted.

6 Z True zero flag.

The processor sets the Z flag if the result of executing a floating-point instruction produces a true zero.

7 N Negative flag.

The processor sets the N flag if the result of executing a floating-point instruction produces a value less than zero.

8 RND Round flag.

You set the RND flag (with the **LFLST** and **WFPOP** instructions) to direct the processor to round (RND = 1) or to truncate (RND = 0) the intermediate result of executing a floating-point instruction.

9-11 Reserved These reserved bits are ignored and returned as zero.

12-15 ID Floating-point identification code that reflects the floating-point revision.

16-21 Reserved These reserved bits are ignored and returned as zero.

22 PAR 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 the PAR bit is one, the floating-point unit operation is serial; if the PAR bit is zero, the floating-point unit operation is parallel.

23-27 Reserved These reserved bits are ignored and returned as zero.

28-31 INP INP bits contain an indicator for an invalid input argument. The definition of the INP bits depends on the setting of the invalid input argument (INV) bit.

If the INV bit is zero, the INP bits are undefined.

If the INV bit is one, the value contained in the INP bits 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 defined values are listed below.

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.
1	**WFLOGS, WFLOGD WFLG2S, WFLG2D WFLNGS, WFLNGD**	FPAC0 contains a value less than or equal to zero.
2	**WFSQRS, WFSQRD**	FPAC0 contains a value less than 0.

Code (octal)	Instruction	Description
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.
6	**WFTANS, WFTAND**	Special overflow for WFTAN (numbers in FPAC0, which are odd integer multiples of values near PI/2, will cause an overflow; even integer multiples of PI/2 return AC0)
7	**WFATN2S, WFATN2D**	The number in FPAC1 equals 0.

NOTES: *If floating-point traps are disabled (TE bit equals 0), more than one invalid input argument error may occur before floating-point traps are again enabled. In this case, the INP bits will contain the error code for the FIRST instruction which caused an invalid input argument error.*

When the processor detects an invalid input error, the state of FPAC0 is undetermined, the contents of the remaining floating-point accumulators are unmodified, and CARRY and overflow are unchanged.

32 0 Bit 32 is processor specific.

Floating-Point Program Counter: 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**) unless specifically set by a Load Floating-Point Status instruction. FPPC is undefined if ANY=0.

NOTE: *All reserved bits in the FPSR must be zero. The OVF, UNF, INV, and MOF status bits are cumulative. These bits are only cleared by FPSR loads which restore them, or the explicit clear instructions (such as* **FCLE***).*

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 double words, 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 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 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).

Figure 4-1 shows the four stack parameters. Items (1) and (2) identify the lower and upper stack limits, 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.

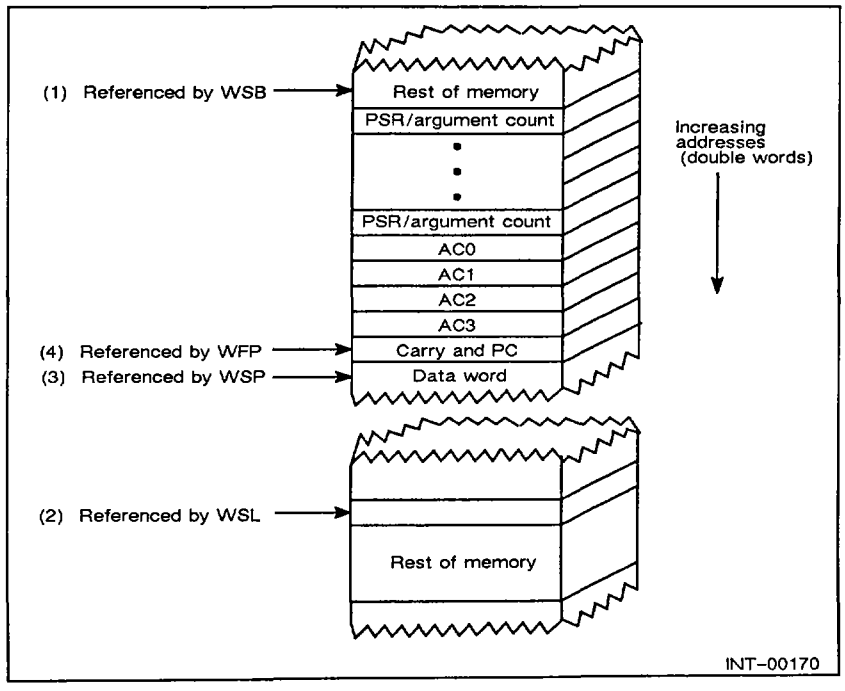


Figure 4-1 *Typical wide stack*

Wide Stack Registers

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

The processor accesses the stack management registers to save or restore them when changing program flow between segments. Figure 4-2 shows the format of these registers.

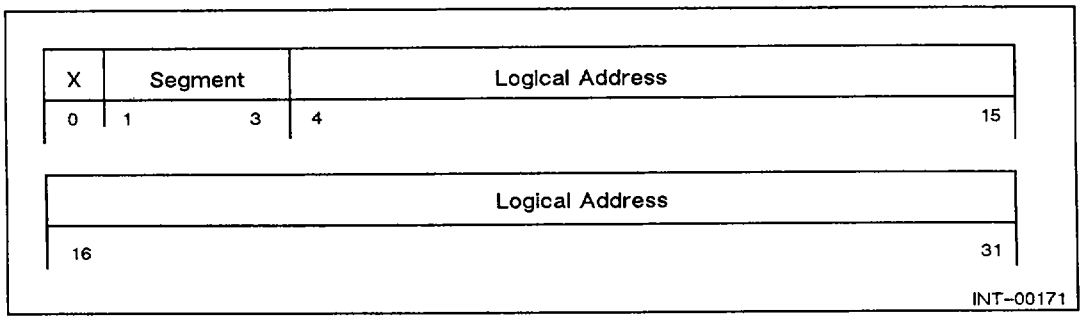


Figure 4-2 *Wide stack management register format*

In Figure 4-2,

x	Bit 0 must be zero, and in the stack limit is set to one when a stack fault occurs.
Segment	Bits 1-3 specify the segment location of the stack.
Logical Address	Bits 4-31 specify 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 double word below the actual address of the first double word 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 16 double words below the actual address of the last double word in the wide stack, so that there will be space for handling a stack overflow if one occurs.

The processor pushes one or more double words 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, or WSSVS) and 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 double-word placed on the stack (when adding data to the stack) or the next double-word 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 double word, the processor increments the wide stack pointer by two and stores a double word onto the stack. A pop operation retrieves one or more double words from the wide stack and decrements the wide stack pointer by two for each double word 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. 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 4-1 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, when the processor transfers program control to another segment, it initializes all four wide stack registers (from reserved memory in page zero of the new segment).

Wide Stack Data Instructions

The wide stack data instructions access a double word or a block of double words. All the wide stack data instructions increment or decrement the wide stack pointer. Instructions that access a double word modify the wide stack pointer by two. Instructions that access a block of double words modify the wide stack pointer by four or more (depending upon the size of the data block or return block). The instructions in Table 4-2 access a double word or a block of double words.

Table 4-1 *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)

Table 4-2 *Wide stack double-word access instructions*

Instruction	Operation
DSZTS *	Decrement the double word addressed by WSP (skip if zero)
ISZTS *	Increment the double word addressed by WSP (skip if zero)
LDATS *	Load accumulator with double word 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 double word 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 4–3 push or pop a return block. Although the return block can take several forms, it usually consists of six double words. (See Table 4–4.)

Table 4–3 *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 4–4 *Standard wide return block*

Double Word Number in Block Block Pushed	Double Word Number in Block Popped	Contents
1	6	PSR (bits 0–15) All zeros or an argument count from LCALL or XCALL (bits 16–31)
2	5	AC0
3	4	AC1
4	3	AC2
5	2	AC3 = Old WFP
6	1	Bit 0 = CARRY; Bits 1–31 = PC return address

The chapter "Instruction Dictionary" presents the subroutine return block with a 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} . The processor detects a stack overflow 16 double words before the actual end of the stack.

```
        .NREL
BASE:   .BLK    66.             ;Reserve 66 words for the wide stack
ENDZ:   .BLK    32.             ;Reserve 32 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
        .
        .
```

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 **WPSH** instruction calculates and pushes a byte address onto the stack.

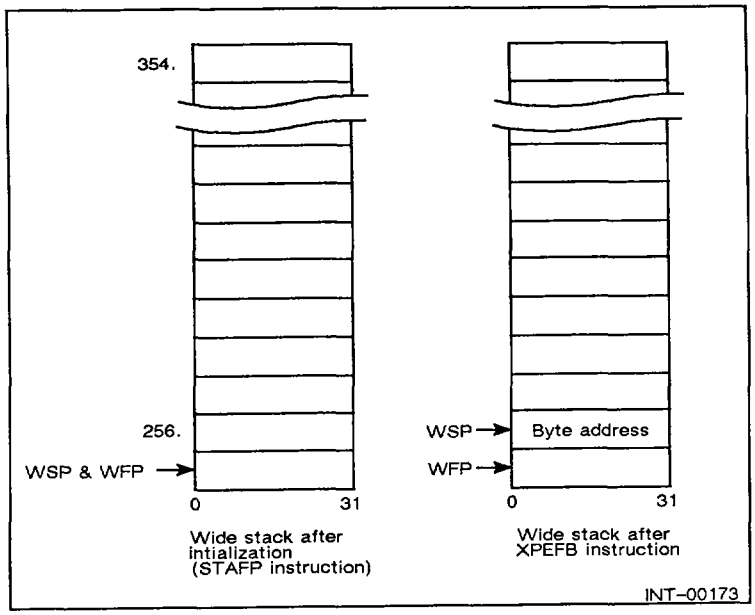


Figure 4–4 *Example of wide stack operations*

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.

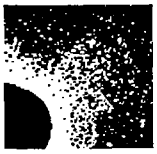
After pushing data onto the stack, the processor checks for a stack overflow by comparing the value of the wide stack pointer to the value of the wide stack limit. If the value of the wide stack pointer is greater, a stack overflow exists. Loading the value 37777777777_8 into the wide stack limit register 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 4-5 lists the instructions that push or pop one or more double words onto the wide stack. Table 4-5 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 4-5 *Instructions affecting the wide stack*

Instruction	Description	Double Words Pushed or (Popped)	Double Words Required Beyond WSL for Stack Fault
BKPT	Breakpoint handler	6	12
LCALL	Subroutine call	1	7
LFAMD, etc.	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.	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



5

Program Flow Management

This chapter explains program flow, related instruction groups, transferring program control to another segment, and handling faults.

The program counter specifies the logical address of the instruction to execute. Thus, it controls the execution sequence of instructions. Address wraparound occurs within the current segment as only bits 4 through 31 are used to increment the program counter.

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

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 **XCT**, jump, skip, and subroutine call or return instruction. Refer to the chapter "Device Management" for I/O interrupt processing.

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 location after the **XCT** instruction, if the contents of the accumulator is the first of a two-word instruction.
- The effective address, if the accumulator contains 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 jumps the first word after the skip instruction and executes the second word as an 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 first two lines of code 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 last three lines of code in Figure 5–1 perform the skip properly.

```
        FSEQ                ;Skip on zero
        LFDMD  0.@OPAND     ;Floating-point divide with a two-word displacment
        .
        .
        .
        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.

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

1. First instruction in the DO-loop sequence when the incremented variable equals (or remains less than) the value.

2. Instruction following the DO-loop sequence when the incremented variable becomes greater than the value.

   The processor skips the DO-loop sequence of instructions by

   1. Adding one and the termination offset (for skipping the DO-loop sequence) to the program counter value.

   2. Loading the 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
LOOP:   NLDAI 5,0               ;Maximum index value
        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**) skips one, two, or three locations when an explicit queue element exists. 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 (Tables 2-9 and 2-10).
- "Floating-Point Computing" for the floating-point skip instructions (Table 3-8).
- "Device Management" for the I/O skip instructions (Tables 7-2 and 7-4).

Subroutine

A subroutine call sequence (except **WEDIT**) pushes a wide return block onto the wide stack and loads the effective address into the program counter. Program flow continues with the effective address in the program counter. (The **WEDIT** instruction transfers control to an edit subprogram without changing the program counter.)

NOTE: *To pass arguments to the subroutine, push the arguments onto the stack before jumping to (***LJSR** *or* **XJSR***) or calling (***LCALL** *or* **XCALL***) the subroutine.*

A subroutine return instruction (except **WEDIT**) pops the wide return block from the wide stack, thus, restoring the CARRY bit, program counter, and accumulators. Program flow continues with the instruction following the subroutine call. (A **WEDIT** subprogram instruction returns program control to the instruction following the **WEDIT** instruction.) Table 5-3 lists the subroutine, save, and return instructions. Table 5-4 illustrates the relationships between the various subroutine instructions.

Table 5–3 *Subroutine instructions*

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

Table 5–4 *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.*

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 a **WPOPB**;

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 (PC 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.

A jump to a subroutine (**LJSR** or **XJSR**) instruction transfers program control to a subroutine in the current segment. The **LJSR** or **XJSR** instruction stores the return address and transfers 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. 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.

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 double word 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. 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.)

Return Block

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

Table 5-5 *Standard wide return block*

Word Number in Block Pushed	Word Number in Block Popped	Contents
1	12	PSR
2	11	All zeros or an argument count from LCALL or XCALL
3-4	9-10	AC0
5-6	7-8	AC1
7-8	5-6	AC2
9-10	3-4	AC3 or old wide frame pointer
11-12	1-2	Bit 0 = CARRY flag Bits 1-31 = return address

Example With Wide Stack Operations

The following explanation 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 two 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.

```
          .
          .
          .
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 *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 PSR, AC0-AC2, old WFP, the carry bit, (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 anywhere within the segment.

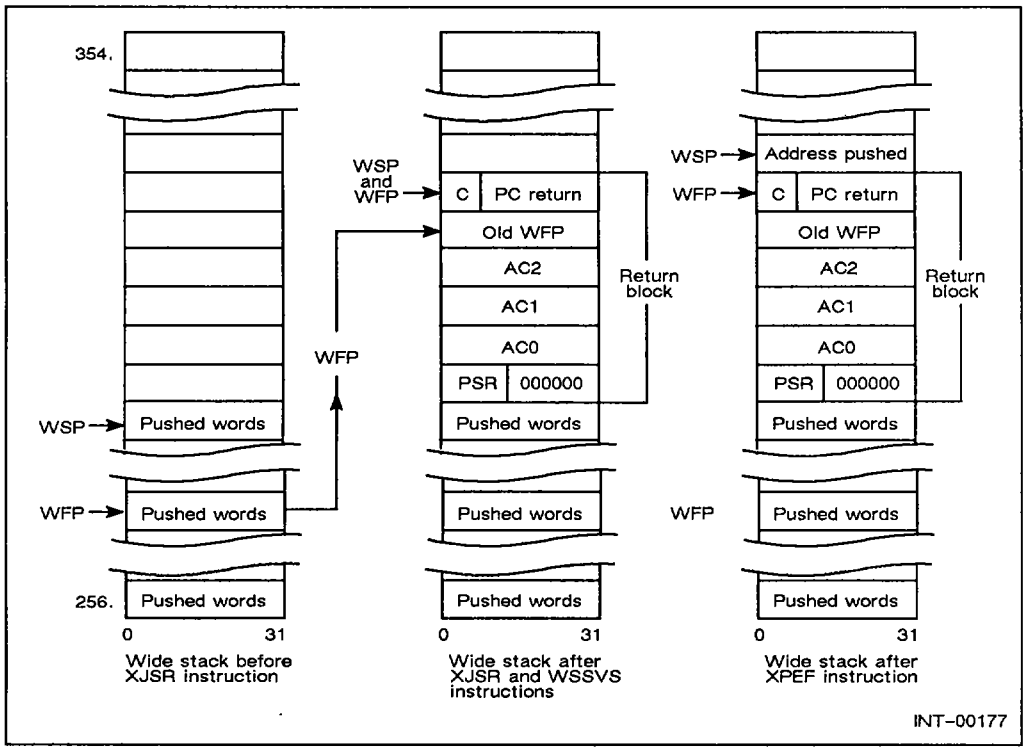


Figure 5-4 *Wide stack operations from* **XJSR** *and* **WSSVS** *instructions*

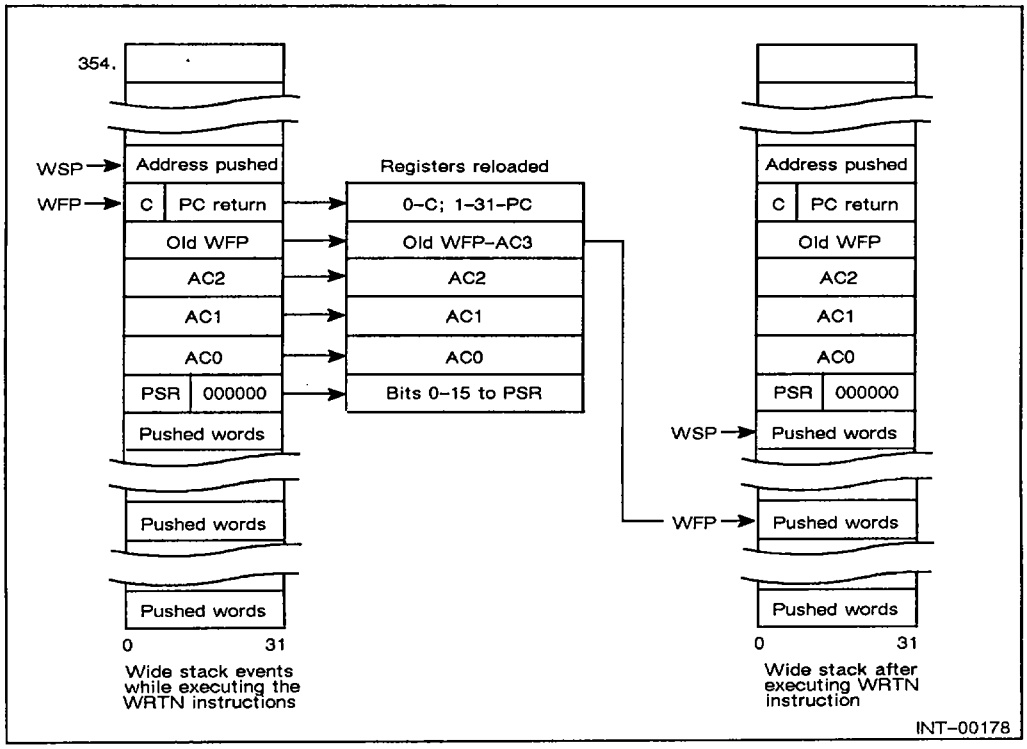


Figure 5-5 *Wide stack operations from* **WRTN** *instruction*

Transferring Program Control To Another Segment

The instructions listed in Table 5-6 transfer program control to or from another segment.

Table 5-6 *Segment 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 zero 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 source 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.

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 of the destination segment. Figure 5-6 shows the format of a gate array.

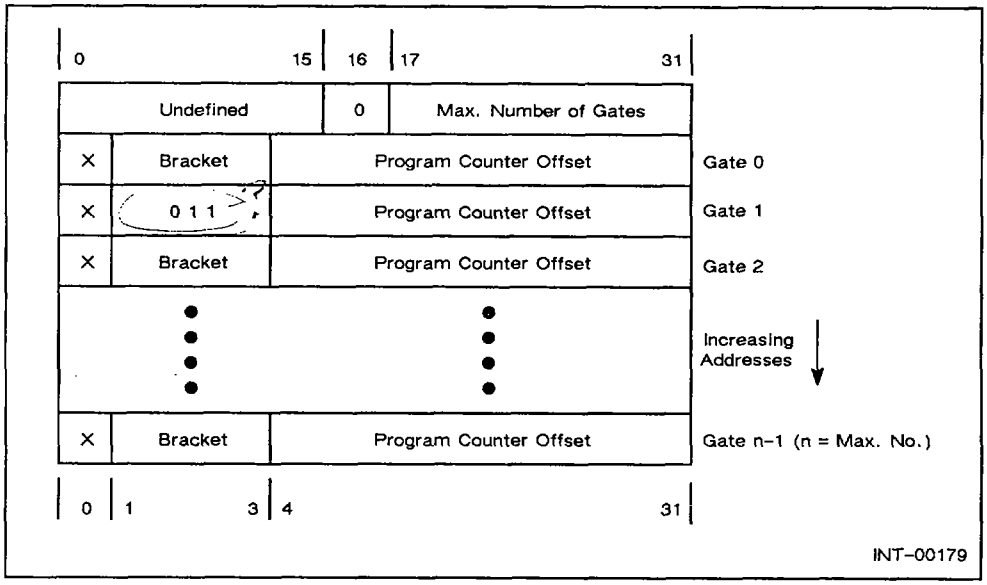


Figure 5-6 *Gate array format*

In Figure 5-6,

Undefined	The processor ignores these undefined bits.
0	Always 0 (bit 16).
Maximum Number of Gates	The maximum number of gates specifies the total number of gates. If the maximum number is zero, the destination segment cannot be the target of an inward segment crossing.
X	The processor ignores this bit.
Bracket	The bracket is the gate bracket.
	The gate bracket is 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 the 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

Figure 5-7 shows how the processor interprets the effective address of the **XCALL** or **LCALL** instruction.

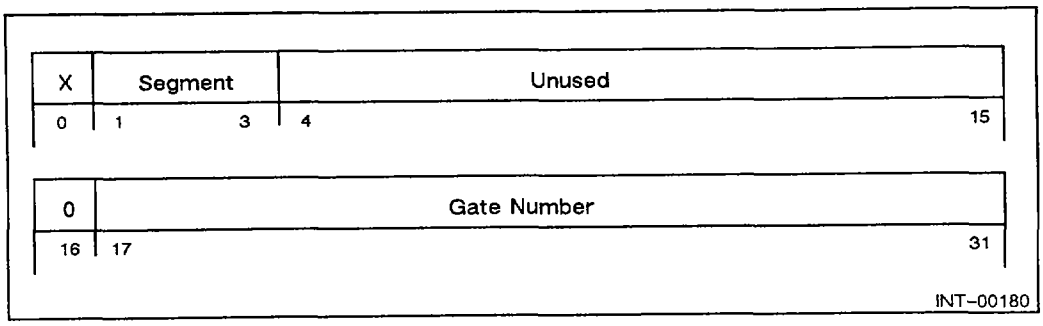


Figure 5-7 XCALL *or* LCALL *effective address*

In Figure 5–7,

X	This bit (bit 0) is ignored by the processor.
Segment	The segment bits (bits 1–3) specify the segment number of the destination segment.
Unused	The unused bits (bits 4–15) are ignored by the processor.
0	Always 0 (bit 16).
Gate Number or Call Offset	If the segment bits (1–3) specify a destination segment less than the current segment of execution, then the gate number (bits 17–31) specified 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 target offset in the current ring to which control is transferred. The processor uses the gate number as an index to an element (a gate) in the gate array.

To perform a valid inward segment crossing, 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 the next step.

   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 referencing these locations because an infinite page fault will be signaled and the processor will halt.*

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

5. Checks for a potential destination stack overflow.

   A parameter of the **LCALL** or **XCALL** instruction specifies the number of arguments to copy. The processor uses the 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.

   NOTE: *Refer to the "Instruction Dictionary" for the* **LCALL** *and* **XCALL** *argument count description.*

6. Copies the 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.*

7. Pushes a double word that contains the processor status register and the number of arguments pushed.

8. 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.

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.

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**. The processor places the contents of the wide frame pointer into the wide stack pointer. Then, the processor

1. Pops the six double-word return block.

   The processor pushed the first five double words of the return block when it executed the **WSAVR** or **WSAVS** instruction. The processor pushed the sixth double word (processor status register and the number of arguments) when it executed the **LCALL** or **XCALL** instruction.

   The processor loads the program counter with the return address in the destination segment, and checks for inward return.

2. 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.

3. 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 referencing these locations because an infinite page fault will be signaled and the processor will halt.*

4. Calculates the address of the double word that precedes the arguments of the calling sequence and loads the wide stack pointer with the double word.

5. Executes program counter from return block (which may be instruction after **LCALL** or **XCALL**).

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 5-7 lists these faults.

Table 5-7 *Faults*

Fault	Type
Nonresident page	Privileged
Protection violation	Nonprivileged
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 indirect 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 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. This manual's "Instruction Dictionary" describes the ECLIPSE instruction set and the instructions which affect the carry bit.

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). The section "Unimplemented Instructions" describes unimplemented opcodes; the chapter "Memory and System Management" describes some of the fault conditions listed in Table 5-8.

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 5-8 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 5-8 *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	Page table validity violation
5	Read, write, or execute access violation
6	Segment crossing violation
7	Unimplemented opcode or instruction

When the processor detects a protection violation (Figure 5-8), it checks the contents of reserved memory location 36_8 in the current ring for the protection fault handler. The contents of this location contain a 16-bit indirectable pointer that determines in which segment the protection fault will be serviced.

NOTE: *Page zero for the current segment and segment zero must be memory resident.*

- If this location is zero, no outer ring protection fault handler has been defined, and the processor performs the following:
- If this location is nonzero the current segment has a protection fault handler defined, and the processor performs the following:

1. Stores the contents of the wide stack pointer and the wide frame pointer into the page zero locations of the current segment.

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

2. Crosses to segment 0 (if the current segment is 1 to 7).

3. Redefines the wide stack for segment 0. The processor initializes the wide stack pointer, wide stack limit, and wide stack base registers from segment 0, page zero locations.

4. Pushes a fault return block, as shown in Table 5-9, onto the segment 0 stack.

   NOTE: *If a protection violation occurs while attempting to get the segment 0 protection fault handler (while pushing the fault return block), an infinite protection fault is generated and the system halts.*

5. Sets the PSR to zero.

6. 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 5-10 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.). See Table 5-10.

1. Pushes a fault return block, as shown in Table 5-9, onto the current segment's stack.

   NOTE: *If a protection violation occurs while attempting to get to the outer ring protection fault handler, control is transferred to the segment 0 protection fault handler.*

2. Sets the PSR to zero.

3. 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 5-10 lists the protection fault codes.

   Sets AC2 equal to the specific address (offending address) that caused the reference problem, if applicable. (Bit 0 0 is undefined.) See Table 5-10.

4. Checks for stack overflow.

   If a stack overflow occurs, the processor pushesa stack fault return block onto the stack and processes the stack fault. The stack fault return block contains the return address to the protection fault handler.

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

5. Jumps to the fault handler and executes the first instruction. The current segment's reserved memory location 368 contains the16-bit nonindirectible starting address of the protection violation fault handler.

   NOTE: *If a protection violation occurs during execution of the outer ring protection fault handlerroutine, the same outer ring protection fault routine will handle the new protection fault. Unless a protectionviolation is generated while pushing a subsequent return pushing a subsequent return block onto the stack, an infinite loop is not an infinite protection fault; it does not halt the machine.*

7. 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 return address to the protection fault fault handler address.

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

8. Jumps to the fault handler and executes the first instruction. Segment 0 reserved memory location 36_8 contains the 16-bit nonindirect starting address of the protection violation fault handler.

   One method of avoiding the above situation is to save reserved memory location 36_8, write zeroes to the location, and then restore the location to the previously saved value before returning from the protection fault handler.

Table 5-9 *Fault return block*

Double Word 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	PC (Bit 0 contains CARRY; bits 1-31 equal the PC of execution if fault type is privileged or I/O, otherwise it is undefined.)

During the servicing of a protection violation:

- If an I/O interrupt request occurs, the processor executes thefirst instruction of the protection violation fault handler before servicingthe interrupt request.
- If a protection violation fault occurs while handling any othertype of nonprivileged fault, the processor aborts the first fault and processesthe protection violation fault. The return block pushed onto the stack for theprotection violation fault is undefined, as are the contents of AC0 and AC1.

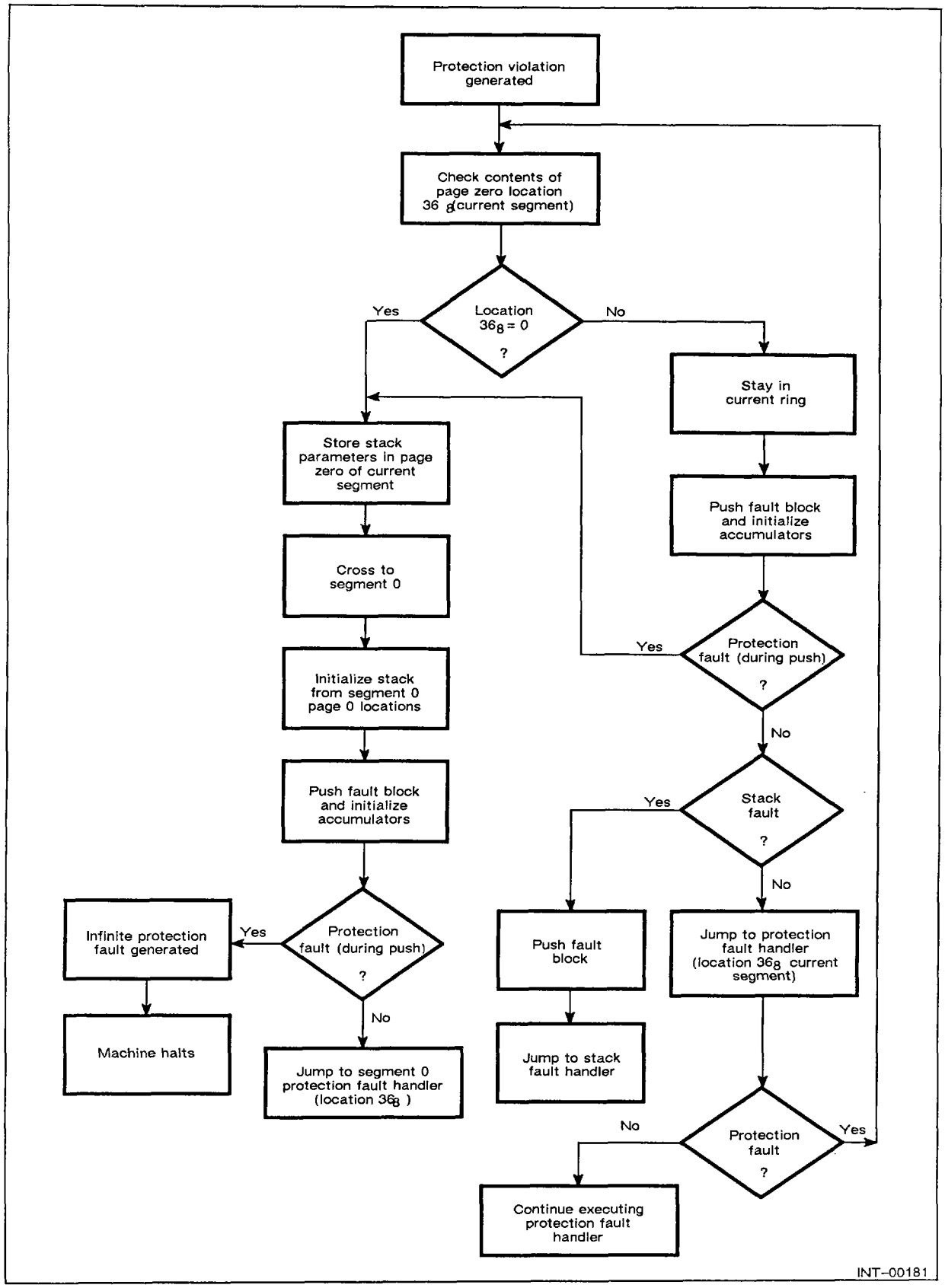


Figure 5-8 *Protection violation sequence*

Table 5–10 *Protection fault codes*

Code (octal)	Address in AC2	Meaning	Explanation
0	Y	Read violation	Bit 2 of the specified PTE contains a zero
1	Y	Write violation	Bit 3 of the specified PTE contains a zero
2	Y	Execute violation	Bit 4 of the specified PTE contains a zero
3	Y	Validity violation (SBR or PTE)	Bit 0 of the specified SBR or PTE contains a zero
4	Y	Inward address reference	Attempted data access to a location in an inner segment
5	Y	Defer (indirect) violation	More than 15 levels 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 zero
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.

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.

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.

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 overflow flag (OVR) to one.

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

If the OVK mask equals zero when the processor sets the OVR flag 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 the OVK fault mask equals one, the processor initiates a fixed-point overflow fault at the end of the current fixed-point instruction. The processor sets AC1, pushes a wide return block, and jumps to the fault handler through the 16-bit indirect pointer in reserved memory. Table 5-11 shows the fixed-point fault return block.

Table 5-11 *Fixed-point fault return block*

Double Word 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	PC (Bit 0 contains CARRY; bits 1-31 contain address of the instruction following the fault-causing instruction)

The PSR word in the return block contains OVR set to zero; OVK set to one; and IRES unchanged. The return address is the address of the instruction the processor executes after servicing the fault.

After the push, AC0 contains the address of the instruction that caused the fault. The processor sets the processor status register to zero and jumps to the fault handler through the 16-bit indirect pointer in reserved memory.

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 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 the TE fault mask 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 the TE fault mask 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. The processor clears the TE bit in the FPSR, and then jumps to the fault handler through the 16-bit pointer in reserved page location 45_8.

The processor services narrow and wide floating-point faults using the same pointer. If the first word (instruction) of the fault handler contains bit 0 set to one and bits 12 through 15 set to 1001_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 5-12 describes the wide return block and Table 5-13 describes the narrow return block.

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

Double Word 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	PC (Bit 0 contains CARRY; bits 1-31 contain address of the instruction following the fault-causing instruction)

Table 5-13 *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 (Bits 17-31 contain address of instruction following the fault-causing instruction)

* *Bits 0-15 of a word correspond to bits 16-31 of a register*

The return address in the return block 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.

After pushing the return block, the processor

1. Sets the TE bit in the FPSR to zero.
2. Sets the PSR to zero (for wide floating-point fault).
3. Transfers program control to the floating-point fault handler.

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.

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 a 16-bit instruction (such as ECLIPSE **EDIT** or **STIX**).

The length and width of the return block depends on the 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.

After pushing the return block, the processor sets the processor status register to zero and places the fault code in AC1 bits 16–31. AC0 contains the value of the program counter for the instruction that caused the fault.

Program control jumps to the fault handler through the 16-bit indirect pointer in reserved memory. Both the wide and narrow faults use the same fault pointer and handler.

Table 5–14 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 the instructions and conditions that caused the fault.

Table 5–14 *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 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 5-15 through 5-17 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 5-15 *Wide return block for decimal data (type 1) fault*

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

Table 5-16 *Wide return block for ASCII data (type 2) fault*

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

Table 5-17 *Wide return block for ASCII data (type 3) fault*

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

Narrow Fault Return Blocks

Tables 5-18 through 5-20 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.

Table 5-18 *Narrow return block for decimal data (type 1) fault*

Word Number in Block Pushed	Contents
1	ACO (Bits 16-31 unchanged)
2	AC1 (Bits 16-31 contain original descriptor)
3	AC2 (Bits 16-31 contain original source indicator -- destination indicator for WSTI or STIX)
4	AC3 (Bits 16-31 undefined)
5	PC (Bit 16 contains CARRY; bits 17-31 contain address of the decimal instruction causing the fault)

Table 5-19 *Narrow return block for ASCII data (type 2) fault*

Word Number in Block Pushed	Contents
1-4	Reserved (for ECLIPSE compatibility)
5	AC0 (Bits 16-31 contain current value of P -- byte pointer to subopcode that caused the fault)
6	AC1 (Bits 16-31 contain original descriptor)
7	AC2 (Bits 16-31 undefined)
8	AC3 (Bits 16-31 undefined)
9	PC (Bit 16 contains CARRY; bits 17-31 contain address of the decimal instruction causing the fault)

Table 5-20 *Narrow return block for ASCII data (type 3) fault*

Word Number in Block Pushed	Contents
1	AC0 (Bits 16-31 unchanged)
2	AC1 (Bits 16-31 contain original descriptor)
3	AC2 (Bits 16-31 undefined)
4	AC3 (Bits 16-31 undefined)
5	PC (Bit 16 contains CARRY; bits 17-31 contain address of the decimal instruction causing the fault)

Stack Faults

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

Narrow Stack Faults

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 16-bit programs (such as the ECLIPSE).

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. For a wide stack underflow, sets the wide stack pointer equal to the wide stack limit.

2. Pushes a wide return block onto the wide stack. (See Table 5-21.)

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

3. Sets the OVK, OVR, and IRES flags (PSR flags) to zero.

4. Sets bit 0 of the wide stack pointer to zero.

5. Sets bit 0 of the wide stack limit to one.

6. Updates the wide stack pointer and reserved memory locations 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 5-22.)

9. Jumps to the wide stack fault handler through the 16-bit indirect pointer in page zero of the current segment.

Table 5-21 *Wide stack fault return block*

Double Word 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	PC (Bit 0 contains CARRY; bits 1-31 contain 31-bit address of the instruction causing the fault if a type 1 fault, else, the instruction address of the next executing instruction)

Table 5-22 *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 -- WMSP, WSSVR, WSSVS, WSAVR, WSAVS (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

If you determine that you must write a wide stack fault handler, the handler must

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.

Narrow Stack Fault Operations

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 narrowstack 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 onto reserved memory page zero ($0–377_8$) without interference from the narrow stack fault handler.

When a narrow stack fault occurs, the processor

1. For narrow stack underflow, sets the narrow stack pointer equal to the narrow stack base.
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 5–23.)

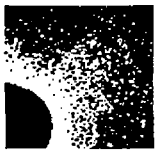
   The return address word in the narrow return block points to the next instruction the processor executes after servicing the fault.
4. Jumps to the narrow stack fault handler through the 16-bit indirect pointer in page zero of the current segment.

Table 5–23 *Narrow stack fault return block*

Word Number 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 16 contains CARRY; bits 17-31 contain the instruction address causing the fault)

If you determine that you must write a narrow stack fault handler, the handler must

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.



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 (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*. Adding a data element to a queue is called *enqueuing*. Removing a data element is called *dequeuing*. 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 *enqueued* anywhere in the queue, not just at the tail. Conversely, data elements can be *dequeued* 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: *Dequeing a data element sets both forward and backward links to -1. If the same data element is dequeued twice, the forward and backward links in the queue descriptor will also contain a -1 which indicates an empty queue.*

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

A data element contains user information as well as the forward and backward links. This user information can precede or follow the forward and backward links. (See Tables 6-1 and 6-2.) The user determines the structure and the meaning of the information.

Table 6-1 *Data element with user data following links*

Position in Data Element	Contents
First double word	Forward link
Second double word	Backward link
Next *n* double words	User information

Table 6-2 *Data element with user data preceding links*

Position in Data Element	Contents
First *n* double words	User information
(*n* + 1)th double word	Forward link
(*n* + 2)th double word	Backward link

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.

Queue Descriptor

Each queue uses a *queue descriptor* that indicates the current head and tail of the queue. A queue descriptor is two 32-bit words. The first double word contains the address of the data element that is currently at the head of the queue: the second contains the address of the data element that is currently at the tail of the queue. (See Figure 6-1.)

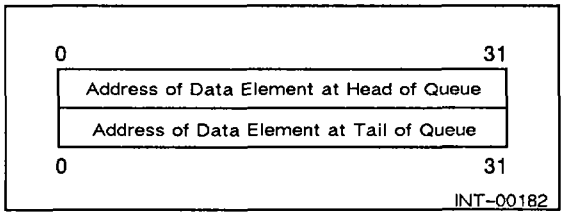


Figure 6-1 *Format of queue descriptor*

Setting Up and Modifying a Queue

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

Examples

The examples below demonstrate how you can form queues, how enqueuing and dequeuing works, and how the processor updates the various links and descriptors.

Queue Descriptor of an Empty Queue

Figure 6-2 shows the queue descriptor for an empty queue.

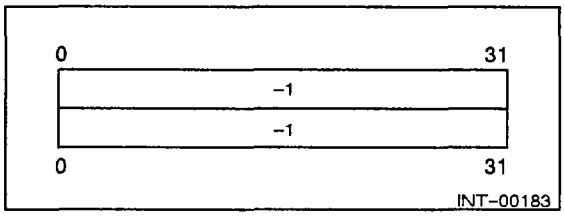


Figure 6-2 *Queue descriptor for an empty queue*

Enqueuing a Data Element into an Empty Queue

Figure 6-3 illustrates how the processor enqueues a data element (at location A) into an empty queue. After enqueuing, 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.

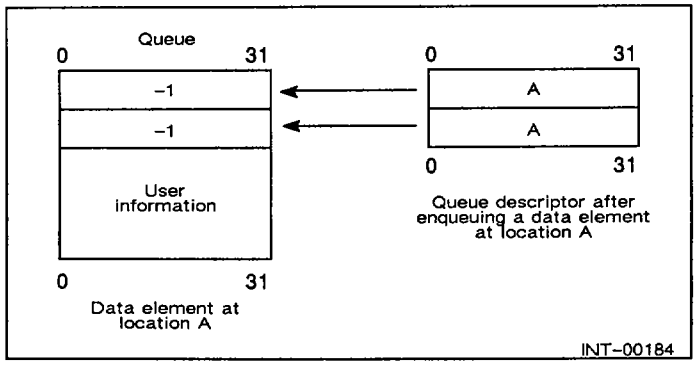


Figure 6-3 *Data element enqueued into an empty queue*

Enqueuing a Data Element at the Head of a Queue

Figure 6-4 illustrates how the processor enqueues a data element (at location B) at the head of the queue before data element A. After enqueuing, 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.

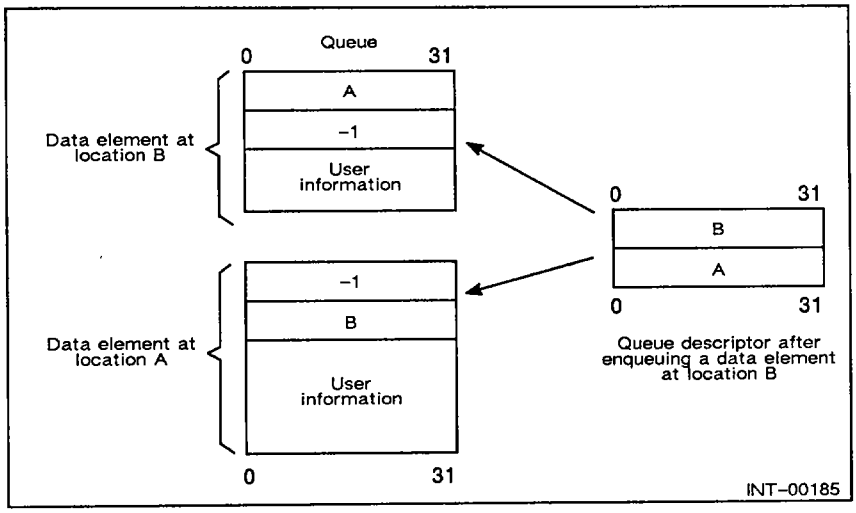


Figure 6-4 *Data element enqueued at head of queue*

Enqueuing a Data Element at the Tail of a Queue

Figure 6–5 illustrates how the processor enqueues 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.

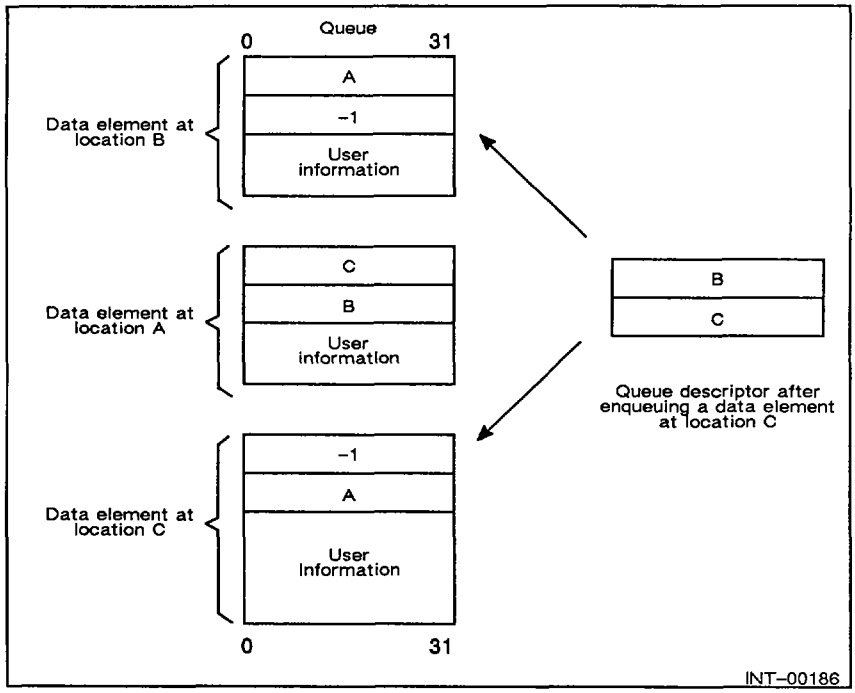


Figure 6–5 *Data element enqueued at tail of queue*

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                   ;Dequeue 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
                                                ;dequeued.
                        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.
```

Dequeueing a Data Element

Figure 6–6 illustrates how the processor dequeues data element B. After the dequeue, 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.

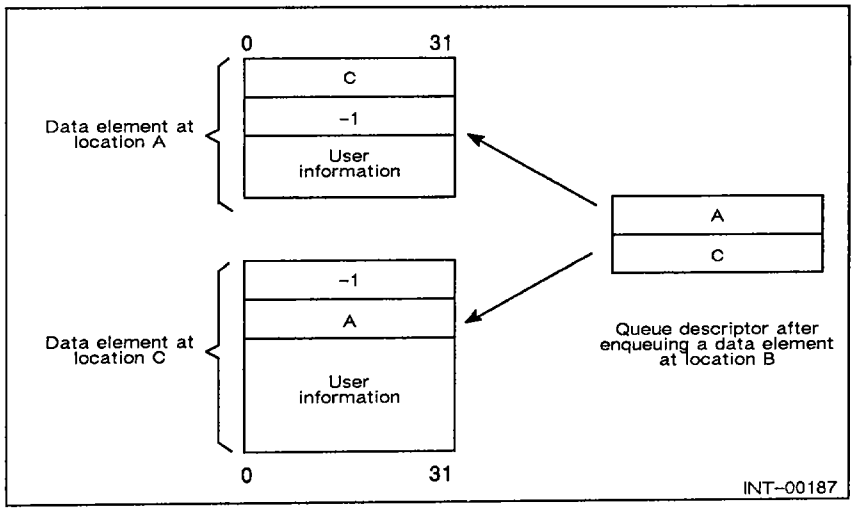


Figure 6–6 *Data element dequeued*

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

```
;This subroutine deques 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 dequeueing
                                                ;element 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 6–3 lists the instructions for manipulating queues. **ENQH** and **ENQT** instructions enqueue data elements onto queues, and the **DEQUE** instruction dequeues 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 data base without software locking.*

Table 6–3 *Queue instructions*

Instruction	Operation
ENQH	Enqueue towards the head; add a data element to queue
ENQT	Enqueue towards the tail; add a data element to queue
DEQUE	Dequeue a queue data element; delete a data element
NBStc	Narrow search queue backward; 16-bit test condition
NFStc	Narrow search queue forward; 16-bit test condition
WBStc	Wide search queue backward; 32-bit test condition
WFStc	Wide search queue forward; 32-bit test condition
WMESS	Wide mask, skip and store if equal

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.

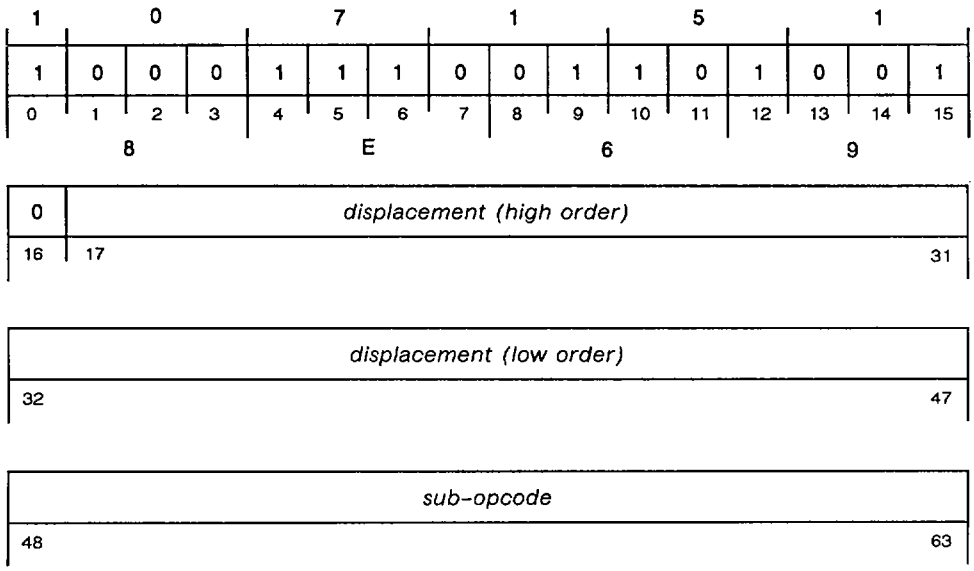
The Graphics Instruction Set, as presented in this manual, is Data General's GIS II for ECLIPSE MV/Family systems. Information on GIS I may be found in the *Data General 4000-Class Integrated Systems, Functional Characteristics* manual (DGC No. 014-001066).

Graphics Instruction Set

GIS 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 identify a GIS instruction; they always contain 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 run-time routine that emulates the GIS instruction functions.

Bits 48–63 are the *sub-opcode*, that identifies 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, references 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).
- 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 "GIS Data Structures" section describes the various properties of the form in detail.

Forms and Bitmaps

Every form is associated with one or two bitmaps. 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 F1's. The operating system must then handle this conflict by copying the obscured part of F1 to the 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.

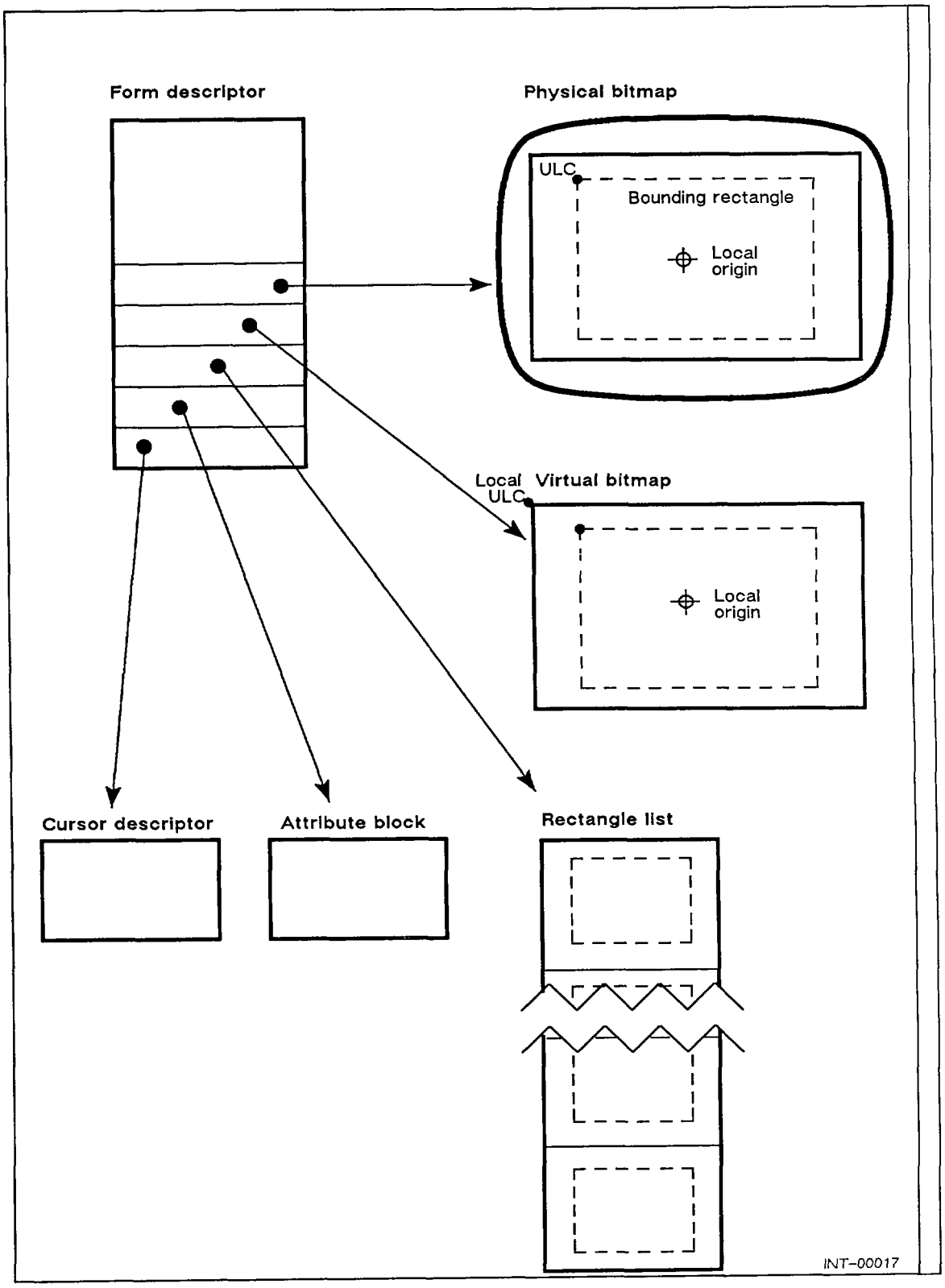


Figure 7-1 *Form data structures*

- 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:

```
Double-word 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.

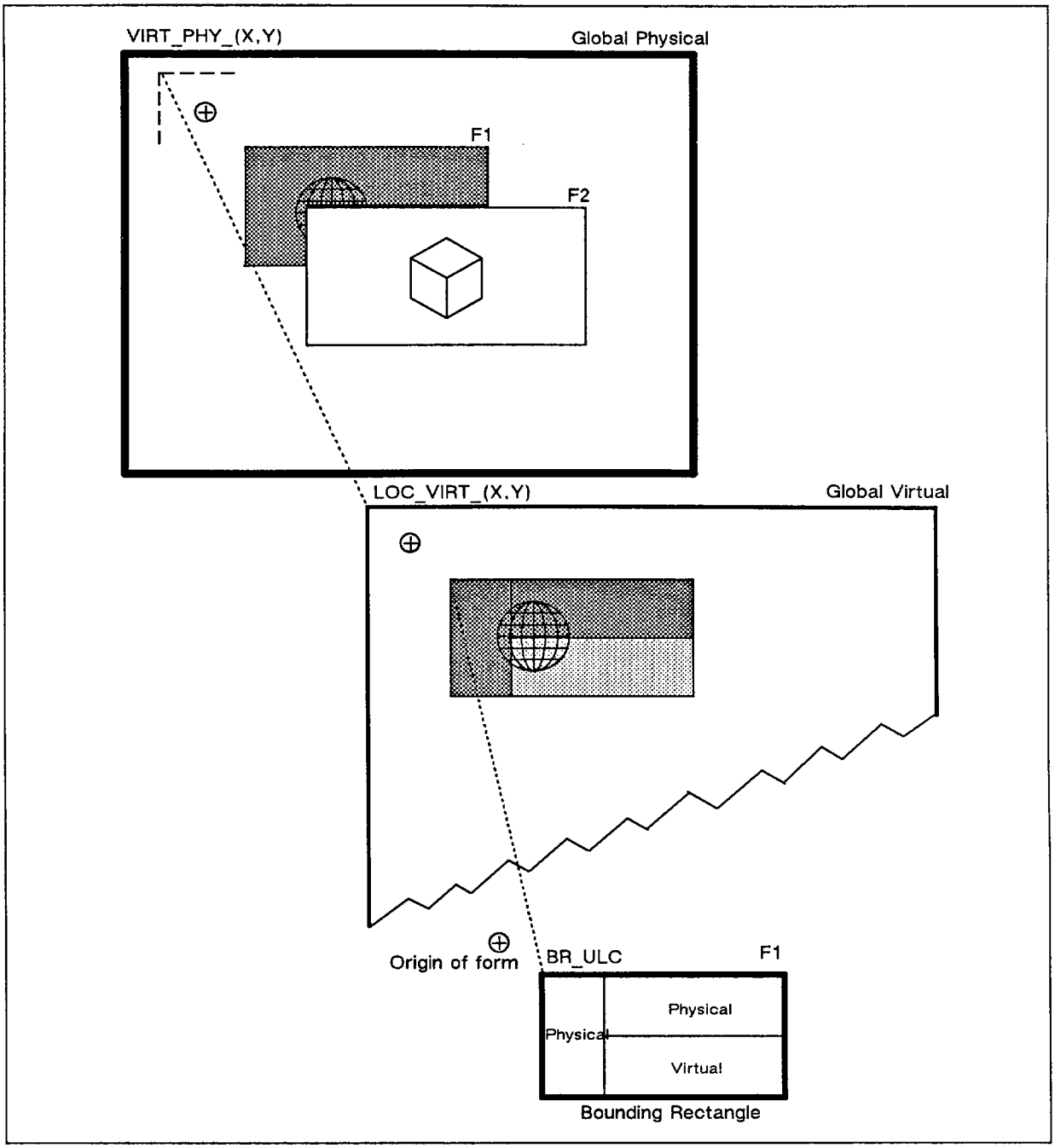


Figure 7–4 *Coordinate conversions*

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

Double Word #	Mnemonic	Integer Type -- Contents
13	FORM_MASK	Unsigned -- Form mask bits. 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 performing a GIS instruction, the FORM_MASK is 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	Unsigned -- Physical address of the start of the form's rectangle list.
15	CURSOR_DESC	Unsigned -- Physical address of a cursor descriptor.
16	DEV_TYPE	Unsigned -- Device type of the physical bitmap. 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 double word 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	Unsigned -- Microcode ID for video board (base address).
18	P_X_PITCH	Unsigned -- Number of bits per pixel in the physical bitmap; also known as "X pitch." This number is a power of 2 in the range 1-32.
19	P_Y_PITCH	Unsigned -- Number of pixels per line for the physical bitmap, also known as "Y pitch." This number must be a power of 2.
20	P_LOG2_XPITCH	Unsigned -- Base 2 logarithm of the X pitch of the physical bitmap.
21	P_LOG2_YPITCH	Unsigned -- Base 2 logarithm of the Y pitch of the physical bitmap.
22	V_BMAP_ADDR	Unsigned -- Logical address of the start of the virtual bitmap memory (this address must be double-word aligned).
23	V_X_PITCH	Unsigned --Number of bits per pixel in the virtual bitmap, also known as "X pitch." This number must be a power of 2 in the range 1-32.
24	V_Y_PITCH	Unsigned -- Number of pixels per line for the virtual bitmap, also known as "Y pitch." This number must be a power of 2.
25	V_LOG2_XPITCH	Unsigned -- Base 2 logarithm of the X pitch for the virtual bitmap.
26	V_LOG2_YPITCH	Unsigned -- Base 2 logarithm of the Y pitch for the virtual bitmap.

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, 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 double word 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. 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 zero means "do nothing with this bit"; a one 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 #	WGRDATTR/ WGWRATTR Index #	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.
4	2	LINE_CTRL	Line control word -- 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. 1 Suppress the initial point. 3 0 Draw the final point. 1 Suppress the final point. 4-31 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.
8	6	CHAR_CTRL	Character control word -- 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.
9	7	CHAR_F_COLOR	Character foreground color -- Pixel value of the foreground color used when drawing characters.
10	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 XOR 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 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.

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.

For solid lines, use a line style of –1 (all bits equal 1). Some other useful values are given in Figure 7–5.

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

INT-00010

Figure 7-5 *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 in the computer industry 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 double word of the descriptor remains the same in both cases. This is the FLAGS double word, which contains bits that determine the format of the descriptor that follows. The format of the cursor descriptors is shown in Figure 7-6, 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.

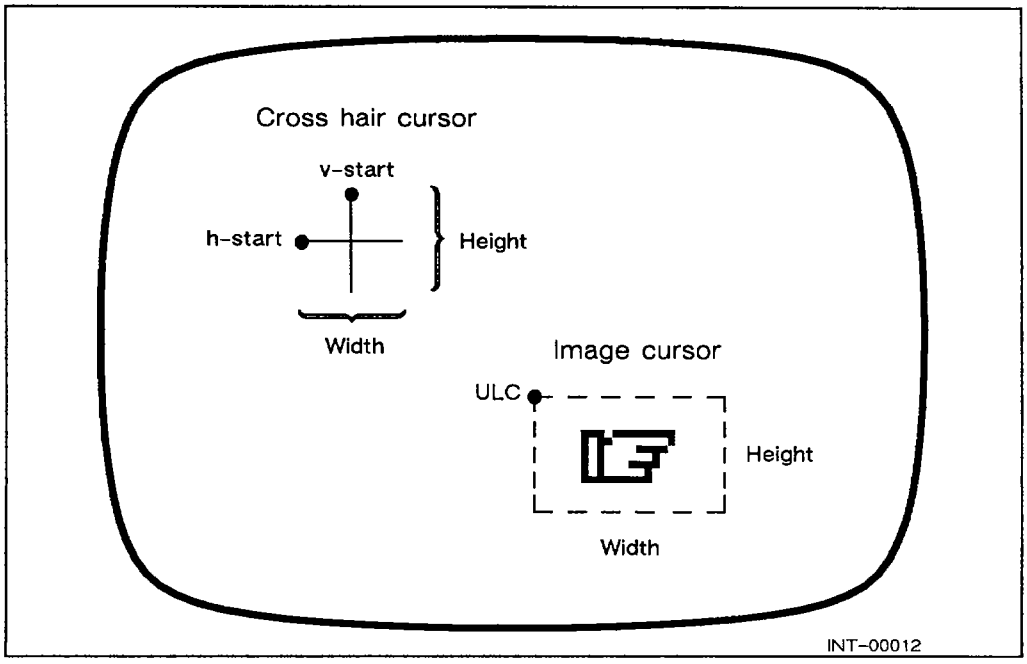


Figure 7-6 *Types of cursors*

Table 7-6 *Cross-hair cursor descriptor*

Double Word #	Mnemonic	Integer Type -- Contents
1	FLAGS	Unsigned -- Flag bits. The individual bits and their interpretations are **Bit # Meaning** 0 1, cursor is visible; 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	Signed -- X coordinate of left endpoint of horizontal line with respect to physical bitmap origin.
3	H_START_Y	Signed -- Y coordinate of left endpoint of horizontal line with respect to physical bitmap origin.
4	H_END_X	Signed -- X coordinate of right endpoint of horizontal line with respect to physical bitmap origin.
5	H_END_Y	Signed -- Y coordinate of right endpoint of horizontal line with respect to physical bitmap origin.
6	V_START_X	Signed -- X coordinate of top endpoint of vertical line with respect to physical bitmap origin.
7	V_START_Y	Signed -- Y coordinate of top endpoint of vertical line with respect to physical bitmap origin.
8	V_END_X	Signed -- X coordinate of bottom endpoint of vertical line with respect to physical bitmap origin.
9	V_END_Y	Signed -- Y coordinate of bottom endpoint of vertical line with respect to physical bitmap origin.

Table 7-7 *Image cursor descriptor*

Double Word #	Mnemonic	Integer Type -- Contents
1	FLAGS	Unsigned -- Flag bits. The individual bits and their interpretations are **Bit # Meaning** 0 1, cursor is visible; 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	Signed -- X coordinate of ULC of cursor rectangle with respect to physical bitmap origin.
3	ULC_Y	Signed -- Y coordinate of ULC of cursor rectangle with respect to physical bitmap origin.
4	EXTENT_X	Unsigned -- Width of cursor rectangle in pixels.
5	EXTENT_Y	Unsigned -- Height of cursor rectangle in pixels.

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 is generated (refer to the "Fault Handling" section). 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 double words 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 non-privileged 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.

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

  a. Context block value causing fault field contains 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. (Refer to the chapters, "Memory and System Management" and "Program Flow Management".)

  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 and 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. Context block value causing fault field contains nothing (undefined).

  b. Operating system traps the process. (Refer to the chapters, "Memory and System Management" and "Program Flow Management".)

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 overflow bit (OVR) in the Processor Status Register 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 7–7 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.

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 7–8, line FG is valid if each endpoint is less than or equal to 2^{29} points to the perpendicular of the 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 "Fixed-Point Overflow Fault" section of the "Program Flow Management" chapter.

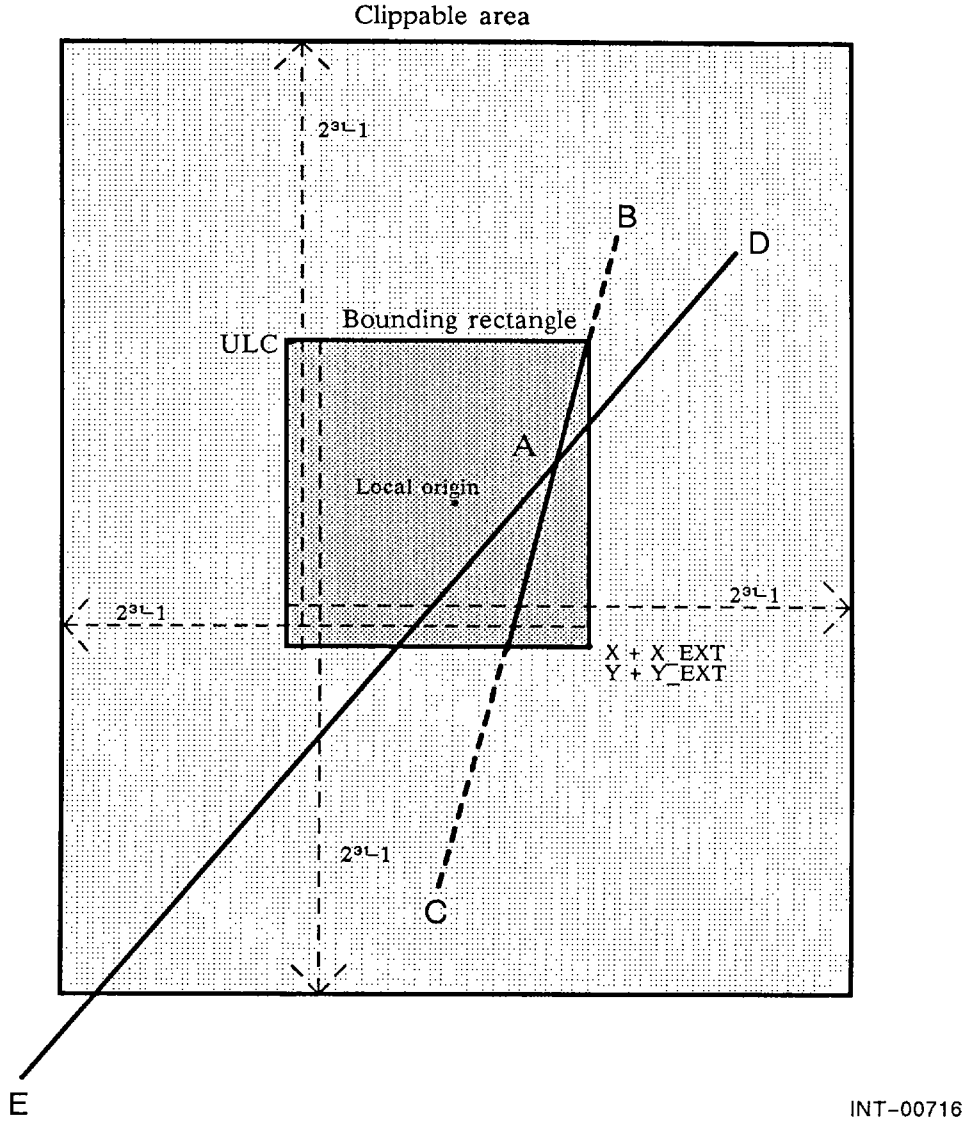


Figure 7–7 *Overdraw condition parameters*

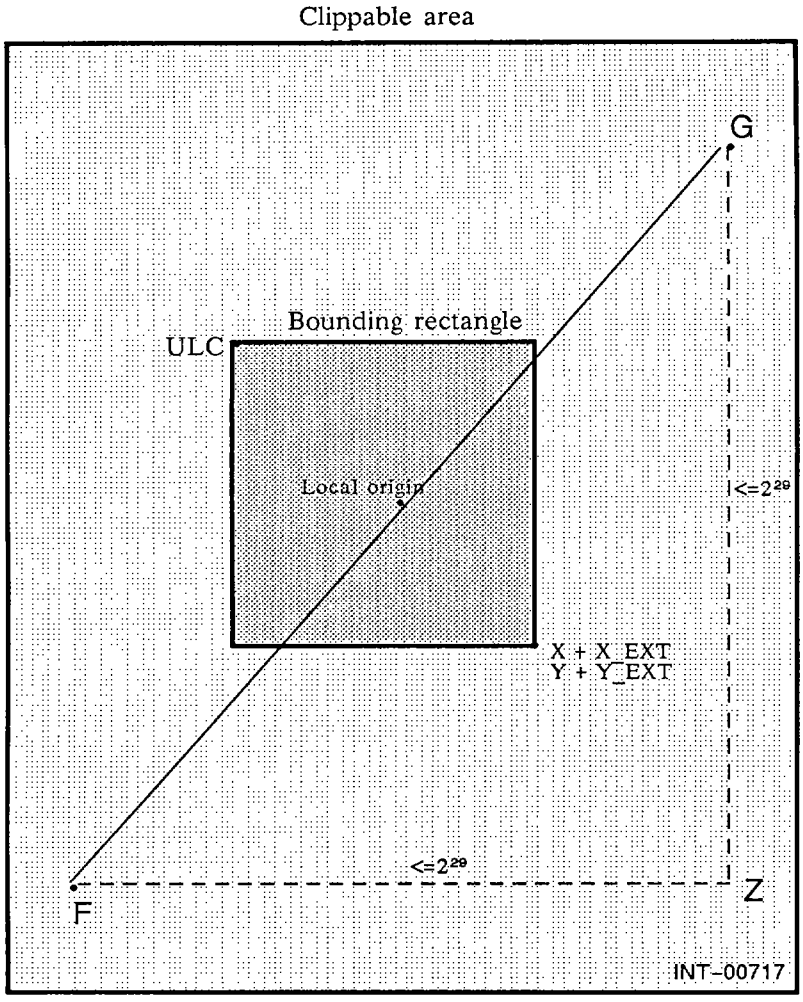


Figure 7-8 *Overdraw condition parameters for endpoints*

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 blocks of words between a medium-speed device and memory. With a burst multiplexor channel I/O facility, the processor transfers blocks of words 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 I/O facility. Finally, the high-speed disks transfer data with the burst multiplexor channel I/O facility.

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 handler or a device driver, which is invoked with a system call.

Device Access

The processor accesses a device through a programmed I/O facility, a data channel I/O facility, or a burst multiplexor channel I/O facility with the address translation disabled or enabled. For the processor to access a device with the address translation disabled, the processor ignores bits 2 and 3 of the segment base register. For the processor to access a device with the address translation enabled, bits 2 and 3 must first be set to enable the I/O instruction execution.

- Bit 2 is the LEF or I/O mode.

  Bit 2 specifies how the processor interprets the LEF and I/O instruction opcodes. For instance, in a segment where the processor executes LEF (Load Effective Address) instructions, bit 2 must be set to one -- selecting the *LEF mode*. Thus, the processor interprets and executes the I/O and LEF instructions as LEF instructions.

  Conversely, in a segment where the processor executes I/O instructions, bit 2 must be set to zero -- selecting the *I/O mode*. Thus, 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.

  Bit 3 enables or disables the execution of an I/O instruction. For example, in a segment where the processor executes I/O instructions, bit 3 must be set to one. If bit 3 equals zero, 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 servicing a protection fault.

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 multi-channel environment, the system uses the default I/O channel with 16-bit I/O instructions. The **PRTSEL** instruction can be used to change the default I/O channel. In a single-channel environment, the **PRTSEL** instruction is a no-op.

  NOTE: *On power-up or after a system reset, channel 0 is the default channel. An I/O reset does not change the default.*

- Direction of the transfer (read or write).

- 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.

The data channel or burst multiplexor channel 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 addresss (**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.

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.

Table 8–1 lists the I/O instructions that affect a device map (a data channel map or burst multiplexor channel map).

Table 8–1 *I/O instructions for data channel/BMC maps*

Instruction	Operation
CIO, CIOI	Issues a read or write command to a register of a device map
IORST *	Sets the status register bits 0, 2–15 to 0 and turns off data channel and BMC mapping
WLMP	Loads a series of double words into a device map
LPHY	Translates a logical address and loads the physical address in an accumulator, for use in the unmapped mode

* *ECLIPSE compatible instruction*

General I/O Instructions

Devices are controlled with I/O instructions. A general set of I/O instructions provide device-independent operations. A special set of I/O instructions communicate with the I/O controller, load a device map, or service a vector interrupt.

The general I/O instructions receive data, send data, and initialize or test a device flag. Table 8–2 lists the general I/O instructions.

Table 8-2 *General I/O instructions*

Instruction		Operation
DIA[*f*]	*	Data In A (from A buffer of device)
DIB[*f*]	*	Data In B (from B buffer of device)
DIC[*f*]	*	Data in C (from C buffer of device)
DOA[*f*]	*	Data out A (to A buffer of device)
DOB[*f*]	*	Data out B (to B buffer of device)
DOC[*f*]	*	Data out C (to C buffer of device)
IORST	*	I/O reset
NIO[*f*]	*	No I/O transfer (initialize a BUSY/DONE flag)
PIO		Issue a programmed I/O command to a device
SKP*t*	*	I/O skip (test a BUSY/DONE flag and skip on condition)

* *ECLIPSE compatible instructions*
[f] and t define optional device flag handling.

Figure 8-1 illustrates the format for a general I/O instruction.

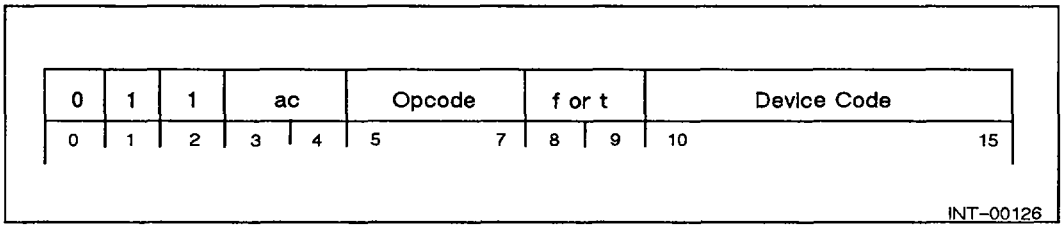


Figure 8-1 *General I/O instruction format*

In Figure 8-1,

011	The 011 binary code indicates an I/O instruction.
ac	The *ac* field indicates a fixed-point accumulator (0–3).
	The accumulator (bits 16–31) contains the data to send or receive from a device.
Opcode	The opcode field identifies the I/O instruction operation.
	Table 8-2 lists the I/O instructions.
f or *t*	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.) For an external and an internal device (except for the CPU), the flags are BUSY and DONE. For the internal CPU device, the flags are interrupt on (ION) and power fail.
Device Code	The device code field identifies a unique device controller to send or to receive the data.

With a 6-bit device code, the processor can communicate with up to 64_{10} device controllers. The assembler translates a standard three-, four-, or five-letter device mnemonic to a device code.

Refer to the "Standard I/O Device Codes" appendix for a complete list of standard device mnemonics and for the corresponding device code.

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 power fail = 1
DZ	1 1	Test for DONE = 0	Test for power fail = 0

The BUSY and DONE flags indicate the device state to a device driver. When both flags equal zero, the device is idle. To start a device, a programmer issues 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 is ready to start another operation, the device sets the BUSY flag to zero and the DONE flag to one.

The interrupt on (ION) flag controls the device interrupt system. When the ION flag equals zero, the processor ignores interrupt requests. When the ION flag equals one, the processor services interrupt requests.

The powerfail flag indicates the processor state to the CPU device driver. When the powerfail flag equals zero, the processor detects the proper power voltage ranges. When the powerfail flag equals one, the processor detects a powerfail condition.

Interrupts

The processor and the 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. The program transmits the request through I/O system calls, which initialize the device and transfer data by invoking the interrupt system.

The operating system maintains control of the interrupt system by manipulating an interrupt on flag, an interrupt mask, and the 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 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 the interrupt mask enable processor recognition of the interrupt request, the processor services the interrupt. To service an interrupt, the processor first determines the action to take on the currently executing instruction, then redefines the interrupt mask, and finally services the interrupt request. The "Interrupt Servicing" section explains the processor actions to transfer program control to the interrupt handler and then to the interrupt service routine.

Interrupt On Flag

With the interrupt on (ION) flag equal to one, the processor responds to an interrupt request. When the ION flag equals zero, the processor cannot respond to an interrupt request.

The **INTDS** and **INTEN** CPU device instructions control the state of the ION flag. Information on CPU device instructions is presented in the latter part of this chapter.

Instruction Interruption

Most instructions are noninterruptible because they require very little CPU execution time. For instructions that require more time, such as the Wide Block Move (**WBLM**) instruction, the processor (if required) interrupts the executing instruction, sets the processor status register (PSR) bit 2 to a 1, and continues servicing the interrupt.

After servicing the interrupt, the processor either restarts or resumes the interrupted instruction.

Table 8-4a shows the processor settings of bits 2 and 3 of the PSR when an interrupt occurs during execution of a resumable instruction. Figure 8-1a summarizes the sequence of events upon the interruption of a resumable instruction.

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

Table 8-4a *State of PSR bits 2 and 3*

Instruction Type	PSR Bit 2 (IRES)	PSR Bit 3 (IXCT)
ECLIPSE or ECLIPSE MV/20000 series-specific	1	1 if executed via **PBX**; otherwise 0

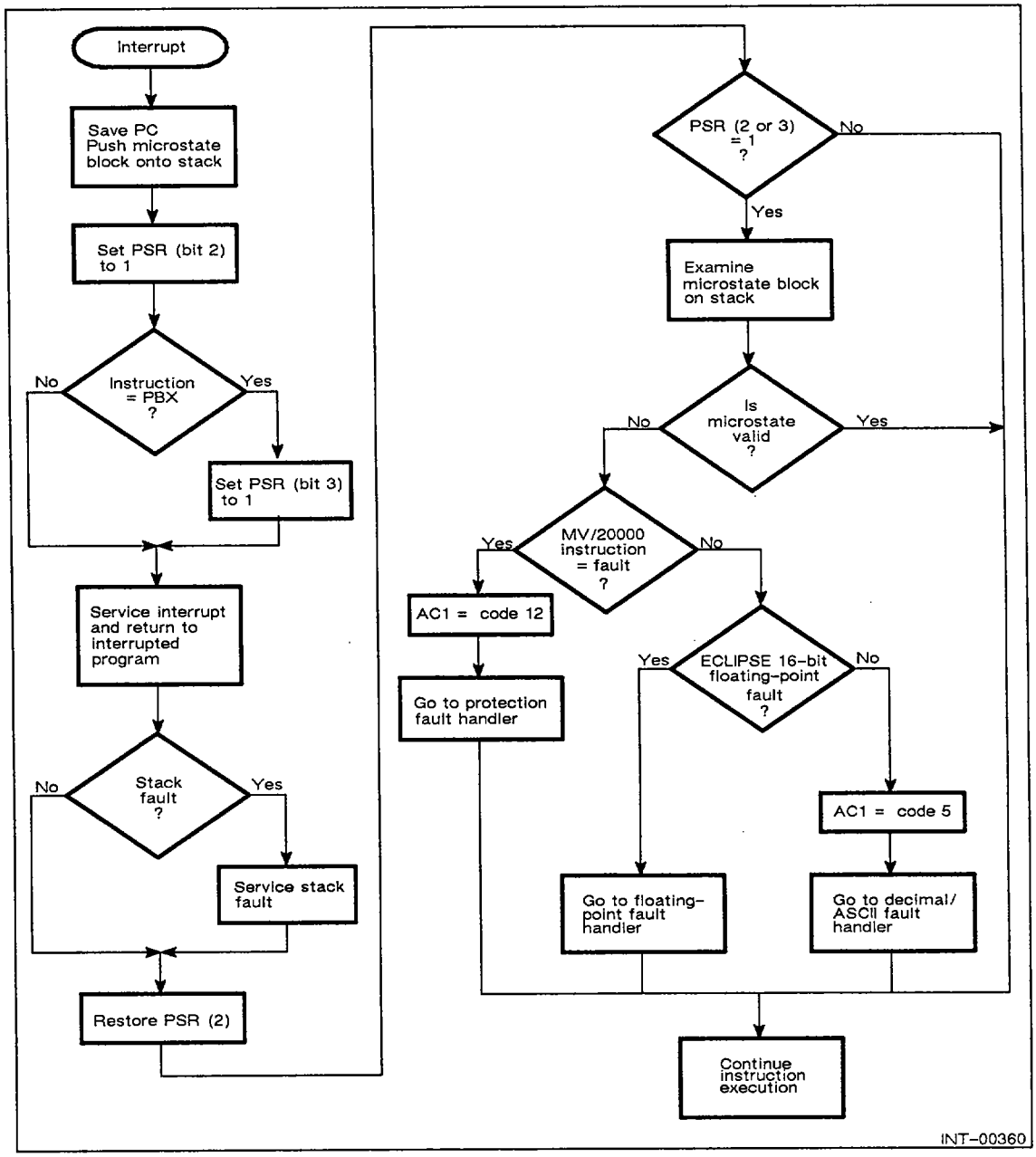


Figure 8-1a *Resumable instruction interrupt sequence*

Interrupt Mask

A device is associated with one of the 16 bits in the interrupt mask. When the bit equals one, the mask blocks an interrupt request to the processor. When the bit equals zero, the processor services an interrupt request from the device. Since the processor can address more than 16 device controllers, it can use a bit in the interrupt mask for two or more devices.

To change the state of a bit in the interrupt mask, use the mask out instruction (MSKO), a CPU device instruction.

Interrupt Servicing

To service an interrupt request (Figure 8-2), the processor

1. Sets the ION flag to zero.
2. Determines if the address translation facilities are enabled.

   Refer to the chapter "Memory and System Management" for information on enabling and disabling the address translation facilities.
3. Fetches the pointer to the interrupt handler.
4. Changes the current segment of execution to segment 0, if the current segment equals segment 1 through 7 and if the address translation facilities are enabled.
5. Resolves the effective address of the interrupt handler.
6. Examines the first word of the interrupt handler, which may be one of the following types:
   - Type 1, a 32-bit processor instruction.

     A 32-bit processor instruction contains bit 0 equal to one and bits 12-15 equal to 1001_2 .
   - Type 2, a 16-bit processor instruction.

     Instructions other than **XVCT** or Type 1 are identified as 16-bit processor instructions.
   - Type 3, a vector interrupt (**XVCT**) instruction.
7. Stores the return address in
   - Logical locations two and three of segment 0 for a type 1 instruction.
   - Location 0 of segment 0 for a type 2 instruction.
   - The vector stack (as part of the return block) for the **XVCT** instruction, during the vector interrupt processing.
8. Jumps indirectly
   - 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.
   - To the ECLIPSE interrupt handler and executes the type 2 instruction as the first instruction of the ECLIPSE interrupt handler.
   - To the vectored interrupt handler and executes the **XVCT** instruction.

     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.

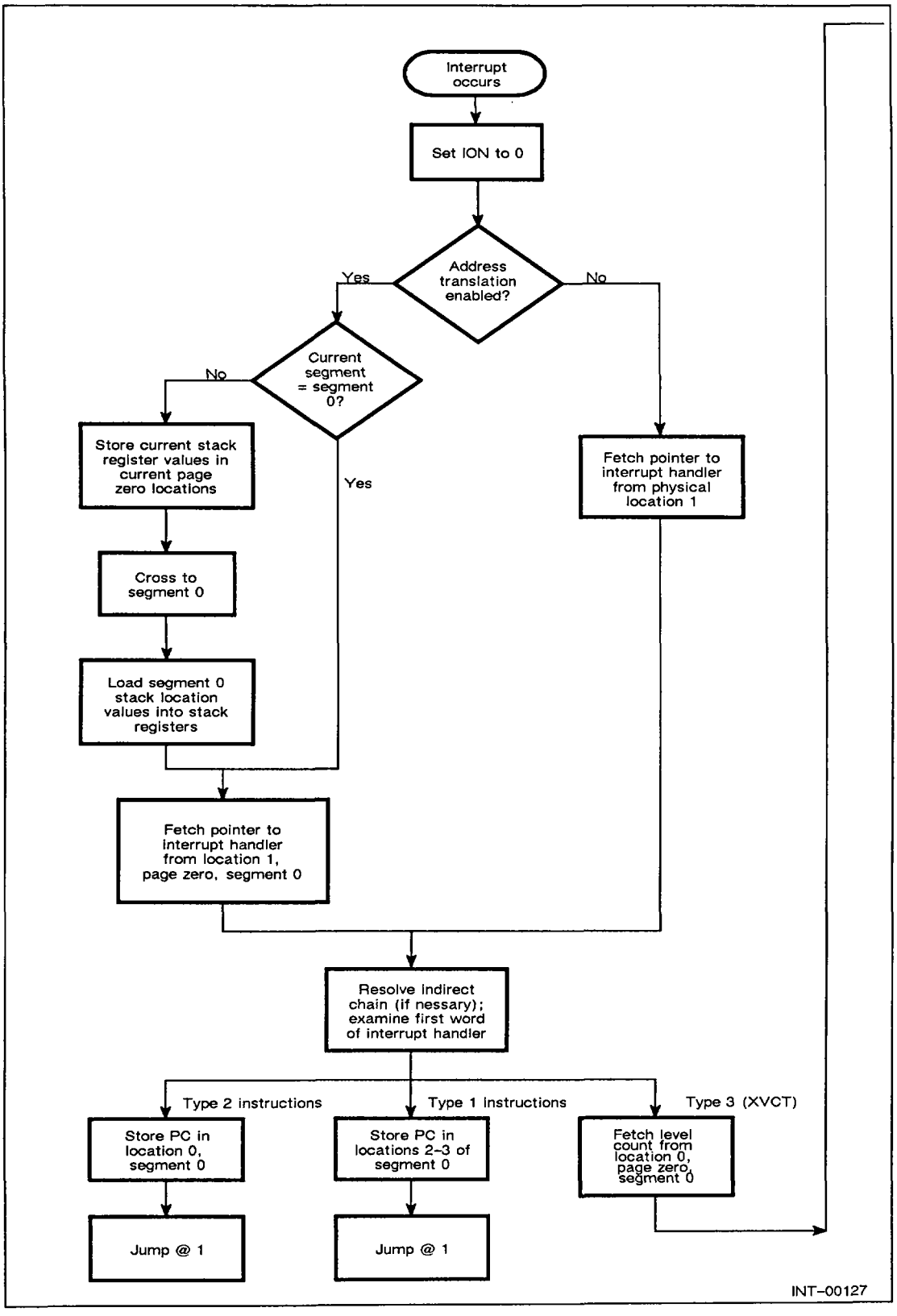


Figure 8-2 *Interrupt sequence (continues)*

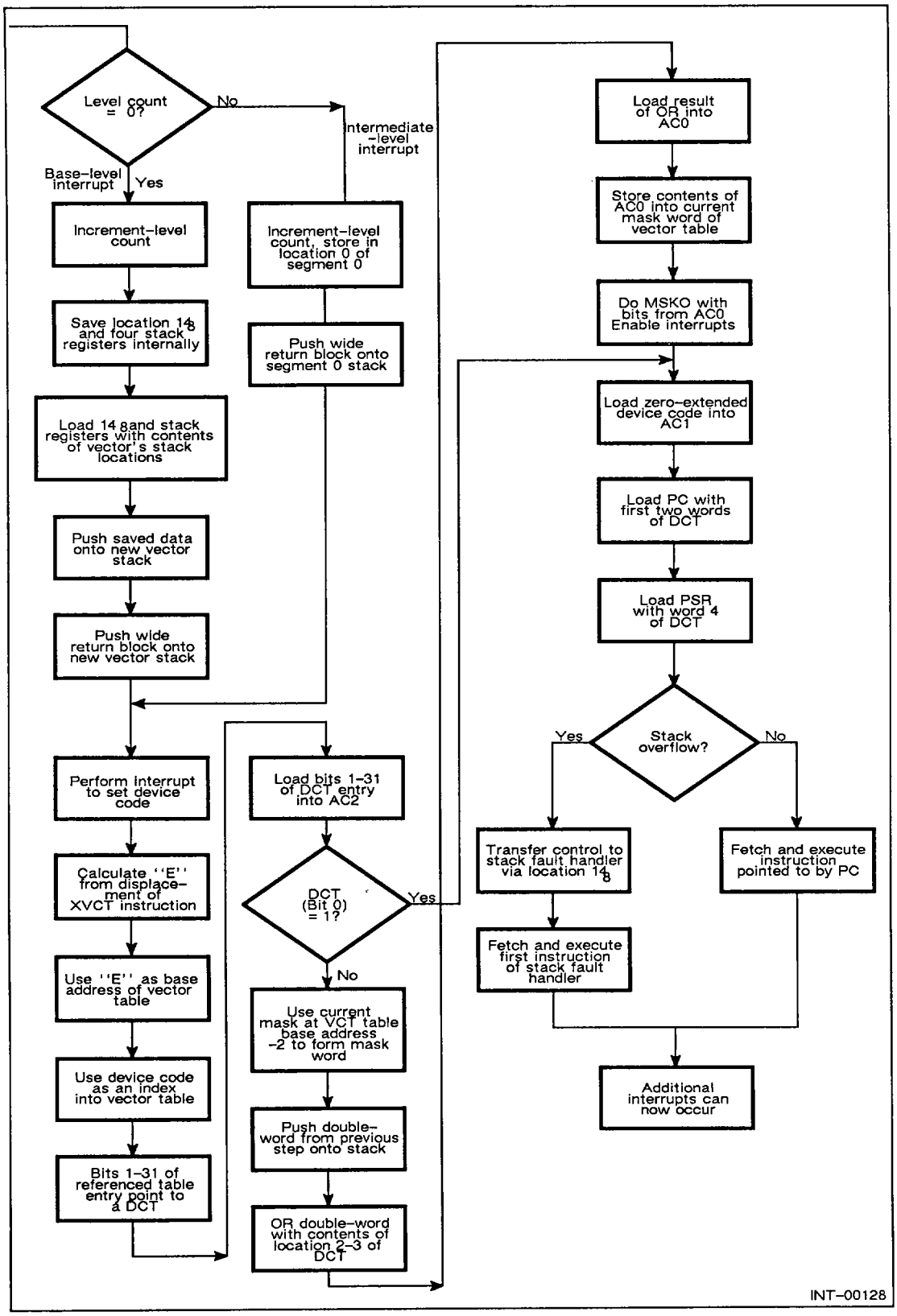


Figure 8-2 *Interrupt sequence (concluded)*

Vectored Interrupt Processing

The processor tests the contents of the interrupt-level word in location zero of zero segment in reserved memory. If the contents equal zero, then the processor begins base-level interrupt processing. If the contents equal nonzero, then the processor begins intermediate-level interrupt processing. The processor, in either case, increments the contents by one.

NOTE: *Software, as part of the interrupt return, must decrement the interrupt-level word by one.*

Base-Level Interrupt Processing

The processor begins base-level interrupt processing at step 1 when the current segment equals segment 1 through 7. The processor begins base-level interrupt processing at step 5 when the current segment equals segment 0.

To service a base-level vector interrupt, the processor

1. 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.)

2. Crosses to segment 0.

3. Saves the wide stack parameters from the reserved memory locations of segment 0 in an internal processor state.

4. Continues execution with step 6.

5. Saves the pointer to the wide stack fault handler and the four wide stack registers in an internal processor state.

6. Uses the three vector stack parameters in reserved memory (locations 4, 6, and 7) to initialize the four wide stack registers and wide stack fault pointer for the vector stack.

   - Vector stack pointer parameter

     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

     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 128K bytes of segment 0.*

   - Vector stack fault address parameter

     Loading the vector stack information enables vector stack underflow and overflow detection.

7. Pushes the previously saved wide stack parameters from the internal processor state onto the vector stack.

8. Pushes a wide return block onto the vector stack.

9. Continues execution as described in the section "Final Interrupt Processing."

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 8–3), the processor

1. Calculates the effective address from the displacement of the **XVCT** instruction. The indirection chain, if any, is narrow.

   The effective address identifies word zero of the vector table. The table contains 64 double-word entries for each I/O channel, one entry for each device on an I/O channel. Figure 8–4 illustrates the vector table.

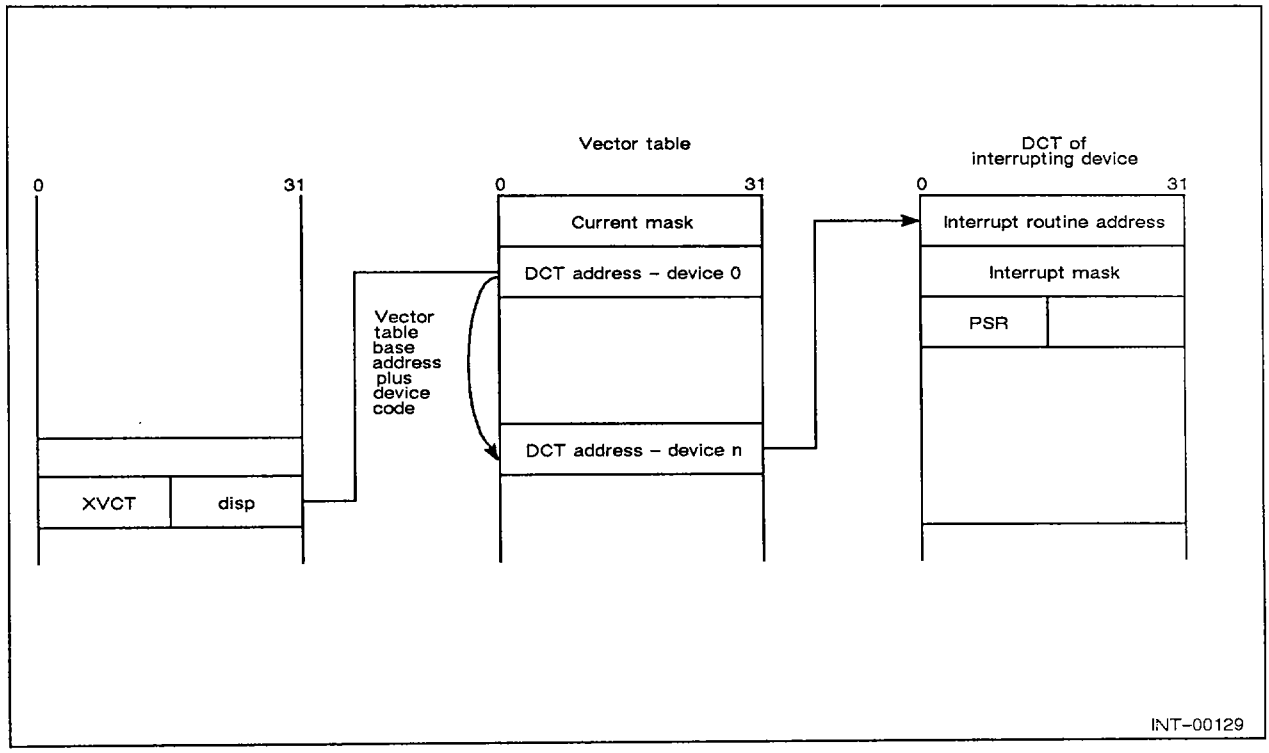


Figure 8–3 *Sequence of actions to conclude interrupt service*

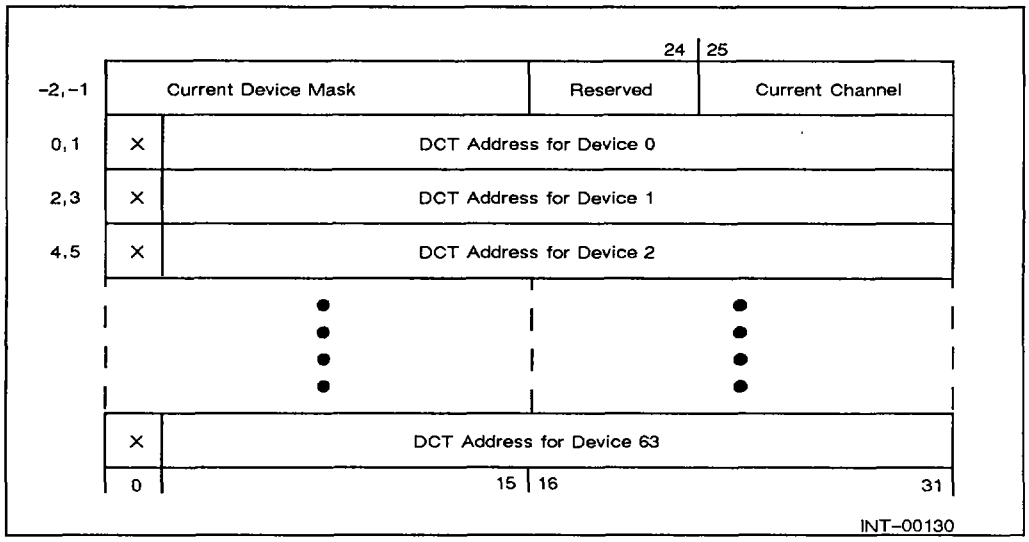


Figure 8-4 *Vector table*

2. Uses the interrupting device number as a double-word 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 words of the device control table are defined by the **XVCT** instruction. These words must be set up as shown in Figure 8-5. 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.

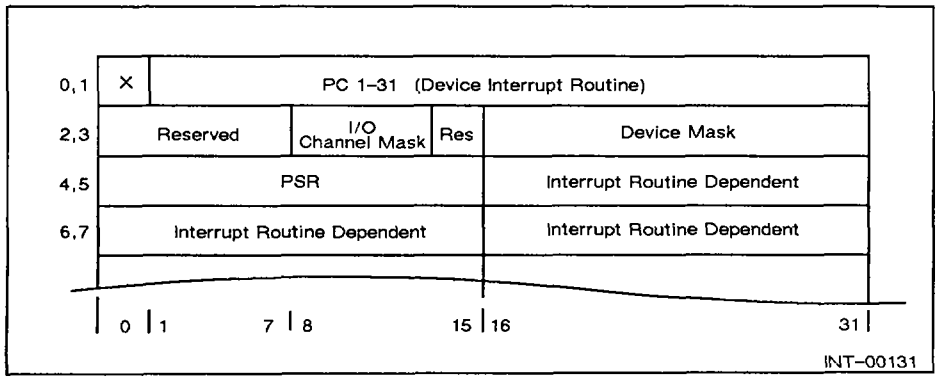


Figure 8-5 *Device control table (DCT)*

3. Loads AC2 with the base address of the device control table.

NOTE: *On machines capable of multiple CPUs, if bit 0 equals 1, proceed to step 8. This allows a program to optionally not use the built-in masking functionality.*

4. Constructs a double word and pushes it onto the vector stack.

   Bits 0-7 of the double word contain all zeros. Bits 8-31 contain the contents of the current mask from the vector table. The I/O channel masks are organized one bit per channel; for example, bit 14 equals I/O channel 7.

5. Loads AC0 with the inclusive OR of the pushed double word and the second double word (words two and three) of the device control table.

6. Stores AC0 into the current mask.

7. Performs the function of a mask out (**MSKO**) instruction with AC0 and enables interrupts.

   When a mask bit equals one, the processor disables interrupt recognition of devices that use the mask bit.

8. Loads the least significant bits of AC1 with the interrupting I/O channel and device number, zero-extended to 32 bits (0-23).

9. Loads the program counter with the address of the device interrupt routine (words zero and one of the device control table).

10. Initializes the PSR from the contents of the PSR word in the device control table.

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 or vector stack fault handler before honoring further interrupts.

    NOTE: *On machines capable of multiple CPUs, 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.

Integral Devices

The following sections of this chapter decribe instructions for manipulation of the following devices:

- central processing unit
- programmable interval timer
- real-time clock
- primary asynchronous line input/output
- system control processor (or program)
- data channel and burst multiplexor channel
- universal power supply controller

The "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

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 enable 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 enable flag equals 0. The CPU interrupt enable flag is controlled by the device flag commands as follows:

f=Not Used	ION flag unchanged.
f=**S**	Sets the interrupt enable flag to 1.
f=**C**	Sets the interrupt enable flag 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. The instruction that initializes the devices and sets the priority mask bits to 0 uses the following standard form:

DIC[*f*] *ac*,**CPU**

The same instruction can take the following special form:

IORST

The special **IORST** assembler statement is equivalent to the following standard assembler statement:

DICC 0,CPU

Both statements set all the 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 (**IORST**).

NOTE: *The assembler detects a fatal format error when a device flag is appended to a special CPU instruction.*

CPU Instructions

Table 8-5 lists the I/O instructions—both standard and special—that affect the CPU. These are all privileged instructions and must be executed in segment 0.

Table 8-5 *I/O instructions for the CPU*

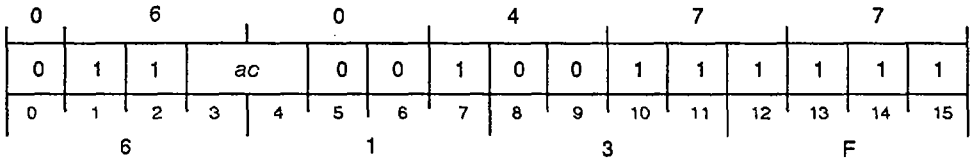
Assembler Statement		Function
Special Form	Standard Form	
READS *ac*	DIA*[f]* *ac*,CPU	Reads console switches (places the contents of the console data switches into an accumulator).
PRTSEL	NIO CPU	On a single I/O channel machine, performs no operation. On a multiple-I/O channel machine, sets the default 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 the ION flag or 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 multiple I/O channels, these instructions execute as multiple-I/O channel instructions.*

Read Switches

READS

READS *ac*



Function: Console switches -> *ac*
Unchanged -> ION flag

Parameters: None

NOTE: **READS** *ac* = **DIA** *ac*,**CPU**

The Read Switches instruction places the contents of the console switches into the specified accumulator.

Arguments

ac(16–31) After execution, contains console switch settings: 1 = ON, 0 = OFF.

Registers, Flags, and Stacks

AC0–AC3 Can be individually specified as *ac*; otherwise unused.

ION Flag Unchanged

CARRY Unchanged

Overflow Unaffected

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

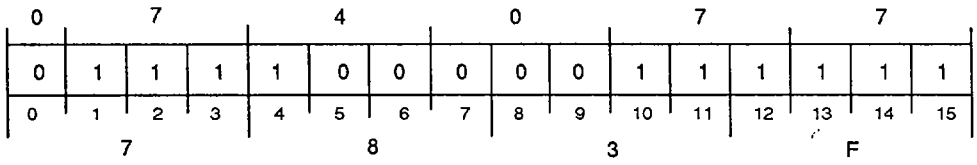
DIA *ac*,**CPU** The assembler recognizes **READS** *ac* to be equivalent to **DIA** *ac*,**CPU**.

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 = -1, then default I/O channel -> AC0

PRTSEL is a no-op instruction on those machines that implement a single I/O channel. On multiple I/O channel machines, 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.

NOTES: *Use this instruction carefully in multiple I/O channel machines.*

In multiple-CPU systems, this instruction changes the default I/O channel on all *CPUs.*

Arguments

None

Registers, Flags, and Stacks

AC0(29-31)	Before execution, contains I/O channel with 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 8-6 shows the effect of using the I/O channel numbers with various I/O instructions.

Table 8-6 *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	Read CPU switch register	Undefined	Undefined
INTA *ac* DIB*[f]* *ac*,CPU	INTA to channel *n*	INTA channel *n*	INTA highest priority channel/device
IORST *ac* DIC*[f]* *ac*,CPU	Reset all I/O	PRTRST channel *n*	PRTRST all channels
MSKO *ac* DOB*[f]* *ac*,CPU	MSKO to channel *n*	MSKO to channel *n*	MSKO all channels
HALT *ac* DOC*[f]* *ac*,*CPU*	HALT to 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	SKP on ION = 1	Undefined	Undefined
SKPBZ CPU	SKP on ION = 0	Undefined	Undefined
SKPDN CPU	SKP if powerfail	Undefined	Undefined
SKPDZ CPU	SKP if no powerfail	Undefined	Undefined

* *n equals a value in the range of 0 to the maximum number of implemented I/O channels*

Exceptions

On a 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):

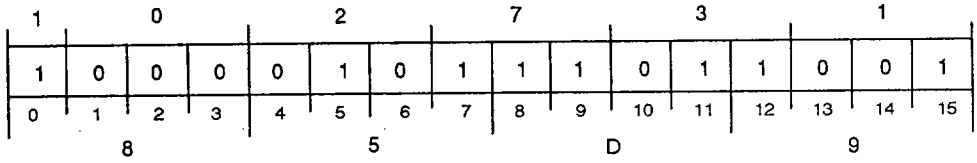
- The ECLIPSE 16-bit compatible I/O instructions use the default I/O channel.
- The **PIO** instruction (with an ECLIPSE 16-bit compatible I/O instruction) uses any implemented I/O channel.

In either case, results are undefined with any channel number other than those implemented.

I/O Channel Reset

PRTRST

PRTRST



Function: I/O channel devices -> clear states
Priority mask -> 0
I/O channel mask bit -> 0
ION flag -> 0

Parameters: AC0 = I/O channel # -> unchanged

PRTRST is a no-op instruction on those machines that implement a single I/O channel. The I/O Channel Reset instruction sends a reset signal to all devices on the I/O channel specified in AC0, instructing those devices to clear their states. In addition, the instruction sets that I/O channel's 16-bit priority mask to 0, clears the I/O channel mask bit (I/O channel mask register), and sets the interrupt on (ION) flag to 0.

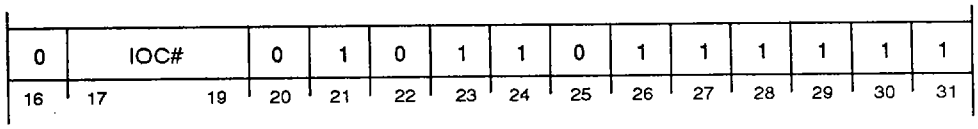
A **PRTRST** issued to an I/O channel resets only that channel; issued to channel 7, it resets all implemented I/O channels. All unimplemented channel numbers produce undefined results.

Arguments

None

Registers, Flags, and Stacks

AC0(16-31) Specifies I/O channel (bits 0-15 are undefined). Format is as follows:



AC1-AC3	Unused
CARRY	Unchanged
ION Flag	After execution, set to 0.
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

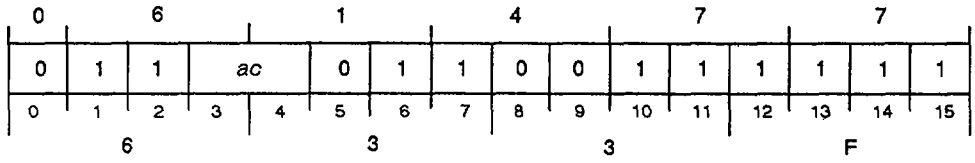
PIO If AC0 contains the specified bit pattern (see AC0 description), the assembler recognizes **PIO** *0,0* to be equivalent to **PRTRST.**

Exceptions

A **CIO** read to the I/O channel mask register will have undefined CIO results.

Interrupt Acknowledge INTA

INTA *ac*



Function: Device code -> *ac*
Unchanged -> ION flag

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 that has the highest priority.

Arguments

ac(26–31) After execution, contains device code; bits 0–25 are set to 0.

Registers, Flags, and Stacks

AC0–AC3 Can be individually specified as *ac*; otherwise unused.

CARRY Unchanged

ION Flag Unchanged

Overflow Unaffected

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

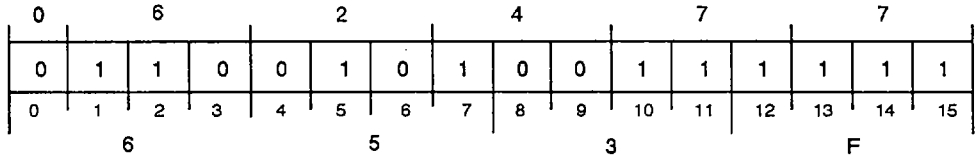
DIB[*f*] *ac*,**CPU** This optional form of Interrupt Acknowledge allows the ION flag to be manipulated.
The assembler recognizes **INTA** *ac* to be equivalent to **DIB** *ac*,**CPU**.

Exceptions

Do not use the **DIBP** *ac*,**CPU** instruction as it is reserved for the **VCT** instruction on the ECLIPSE C/350 computer.

I/O Reset IORST

IORST



Function: Clear all I/O devices
0 -> Priority mask
0 -> PSR
0 -> FPSR(0-9)
0 -> ION flag
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 sets the 16-bit priority mask to 0; disables logical address translation; sets the PSR to 0; sets bits 0 through 9 of FPSR to 0; and sets the ION flag to 0.

NOTES: *In multi-I/O channel environments,* **IORST** *sets the I/O channel mask register flag for channel 0 to 0, and sets bits 0, 3, 4, 7, 8, 9, and 14 of the I/O channel definition register (6000_8) to 0.*

In multiple-CPU systems, **IORST** *also resets IMODE to 0, 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 Flag	Set to 0
Overflow	Unaffected
PC	PC + 1
PSR	Set to 0
FPSR(0-9)	Set to 0
Stack	Unchanged

Related Instructions

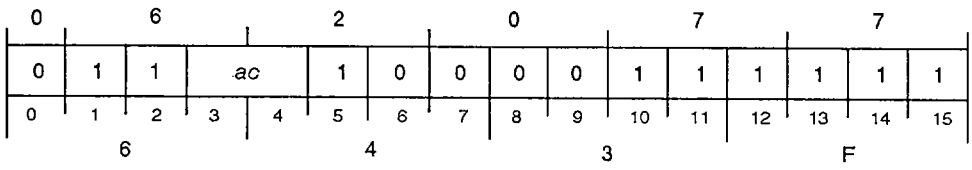
DIC[*f*] *ac*,**CPU** This optional form of the I/O Reset instruction allows manipulation of the ION flag. When using the optional form of the I/O Reset instruction, an accumulator must be coded to avoid assembly errors. During execution, the processor ignores the accumulator field, and the contents of the accumulator remain unchanged.
The assembler recognizes **IORST** to be equivalent to **DICC 0, CPU**.

Exceptions

None

Mask Out MSKO

MSKO *ac*



Function: *ac* -> Priority mask
Unchanged -> ION flag

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 at devices that use that bit as a mask. (Refer to the "Standard I/O Device Code" appendix.)

After execution, remains unchanged.

Registers, Flags, and Stacks

AC0-AC3 Can be individually specified as *ac*; otherwise unused.

CARRY Unchanged

ION flag Unchanged

Overflow Unaffected

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

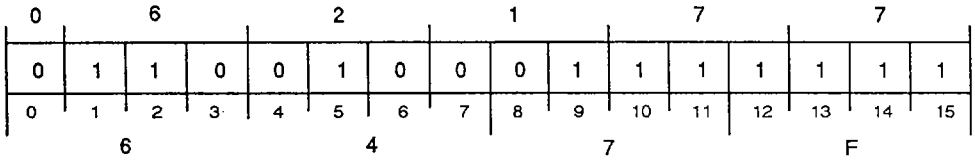
DOB[*f*] *ac*,**CPU** This optional form of Mask Out allows manipulation of the ION flag. The assembler recognizes **MSKO** *ac* to be equivalent to **DOB** *ac*,**CPU**.

Exceptions

None

Halt HALT

HALT



Function: Stops the processor
Unchanged -> ION flag

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 Flag	Unchanged
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

DOC[*f*] *ac*,**CPU** This optional form of Halt allows manipulation of the ION flag. 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.

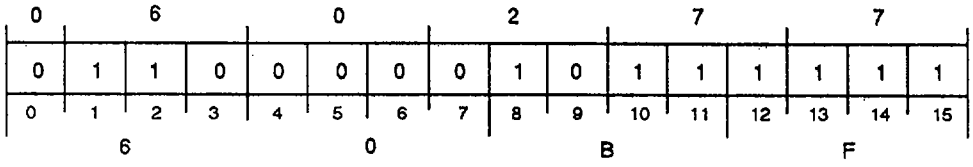
The assembler recognizes **HALT** to be equivalent to **DOC 0,CPU**.

Exceptions

None

Interrupt Disable INTDS

INTDS



Function: 0 -> ION flag

Parameters: None

NOTE: INTDS = NIOC CPU

The Interrupt Disable instruction sets the ION flag to 0.

Arguments

None

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
ION Flag	Set to 0
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

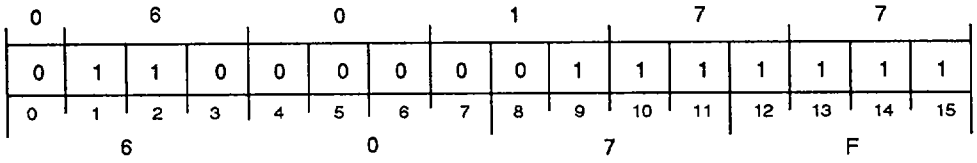
NIOC CPU	The assembler recognizes the mnemonic **NIOC CPU** to be equal to **INTDS**.

Exceptions

None

Interrupt Enable INTEN

INTEN



Function: 1 -> ION flag

Parameters: None

NOTE: INTEN = NIOS CPU

The Interrupt Enable instruction sets the ION flag to 1.

Arguments

None

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
ION Flag	Set to 1
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

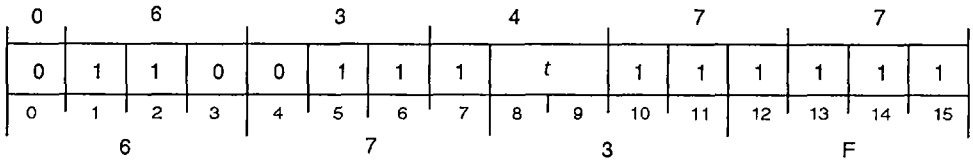
NIOS CPU The assembler recognizes the mnemonic **NIOS CPU** to be equal to **INTEN.**

Exceptions

If the instruction changes the state of the ION flag, 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
(false test return)
(true test return)



Function: If *t* = true then skip
BUSY and DONE flags -> unchanged

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
BN	0 0	Test for ION = 1
BZ	0 1	Test for ION = 0
DN	1 0	Test for power fail = 1
DZ	1 1	Test for power fail = 0

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
ION Flag	Unchanged
Overflow	Unaffected
PC	PC + 1 (false test) PC + 2 (true test)
PSR	Unchanged
Stack	Unchanged

Related Instructions

None

Exceptions

None

Programmable Interval Timer

Device Code
43_8

Assembler Mnemonic
PIT

Priority Mask Bit
6 or 11 (See 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. It 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. Since the counter continues incrementing after reaching 0, it is necessary to reload the PIT counter with the initial value to restart the count. 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* = 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

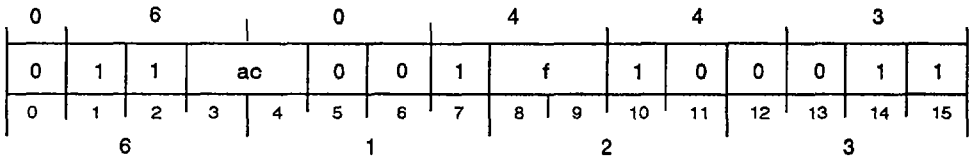
Table 8-7 lists the I/O instructions that affect the PIT device.

Table 8-7 *Instructions affecting the PIT*

Assembler Statement	Function
DIA*[f]* *ac*,PIT	Reads the counter value into the accumulator
DOA*[f]* *ac*,PIT	Loads the counter with a value (PIT initializes the counter with the value each time the counter starts or overflows)
IORST	Stops the counting cycle and sets the BUSY and DONE flags, the interrupt mask bit, the initial count register, 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 (two's complement); bits 0 through 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

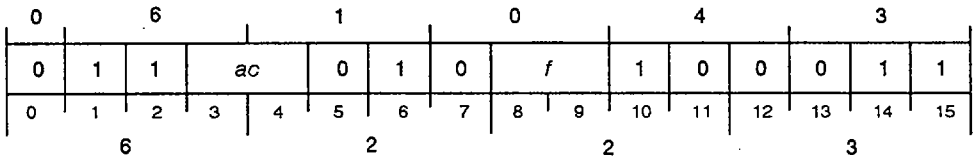
DOA[*f*] *ac*,**PIT** Specify Initial Count

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 ac-line frequency, 10 Hz, 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 **IORST** is issued, the processor sets the clock to the line frequency. The ac-line frequency pulses are available immediately after powerup. However, five seconds must elapse before the clock produces a steady pulse train for the 10 Hz, 100 Hz, or 1000 Hz frequencies.

Device Flag Control

Device flag commands to the RTC enable or disable RTC interrupts.

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

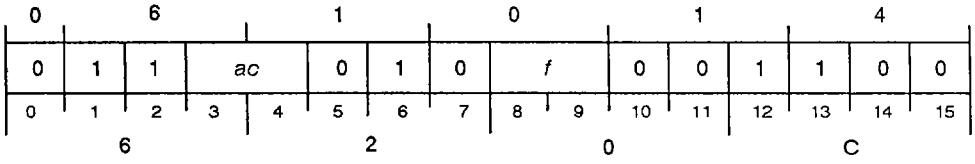
Table 8-8 lists the I/O instructions that affect the RTC device.

Table 8-8 *Instructions affecting the RTC*

Assembler Statement	Function
DOA*[f]* *ac*,RTC	Selects a clock frequency with a 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. Selects clock frequency as follows:

Bits	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 Device Code	OUTPUT Device Code
10_8	11_8
Assembler Mnemonic	**Assembler Mnemonic**
TTI	TTO
Priority Mask Bit	**Priority Mask Bit**
14	15

The primary asynchronous interface (TTI/TTO) is the communication link between the processor and the master terminal. It supports asynchronous communication at selected rates from 110 to 4800 baud (depending on the machine, this range may be greater) in 7-bit codes with program-generated parity or in 8-bit codes with no parity. You can use 1 or 2 stop bits with either format.

Because the asynchronous communications input and output can generate program interrupts independently, each has its own device code and is controlled by its own set of BUSY and DONE flags.

The TTI/TTO 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 receives each character, it places the character in an input character 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 character 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 TTI/TTO determine the flag settings and whether or not the output character is transmitted.

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

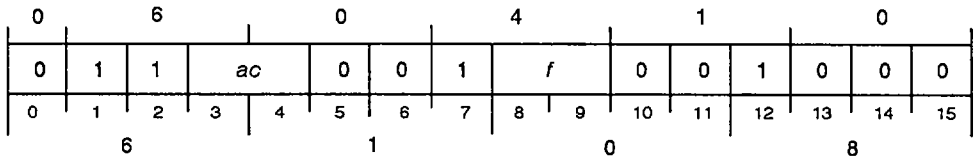
Table 8-9 lists the I/O instructions that affect the TTI/TTO device.

Table 8-9 *I/O instructions for TTI and TTO*

Assembler Statement	Function
DIA*[f] ac*,**TTI**	Reads a character from the device into an accumulator
DOA*[f] ac*,**TTO**	Sends a character from an accumulator to the 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

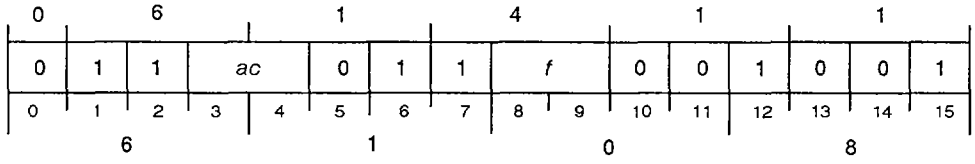
DOA*[f] ac*,**TTO** Load Character Buffer

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 functions.

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

Diagnostic Remote Processor

Device Code
45_8

Assembler Mnemonic
SCP

Priority Mask Bit
15

The diagnostic remote processor (DRP), described in Chapter 1, is a dedicated processor that serves as the front end to the ECLIPSE MV/20000 series systems. The DRP provides the line interface to the system console and connects to the major components of the computer through the primary IOC.

The DRP subsystem performs the following four types of tasks:

- Monitors system powerup—The DRP monitors each step in the testing and initialization sequence. In dual CPU configurations, the DRP synchronizes the CPUs. If an error occurs, the DRP responds by displaying an error code indicating the faulty element. For further information, refer to the manual, *Using the ECLIPSE MV/20000™ Series System Control Program.*
- Controls programmed I/O for the TTI, TTO, ECLIPSE MV/20000 Models 1 and 2 PSC, and SCP—The TTI, TTO, and PSC are described elsewhere in this chapter. The system control program (SCP) information is presented in this section.
- Holds most of the SCP operating system—These functions are described in the manual, *Using the ECLIPSE MV/20000™ Series System Control Program.*
- Controls data flow between the operator's console, remote diagnostic port, and the user port—This and other system functions are performed by the DRP operating system.

The DRP also supports the following features:

- Boot clock—A battery backed-up clock/calendar that comes preset and is used to initialize the time-of-day clock, at operating system boot time, and for error log time stamping.
- System configuration state—A nonvolatile configuration RAM that contains the CPU, IOC, and memory configurations for system configuration and testing.
- Front control panel—The front panel LEDs, hexadecimal displays, and switches are controlled by the DRP.

The SCP is a ROM-based operating system that runs on both the DRP and the main CPUs. During full operation, the SCP allows the system operator to load, examine, and modify main memory or writable control store and step through the instructions of a program.

Device Flag Control

Device flag commands to the SCP determine the settings of the BUSY and DONE flags.

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 8-10 lists the instructions that permit the CPU to communicate with the SCP.

Table 8-10 *SCP instructions*

Mnemonic	Name	Function
DOBS *ac*,**SCP**	Enable/Disable Error Reporting	Enables/disables CPU error reporting and performs indicated command
DIBC *ac*,**SCP**	Return SCP Status	Returns the current status of the SCP
SKP*t* **SCP**	Skip Test	Tests the SCP BUSY/DONE flag and skips next instruction if true
NIOC SCP	Clear SCP BUSY/DONE Flags	Clears the SCP BUSY/DONE flags; leaves the SCP in diagnostic mode
IORST	I/O Reset	Disables CPU error reporting

The **SKP** and **IORST** instructions are described earlier in this chapter. Note that the **DIA, DOA, DIC,** and **DOC** instructions to the SCP are no-ops.

Before issuing a **DOBS SCP** instruction, the process should check the SCP BUSY flag. If the BUSY flag is 0, the SCP is ready to accept the next **DOBS** instruction.

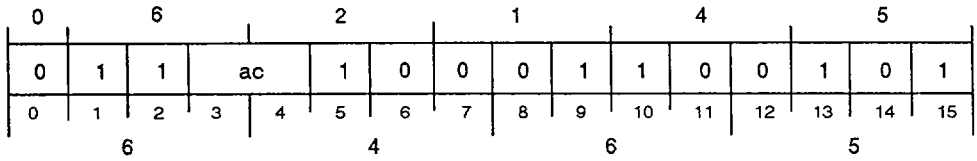
The process should also test the CPU-to-SCP buffer for availability. Protocol requires the process to set word 0 of this buffer to nonzero before using it; the SCP sets word 0 to zero after the SCP has used the information in it.

The S pulse of the **DOBS** *ac*,**SCP** instruction notifies the SCP that the B register is full. The contents of the accumulator specified in the **DOBS** *ac*,**SCP** instruction is interpreted by the SCP as a function request and an SCP log enable/disable request. (See the description of the Enable/Disable Error Reporting instruction.)

The SCP then reads the B register and performs the indicated function. If the indicated function uses the CPU-to-SCP buffer, the SCP sets word 0 of the buffer to zero after it has finished using the information stored in that buffer. The SCP sets the BUSY flag to 0 and the DONE flag to 1 once the B register is available to the process again.

Enable/Disable Error Reporting

DOBS *ac*,**SCP**



Function: SCP error reporting -> enable/disable
Perform command
BUSY = 1; DONE = 0

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	Name	Contents or Function
16	E	The E flag enables the SCP error reporting 1 = enable; 0 = disable
17-23	Command	The SCP performs the function defined by these bits:
24-31	Interface Block	The CPU or SCP uses bits 24 through 31 as either a physical address pointer (in the range of 0 to 377_8) to a multiple word block in page zero or as a type indicator for the Size command.

Command (octal)	Name
000	No-op
001	Reserved
002	Reserved
003	Reserved
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	Reserved
013	Set boot clock time
014	Get boot clock time
015	Set GMT offset
016	Get GMT offset
017	Reserved
020-176	Undefined
177	Enter diagnostic sequence

The Enable command enables CPU error reporting. When the CPU or SCP reports an error, it uses the page zero address specified by the last Set Block command as a pointer to a double-word 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.

If the SCP interrupts the CPU, the SCP disables error reporting until the process 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 used commands 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 it does not change the current ERCC reporting mode.

- Set Block (command 004_8)

The Set Block command specifies to the SCP the address of the SCP-to-CPU interface block. The address points to a two double-word block in page zero. The first double word contains the physical address of the SCP-to-CPU buffer; the second double word contains the physical address of the CPU-to-SCP buffer. Note that both physical addresses are nonindirectable. CPU-to-SCP buffer lengths are specified by the functions that use them. SCP-to-CPU logging requires an SCP-to-CPU buffer of 16 words.

- 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.

Single-bit	1-bit ERCC error detected during memory read
Multi-bit	2-bit (or more) ERCC error detected during memory read
Soft-sniff	1-bit ERCC error detected during memory refresh
Hard-sniff	2-bit (or more) ERCC error detected during memory refresh

This function requires an SCP-to-CPU buffer of five words.

NOTE: *After a reset or power restore, reporting of any ERCC codes is turned off until this function code (005_8) is issued.*

- Set Boot Clock Time (command 013_8)

The Set Boot Clock Time command disables error reporting and sends new boot clock values to the SCP. This command requires a CPU-to-SCP buffer of seven words, and the contents of words one through six must be binary coded decimal (BCD) as follows:

Word #	Contents
0	Nonzero protocol word
1	Seconds (BCD)
2	Minutes (BCD)
3	Hours (BCD)
4	Days (BCD)
5	Months (BCD)
6	Years (BCD)

Upon completion, the SCP sets word zero of the SCP buffer to 0, returns a value to the B register, and sets the DONE flag. A B register value of 2 indicates an acknowledgement; a value of 3 means the SCP cannot perform this function (in this case,

no further action is taken by the SCP). Note that if the clock is changed via the SCP-CLI, the SCP places the value 0 into B register, enters code 206_8 into its error log, and posts an interrupt.

- Get Boot Clock Time (command 014_8)

The Get Boot Clock Time command disables error reporting and requests the boot clock time from the SCP. The SCP writes the requested information into the SCP-to-CPU buffer. Words one to six are BCD, and the contents of the buffer are interpreted as follows:

Word #	Contents
0	1 (subcode indicating this buffer contains Boot Clock data)
1	Seconds (BCD)
2	Minutes (BCD)
3	Hours (BCD)
4	Days (BCD)
5	Months (BCD)
6	Years (BCD)

Upon completion, the SCP returns a value to the B register and sets the DONE flag. A value of 4 indicates that the requested information is now in the appropriate buffer; a value of 3 means the SCP cannot perform this function (in this case, 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 offset values to the SCP. This command requires a CPU-to-SCP buffer of eight words. Words one through six are BCD, and the contents of the buffer are interpreted as follows:

Word #	Contents
0	Nonzero protocol word
1	Seconds (BCD)
2	Minutes (BCD)
3	Hours (BCD)
4	Days (BCD)
5	Months (BCD)
6	Years (BCD)
7	Offset sign (0 = positive; 1 = negative)

Upon completion, the SCP sets word zero of the SCP buffer to 0, returns a value to the B register, and sets the DONE flag. A B register value of 2 indicates an acknowledgement; a value of 3 means the SCP cannot perform this function (in this case, no further action is taken by the SCP). Note that if the GMT offset is changed via the SCP-CLI, the SCP places the value 0 into 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 offset values from the SCP. The SCP writes eight words into the SCP-to-CPU buffer. Words one to six are BCD, and the contents of the buffer are interpreted as follows:

Word #	Contents
0	2 (subcode indicating this buffer contains GMT offset data)
1	Seconds (BCD)
2	Minutes (BCD)
3	Hours (BCD)
4	Days (BCD)
5	Months (BCD)
6	Years (BCD)
7	Offset sign (0 = positive; 1 = negative)

Upon completion, the SCP returns a value to the B register and sets the DONE flag. A value of 4 indicates that the requested information is now in the appropriate buffer; a value of 3 means the SCP cannot perform this function (in this case, 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 page zero address entered with this **DOBS SCP** instruction. Instead, the SCP uses the previous page zero address as a pointer to the SCP-to-CPU interface block. The SCP clears its BUSY flag and remains in diagnostic mode until either a console reset occurs or the process issues another **DOBS SCP** instruction.

When the process issues the second **DOBS SCP** instruction, the SCP first places the contents of bits 16 through 31 of the specified accumulator into word 0 of the SCP-to-CPU buffer. The SCP then reads words 1 through 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 transmits a status 0 to the host, sets the DONE flag, and interrupts the CPU.

An **IORST** instruction also clears diagnostic mode, and the SCP clears diagnostic mode on RESET, START, and BOOT commands.

NOTE: *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

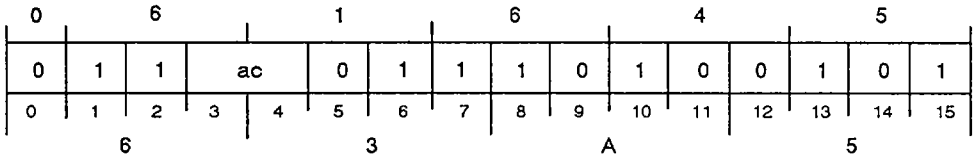
None

Exceptions

IORST	Issuing the **IORST** instruction causes double-bit errors to be checked and corrected, but not reported.

Return SCP Status

DIBC *ac*,**SCP**



Function: Status code -> *ac*
BUSY = 0; DONE = 0

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 via the SCP-to-CPU interface block. 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) 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 listed below.

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 multi-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:
2	Physical page number
3	Double word on module
4	Syndrome bits

Bits	Definition
0-11	Reserved
12	CPU access
13	I/O access
14	Reserved
15	Sniff access

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, 147	Reserved
150	Battery back-up 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
	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, 170	Reserved
	171	SCP RESET command has been executed
	172	Address translation unit enabled
	173	Address translation unit disabled
	174	Illegal PIO command
	175, 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
	207	GMT offset changed
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 acknowledgement indicates to the process that the SCP has completed a requested function.	
000003	SCP requested function in error. The SCP reports an unknown error with this code. The SCP issues this code if a required SCP/HOST interface block has not been defined, an undefined function request is made, or an invalid data is passed to the SCP (through the HOST-to-SCP buffer).	
000004	Data requested in buffer. The SCP has placed the requested information in the SCP-to-CPU buffer. The subcode values returned to the first word of the buffer indicate the contents of the buffer:	
	Subcode	**Definition**
	001	Boot Clock data
	002	GMT offset data
177777	SCP in diagnostic sequence.	

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
BUSY and 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

IORST Issuing the **IORST** instruction causes double-bit errors to be checked and corrected, but not reported.

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.

DCH/BMC Maps

A map controls a DCH or BMC. This map is a series of contiguous map slots, each of which contains a pair of map registers: an even-numbered register and its corresponding odd-numbered register.

ECLIPSE MV/Family computer systems support 16 DCH maps, each of which contains 32 map slots. With each 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.

The device controller performing the data transfer controls the BMC. Program control or CPU interaction is only required when setting up the BMC's map table. The BMC has two address modes and contains its own map.

BMC Address Modes

The BMC operates in either unmapped mode (physical) or mapped mode (logical).

In unmapped mode, the BMC receives 20-bit addresses from the device controllers and passes them directly to memory. As the BMC transfers each data word to or from memory, it increments the destination address, causing successive words to move to or from consecutive memory locations.

If the controller specifies the mapped mode for a data transfer, the high-order 10 bits of the logical address form a logical page number, which the BMC map translates into a 10-bit physical page number. This page number combines with the 10 low-order bits from the logical address to form a 25-bit physical address, which the BMC uses to access memory.

BMC Map

The BMC uses its own map to translate logical page numbers into physical ones. (On those machines that implement it, the **SSPT** instruction defines the memory locations of the BMC map.) The map table contains 1024 map registers, and each odd-numbered register contains a 10-bit physical page number. The BMC uses the logical page number as an index into the map table, and the contents of the selected map register become the 10 high-order bits of the physical address.

Note that when the BMC performs a mapped transfer, it increments the destination address after it moves each data word. If the increment causes the 10 low-order bits to overflow, the BMC selects a new map register 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.

DCH/BMC Registers

ECLIPSE MV/Family systems contains 512 DCH registers and 1024 BMC registers. The device map registers are numbered from 0 through 7777_8, as explained in Table 8-11 and depicted in Figure 8-6.

Table 8-11 *Device map registers 0000-7777_8*

Registers (Octal)	Description
0000-3776	Even-numbered registers are the most significant half of BMC map positions 0-1777
0001-3777	Odd-numbered registers are the least significant half of BMC map positions 0-1777
4000-5776	Even-numbered registers are the most significant half of DCH map positions 0-777
4001-5777	Odd-numbered registers are the least significant half of DCH map positions 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

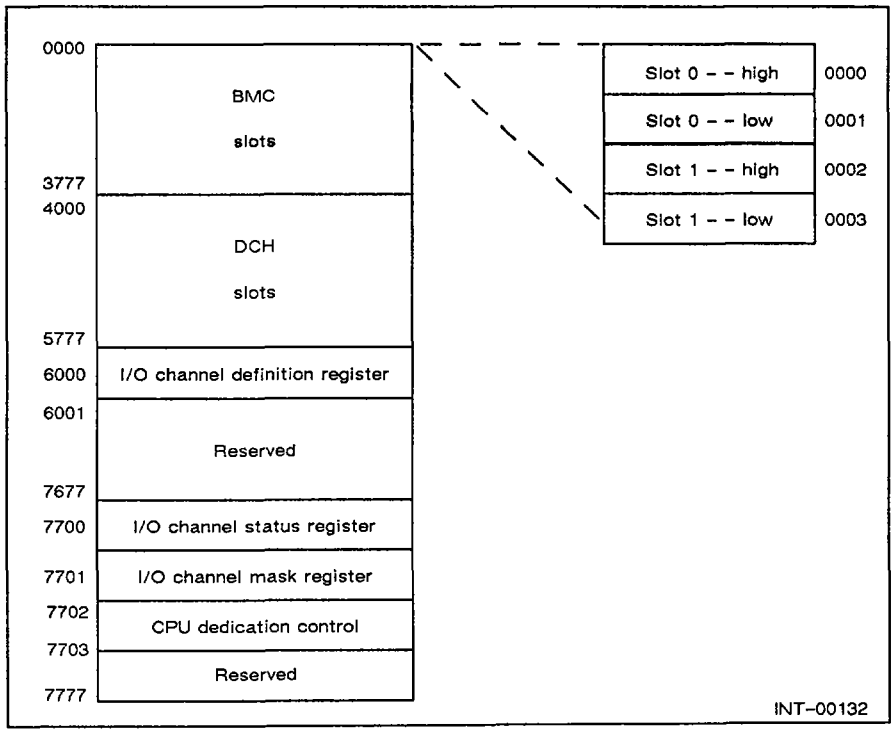
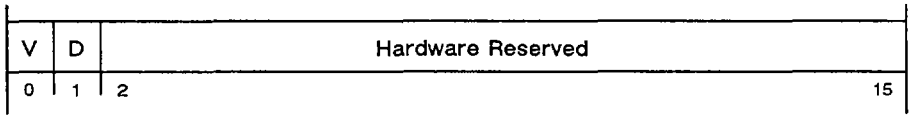


Figure 8-6 *DCH/BMC registers*

BMC/DCH Even-Numbered Register Formats

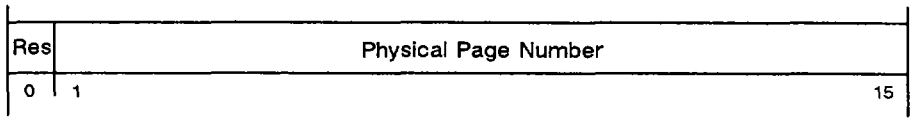
The processor translates the contents of the BMC and DCH even-numbered address registers ($0000–3776_8$ and $4000–5776_8$, respectively) as diagrammed below.



Bits	Name	Contents or Function
0	V	Map validity bit If 0, access to page is allowed. If 1, processor denies access.
1	D	Data bit If 0, the channel transfers data to device or memory. If 1, the channel transfers zeros to device or memory.
2–15	Hardware Reserved	Write to with zeros; reading these bits returns zeros

BMC/DCH Odd-Numbered Register Formats

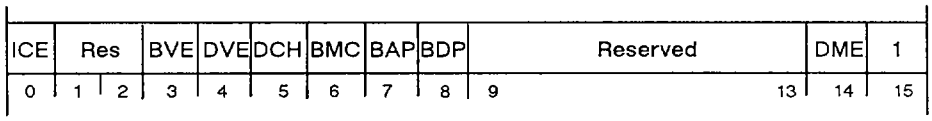
The processor translates the contents of the BMC and DCH odd-numbered address registers ($0000–3777_8$ and $4001–5777_8$, respectively) as diagrammed below.



Bits	Name	Contents or Function
0	Res	Hardware reserved; write to with zeros Reading these bits returns 0
1–15	Physical Page Number	Physical page number associated with the logical page that referred to that particular slot

I/O Channel Definition Register Format

The I/O channel definition register (6000_8) provides status information.

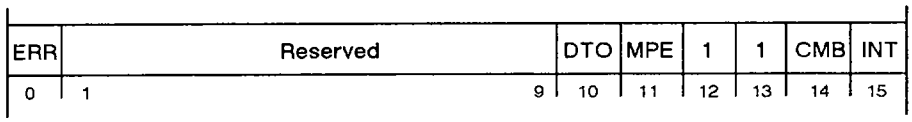


NOTE: *Writing to bits 3, 4, 7, 8, or 14 with a 1 complements these bits. The* **IORST** *instruction clears bits 3, 4, 7, 8, and 14.*

Bits	Name	Contents or Function
0	ICE	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	Res	Reserved for future use and returned as zero
3	BVE	BMC validity error flag: if 1, BMC address validity error has occurred
4	DVE	DCH validity error flag: if 1, DCH address validity error has occurred
5	DCH	Data channel transaction: if 1, a data channel transaction in progress
6	BMC	BMC transfer flag: if 1, 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
9–13	Reserved	Reserved for future use and returned as zeros
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	If 1, the I/O channel has detected an error indicated by the IOC status register or a memory parity error
1-9	Reserved	Bits 1 through 10 are reserved for future use
10	DTO	If 1, a DCH read-modify-write operation time-out error has occurred
11	MPE	If 1, 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 channel mask bit: if 1, prevents all devices connected to the channel from interrupting the CPU: however, the INTA instruction returns the device code of any device with its DONE flag set.
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 **INTA** instruction with a channel number returns on that channel the device code of the highest priority interrupting device, 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 Bus Reset **PRTRST** instruction sets the mask bit to a 0 for one channel 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.

Reserved		MK0	MK1	MK2	MK3	MK4	MK5	MK6	R
0	7	8	9	10	11	12	13	14	15

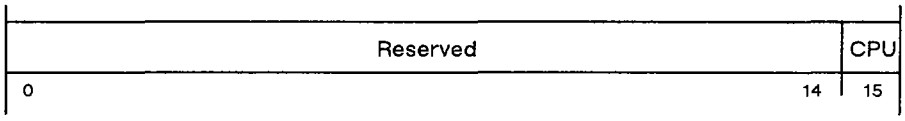
Bits	Name	Contents or Function
0-7	Reserved	Reserved for future use
8	MK0	If 1, prevents all devices connected to channel 0 from interrupting the CPU
9	MK1	If 1, prevents all devices connected to channel 1 from interrupting the CPU
10	MK2	If 1, prevents all devices connected to channel 2 from interrupting the CPU
11	MK3	If 1, prevents all devices connected to channel 3 from interrupting the CPU
12	MK4	If 1, prevents all devices connected to channel 4 from interrupting the CPU
13	MK5	If 1, prevents all devices connected to channel 5 from interrupting the CPU
14	MK6	If 1, prevents all devices connected to channel 6 from interrupting the CPU
15	Reserved	Reserved for future use

NOTE: *A system reset sets MK0 to 0, and MK1 through MK 6 to 1*

CPU Dedication Control

Each IOC contains a 16-bit command I/O register 7702_8, which controls CPU dedication. This read/write register is available only while the system is in dedicated mode.

*



Bits	Name	Contents or Function
0-14	Reserved	Reserved for future use.
15	CPU	CPU number (0 or 1). CPU 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 CPU. On execution of an **IORST** instruction, this number is set to the value of the CPU that issued the instruction.

DCH/BMC Map Instructions

The **CIO, CIOI,** and **WLMP** instructions initiate DCH/BMC map loads and reads when in mapped mode, with the **LPHY** instruction used for loads in unmapped mode. The I/O channel sets its BUSY flag to 1 when a map load or read is in progress. There is no DONE flag, and the channel never causes program interrupts. Table 8–12 lists the instructions that affect the DCH and BMC maps.

Table 8–12 *DCH/BMC map instructions*

Assembler Statement	Function
WLMP	Loads BMC/DCH map slots from memory
CIO, CIOI	Returns BMC/DCH status or loads map registers (1/2 slot) from accumulators
LPHY	Translates logical addresses to physical addresses
IORST	Clears bits 3, 4, 7, 8, and 14 of the I/O channel definition register, which disables data channel maps

Power Supply Controller

Device Code
4_8

Assembler Mnemonic
PSC (ECLIPSE MV/20000 Model 1 or 2)

Priority Mask Bit
13

The power supply controller (PSC) performs powerup diagnostic self–tests, monitors the system power, and reports failures, problems, and status information to an ECLIPSE MV/20000 series computer system. An MV/20000 CPU communicates with a PSC through the diagnostic remote processor, using device code 4 on the primary IOC.

The PSC provides power–up and power–down sequencing, I/O operations with ECLIPSE MV/20000 series systems, and output voltage margining, and it handles transfers to battery operation.

In addition, the PSC does the following:

- Monitors problems with the power supplies (such as overtemperature and overcurrent conditions)
- Monitors ac overvoltages and undervoltages
- Monitors overload on +5V
- Determines that the power switch was turned off
- Monitors battery back–up faults
- Monitors fan failures

Device Flag Control

Device flag commands to the power supply controllers determine the enabling or disabling of PSC interrupts.

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

Table 8-13 lists the I/O instructions that affect the power supply controllers.

Table 8-13 *I/O instructions for power supply controllers (PSC)*

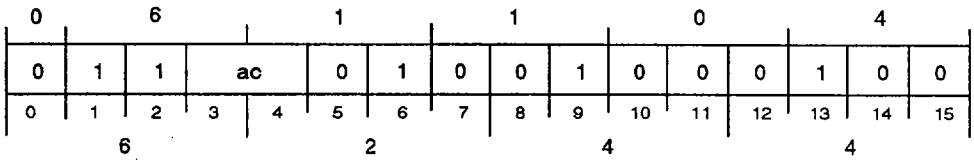
Assembler Statement	Function
DOA[*f*] *ac*,**PSC**	Write data to PSC
DIA[*f*] *ac*,**PSC**	Read data from PSC
IORST	Clears BUSY and DONE flags and interrupts priority mask bit
NIO[*f*] **PSC**	No-op and manipulate BUSY and DONE flags

The **IORST** instruction is described earlier in this chapter. Note that the **DIB**, **DIC**, and **DOC** instructions to the power supply controllers are no-ops.

The sections that follow describe the I/O instructions used to communicate with the power supply controller.

Write Data to PSC

DOAS *ac*,PSC



Function: *ac* -> PSC
BUSY = 1, DONE = 0

Parameters: *ac* = Data -> unchanged

The Write Data to PSC instruction sends the contents of the accumulator to the power supply controller. When the **DOAS PSC** instruction is issued, the PSC busy flag is set until the PSC completes the write. After the power supply controller completes the operation, it resets the BUSY flag and sets the DONE flag.

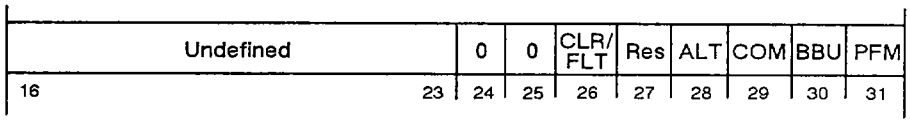
Arguments

ac(16-31) Bits 24 and 25 specify which register on PSC will receive the data. The following listing defines the four 16-bit registers that can be written to on the PSC.

ac Bits 24, 25	Name	Contents or Function
00	Control register	Selects reporting mode, power margining, and enable/disable battery back-up
01	Power margining register	When margining is enabled, the logic and memory voltages can be increased or decreased
10	Reserved	Reserved for future use and returned as zeros
11	Reserved	Reserved for diagnostic purposes and returned as zeros

The following diagrams and tables explain the function when a bit is set.

Control Register (Register 0)

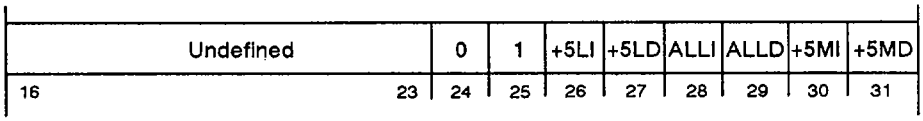


ac Bits	Name	Contents or Function
16-23	Undefined	Unused
24,25	0,0	The control register bits 24 and 25 equal zero
26	CLR/FLT	Clear fault code register
27	Res	Reserved for diagnostic purposes
28	ALT	Mask out powerfail interrupt. When ALT is 1, the CPU Skip instruction will still identify the state of the powerfail flag.
29	COM	PSC can interrupt computer when a fault occurs (0 disables all I/O interrupts from PSC)
30	BBU	Disables the battery back-up unit
31	PFM	Enable 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 programmed for margining.



ac Bits	Name	Contents or Function
16-23	Undefined	Unused
24,25	0,1	The power margining register bits 8 and 9 equal 01_2
26	+5LI	Increase +5V logic
27	+5LD	Decrease +5V logic
28	ALLI	Increase -5V and +12V logic
29	ALLD	Decrease -5V and +12V logic
30	+5MI	Increase +5V memory
31	+5MD	Decrease +5V memory

*

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

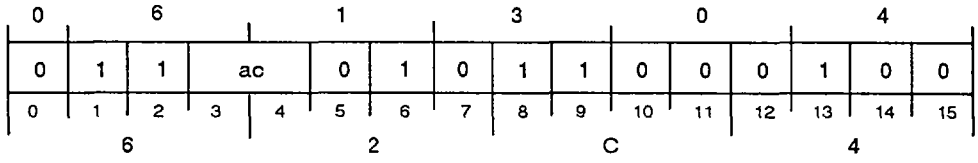
None

Exceptions

None

Request Data From PSC

DOAP *ac*,**PSC**



Function: Requests PSC data

Parameters: *ac* = Request -> unchanged

The Request Data From PSC instruction uses the specified accumulator to request specific information from the PSC.

Arguments

ac(28–31) Contains information on the PSC. Bits 0 through 27 are reserved and must be set to 0. The *ac* contents are defined in the following table.

Bits 28–31 (Octal Value)	Function
0	Read control bits
1	Read margining bits
2	Read power supply system status
3	Read (fault code register) latest fault code/status
4	Read PSC revision number (major code)
5	Read PSC revision number (minor code)
6	Second latest fault code/status
7	Third latest fault code/status
10	Return 252_8 (used for sizing)

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

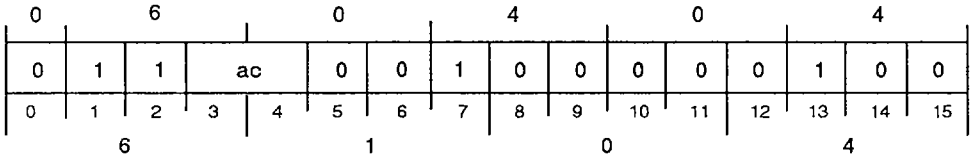
None

Exceptions

None

Read Data From PSC

DIA[*f*] *ac*,PSC



Function: PSC (A buffer) -> *ac*

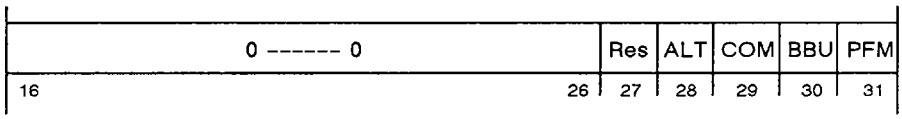
Parameters: *ac* = ? -> requested data

The Read Data From PSC instruction loads the data from the PSC A buffer into the accumulator. The instruction is used to read the latest fault code following an interrupt from the PSC or following a Request Data From PSC (**DOAP**) instruction.

Arguments

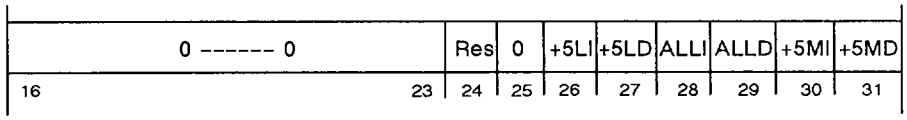
ac(16-31) After execution, contains data from the PSC A buffer. The following tables describe the results when a bit is set.

Read Control Bits



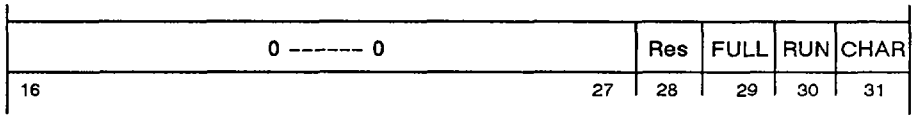
ac Bit	Name	Contents or Function
16-26	0 --- 0	Reserved and returned as zero
27	Res	Reserved
28	ALT	Powerfail interrupt is masked out
29	COM	PSC can interrupt the computer when fault occurs
30	BBU	The battery back-up unit is disabled
31	PFM	Power margining is enabled through program control

Read Battery Back-up and Margining Bits



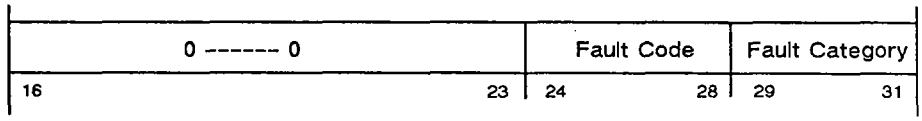
ac Bit	Name	Contents or Function
16-23	0 ---- 0	Reserved and returned as zero
24	Res	Reserved
25	0	Reserved and returned as zero
26	+5LI	+5V logic is increased
27	+5LD	+5V logic is decreased
28	ALLI	-5V and +12V logic voltages are increased
29	ALLD	-5V and +12V logic voltages are decreased
30	+5MI	+5V memory is increased
31	+5MD	+5V memory is decreased

Read Power Supply System Status



ac Bit	Name	Contents or Function
16-27	0 ---- 0	Reserved and returned as zero
28	Res	Reserved
29	FULL	The system is equipped with full battery back-up
30	RUN	The system is running on the batteries
31	CHAR	The batteries are recharging

Read Fault Code Register



ac Bit	Name	Contents or Function
16-23	0 --- 0	Reserved and returned as zero
24-28	Fault code	Specifies the fault code for a specific fault category
29-31	Fault category	Specifies the fault categories (ranging from 0 through 7)

NOTE: When *the PSC power system detects a fault, it loads the fault code and category into the fault code register and then flashes the code on the front panel LEDs. 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. See "Fault Codes" appendix for categorized lists of fault codes.*

[*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.
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

None

Exceptions

None

Multiple Central Processing Units

The ECLIPSE MV/20000 Model 2 system contains two central processing units. Each processor is identical in terms of architecture, features, options, and function, and each processor is independent of the other processor. These dual-processor computers are tightly coupled systems; their processors share the same memory subsystem. Each processor may also support its own floating-point processor (FPU); however, more than one processor may not share a single FPU. The ECLIPSE MV/Family instruction set includes a subset of multiprocessor instructions for interprocessor communication.

Since each processor runs independent of the other, programming multiple processors is generally identical to programming a single-processor system. Differences include the system initialization process, a processor's view of memory, and the interprocessor and I/O communications, which are described in the following sections.

Initialization

The processors are coequal 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 first. When the initial processor has passed all of its tests and has loaded its own control store, it passes control to another processor, which performs a similar initialization procedure. 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 that is booted first, has the default connection to the operator's console, has errors reported to it from the memory control unit, and receives all interrupts at powerup. In all other hardware aspects, the processors are equal in function.

In the following discussion, references to the initial processor are made for identification purposes only and are valid only when the processors are initialized. The following sequence starts at the system console:

- The initial processor is booted.
- The initial processor issues an **SSPT** instruction.
- The initial processor may then determine the state of the other processor by issuing a **JPSTATUS** instruction.
- If the initial processor decides to bring up the other processor (target processor), it
  - Issues a **JPLCS** instruction to the target processor. This continues until the target processor has loaded all of its writable control store. At this point, the target processor has the ability to respond to multiprocessor operations.
  - Places the starting PC value and other data, such as the accumulator values, into the appropriate sections of a processor state block. The initial processor then issues a **JPSTART** instruction to start the target processor.
- The state (running/stopped) of any processor may then be read with the **JPSTATUS** instruction.

Processor State Block

Each processor maintains a processor state block (also referred to as a JP state block) in physical main memory. This block contains information that 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 double-word aligned
- Be resident

The ECLIPSE MV/20000 Model 2 state block consists of 50 words in the following format:

Word #	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)
48-49	Physical JPLOAD/FLUSH 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 ION flag; 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. For further information, refer to the "Data Cache" section in Chapter 1.

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 that accesses memory, such as processors, IOCs, and (indirectly) I/O devices.

- Indivisible—The operation is performed as one atomic unit; in other words, no intermediate results of the operation are visible to another observer. To another observer, the operation appears to have never started or to have fully completed. The "Instruction Dictionary" describes those instructions that are indivisible.

  Certain store-type, memory-reference instructions are guaranteed to be indivisible if the target address is aligned to the data width (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.

- Uninterruptible—I/O interrupts are not honored during execution of these instructions, unless an exception occurs that forces a partial completion (such as a page fault).

- Serializable—Each operation goes to completion in all observers' views before the next operation begins. The result (processor state and 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:

- **ISZ, EISZ, XNISZ, LNISZ, XWISZ, LWISZ, DSZ, EDSZ, XNDSZ, LNDSZ, XWDSZ,** and **LWDSZ.**

- **SZBO, WSZBO,** and **WMESS.** (When 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 with respect to themselves. They are not serializable 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 programming example demonstrates how different processors have different memory views.

```
Time        Processor 0          Processor 1

 0          STA   0, 0, 0        STA   0, 1, 0
 1          XWLDA 0, 0, 0        XWLDA 0, 0, 0
```

Initially, ACO (processor 0) contains X', AC0 (processor 1) contains Y', and double-word 0 contains XY.

At Time 1, AC0 (processor 0) will get either X'Y or X'Y' and AC0 (processor 1) will get either XY' or X'Y'.

There is no serial execution of the two store instructions that gives either X'Y or XY'. If these instructions were serializable, either processor's AC0 would get X'Y'.

I/O Communication

The multiprocessor ECLIPSE MV/20000 series systems execute I/O instructions, such as CIO or PIO, from any of the processors residing on the system bus. All processors can make I/O requests to all devices on all IOCs (in a multiple-IOC configuration). I/O instructions that are broadcast to all channels, such as **MSKO,** are executed by all IOCs in the system simultaneously (the exception to this is an **INTA** instruction; refer to the section, "I/O Interrupt Handling" in this chapter).

Each IOC in the system maintains one 16-bit data input buffer for each processor. This allows the requesting processor to retrieve its input data at any time after the completion of the I/O command and eliminates the possibility of data overrun that could occur as a result of another processor's I/O activities.

Multiple I/O Channels

A multiprocessor ECLIPSE MV/20000 series system may support more than one IOC. Each I/O channel supports its own DCH and BMC. (In configurations with three IOCs, the third IOC supports only the DCH.) IOC0 contains the following integral devices:

Device	Code (octal)	Description
TTI	010	TTY input
TTO	011	TTY output
PSC	004	Power supply
SCP	045	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.

The primary asynchronous interface (TTI/TTO) will support either EIA RS-422 (differential) or EIA RS-232-C (single-ended). The following baud rates are available: 75, 110, 134.5, 150, 300, 600, 1200, 1800, 2000, 2400, 4800, 9600, and 19200.

I/O Interrupt Handling

A multiple-processor system handles I/O interrupts using a dedicated device mode. In this mode, each IOC dedicates itself to a particular processor. All interrupts generated by an IOC are directed to its processor so that a 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. The results of an **INTA** instruction, which is broadcast to all IOCs, are only sent to the IOCs dedicated to the issuing processor.

Each IOC contains a 16-bit command I/O register (7702_8), which controls processor dedication. The low-order bits of this register contain the processor number to which all NOVA type interrupts (except cross interrupts) will be directed.

Interprocessor Communication

The processors in multiple-processor configurations communicate using a set of privileged multiprocessor instructions. Communication 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. The ECLIPSE MV/20000 Model 2 Acceptable processor IDs for the ECLIPSE MV/20000 Model 2 are either 0 or 1.

A multiprocessor instruction executing on one processor affects one or more processors. Table 8-14 lists the multiprocessor instructions; the complete instruction descriptions are given at the end of this chapter. Some standard ECLIPSE MV/Family non-multiprocessor instructions affect multiple-processor systems to varying degrees; Table 8-15 lists these instructions. All other ECLIPSE MV/Family instructions (including I/O to the processor) executing on one processor affect only that processor.

Table 8-14 *Multiprocessor instructions*

Mnemonic	Description
CINTR	Request, remove, or query a request for a cross interrupt.
IMODE	Select the system-wide interrupt mode.
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 an attached 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 8-15 *ECLIPSE MV/Family instructions with multiprocessor functionality*

Mnemonic	Description
IORST	In addition to its normal functionality, the I/O reset instruction also resets IMODE to 0, clears all pending cross interrupts, and redirects all IOC traffic to the processor that issued the **IORST** instruction.
HALT	This instruction halts this processor and notifies the SCP. Note that a processor will not be halted unless all other processors are idle. Execution of the **HALT** instruction causes the processors to check if all the other processors are idle. If they are, then the processor that executed the **HALT** instruction will go directly to a halted state from its running state. The idled processors, when they see the last running processor go to a halted state, will also go to a halted state from their idle state. If all processors execute a **HALT** instruction, only one will be entering the halted state from the running state; all others must be coming from the idled state.
SSPT	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 the clocks (RTC and PIT) and multiprocessor synchronization within the state area. If the state area is moved, multiprocessor operations will be disrupted; cross interrupts and any pending messages (JPSTOP, JPFLUSH, JPSTATUS) will be lost. The RTC and PITs will also become invalid. Since the current mask of IOC0 is stored in this area, IOC0 will have to be re-masked out (MSKO).
PRTSEL	This instruction changes the default I/O channel on ALL processors.

Error Codes

Conditions that produce errors during execution of multiprocessor instructions return a value to AC1. Table 8–16 lists these errors.

Table 8–16 *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 that does not exist.
3	JPLCS	LCS error.
4	CINTR, JPLCS, JPSTART, JPSTATUS, JPSTOP	Processor failure. The processor to receive the request is not running.
5	JPFLUSH	No JP state block.
6	CINTR, IMODE	Illegal option.
7	JPSTOP	Processor not running.

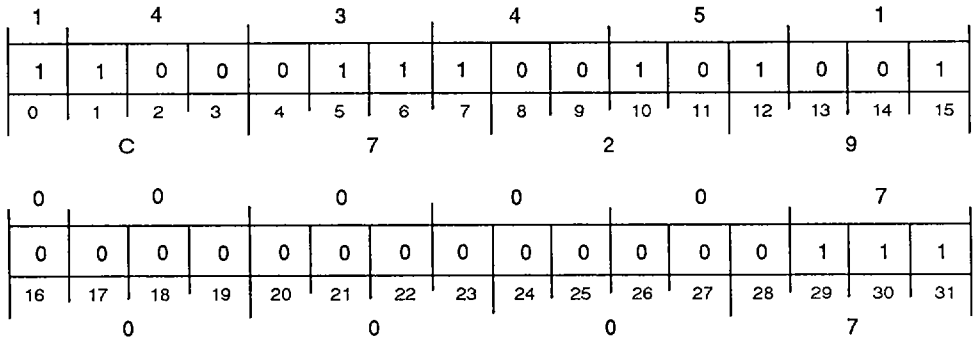
Multiprocessor Instructions

The following describes each of the ECLIPSE MV/Family instructions for multiprocessor communication. The instruction format is fully explained in the introduction to the chapter, "Instruction Dictionary."

Cross Interrupt Control CINTR

Multiprocessor Instruction

CINTR
(error return)
(normal return)



Function: Cross interrupt control (AC1)
PC = PC+3 (no error) PC+2 (error)

Parameters: AC0 = Processor ID --> unchanged
AC1 = Interrupt value [0 inquire] --> unchanged (normal)
[1 set] or code (error)
[2 clear]
AC2 = ? --> (if AC1 = 0) [0 no interrupts pending]
[1 interrupt pending]

NOTES: Cross interrupts are masked by channel 0, mask bit 3, channel 0, channel mask bit, or ION.

IORST clears all cross interrupts.

If an error is detected, AC1 contains an error code.

CINTR controls cross interrupts. **CINTR** either requests, removes, or queries a request for a cross interrupt. Cross interrupts respond to the **INTA** instruction with device code 77_8.

Arguments

None

Registers, Flags, and Stacks

AC0 Before execution, contains 32-bit ID of target processor.

After execution, contents unchanged.

AC1 Before execution, contains interrupt control value as follows:

Value	Description
0	Inquire interrupt pending state. (Are there any cross interrupts pending on target processor?)
1	Set cross interrupt on target processor.
2	Clear cross interrupt on target processor.

After execution, if no errors detected, contents unchanged. If error is detected, contains error code.

AC2	After execution, if AC1 initially contains 0 (inquiry), AC2 contains either 0 for no interrupt pending or 1 for interrupt pending. Otherwise, unchanged.
AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 3 (Normal return) PC + 2 (Error return)
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load immediate

Use these instructions to place appropriate values into AC0 and AC1.

Exceptions

If one of the following conditions is true, **CINTR** executes the next sequential word and returns an error code to AC1:

Condition	Error Code
Nonexistent processor	2
Processor failure	4
Illegal option	6

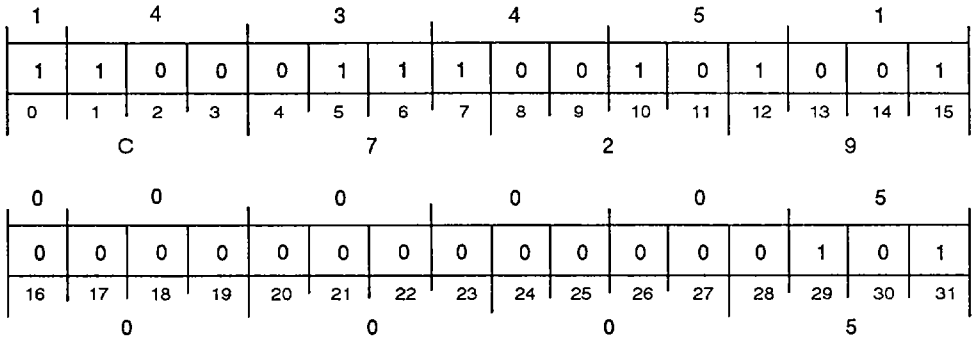
Cross interrupts are masked by channel 0, mask bit 3, channel 0, channel mask bit, or the ION flag.

All cross interrupts are cleared by the **IORST** instruction.

Select System Interrupt Mode — IMODE

Multiprocessor Instruction

IMODE
(error return)
(normal return)



Function: Select interrupt mode

Parameters: AC0 = Mode --> unchanged

NOTE: If AC0 = -1, then current mode --> AC0

IMODE selects the system-wide interrupt mode. If no errors are detected, **IMODE** skips the next sequential word.

Arguments

None

Registers, Flags, and Stacks

AC0 — Before execution, contains new mode value, as follows:

Number	Mode Selected
0	Dedicated (Currently, this is the only accepted mode of operation.)
-1	Retain current mode, and return mode number

After execution, contents unchanged. If contents initially -1, then contains current mode number.

AC1-AC3 — Unused

CARRY — Unchanged

Overflow — Unaffected

PC — PC + 3 (Normal return)
PC + 2 (Error return)

PSR — Unchanged

Stack — Unchanged

Related Instructions

Load immediate

Use these instructions to place a value into AC0.

Exceptions

If you specify an illegal option in AC0, **IMODE** executes the next sequential word and returns error code 6 to AC1.

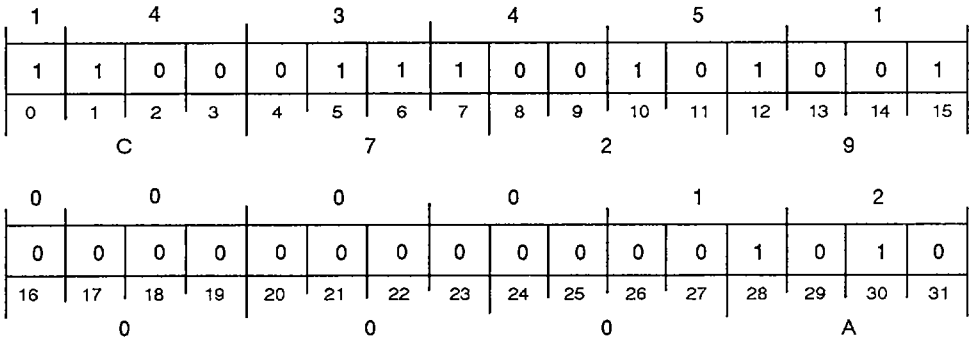
Upon a system reset, the interrupt mode is set to 0 (dedicated) with the I/O channels dedicated to the initial processor.

Upon the execution of an **IORST** instruction, the interrupt mode is set to 0 (dedicated) with the I/O channels dedicated to the processor that issued the **IORST** instruction.

Load State Block (no SBRs) JPFLOAD

Multiprocessor Instruction

JPFLOAD



Function: Load JP state block (no SBRs) @ (AC0)
Load wide stack registers

Parameters: AC0 = pointer to state block --> unchanged

NOTE: **JPFLOAD** does not load SBRs from the state block.

JPFLOAD loads the specified JP state block minus the segment base registers (SBRs) into the processor issuing the instruction. It then loads the wide stack registers from the appropriate ring. The state block is stored in internal processor state and used by subsequent **JPFLUSH** instructions. Use **JPFLOAD** to accelerate execution by eliminating the overhead of loading SBRs and purging the translation buffers.

Arguments

None

Registers, Flags, and Stacks

AC0	Before execution, contains pointer to JP state block.
	After execution, contains AC0 from state block.
AC1–AC3	After execution, contain AC1–AC3 from state block.
CARRY	After execution, contains CARRY from state block.
Overflow	Unaffected.
PC	After execution, contains PC from state block.
PSR	After execution, contains PSR from state block.
Stack	Wide stack registers are loaded from the appropriate ring.

Related Instructions

Load effective address
Use these instructions to place an address in AC0.

JPLOAD Load state block (with SBRs)

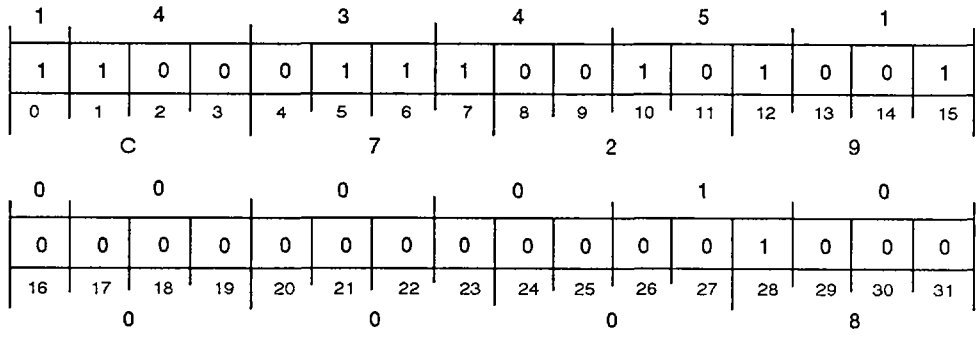
Exceptions

JPFLOAD does not load SBRs from the JP state block.

Flush State Block JPFLUSH

Multiprocessor Instruction

JPFLUSH
(error return)
(normal return)



Function: WSP, WFP --> memory
Fill JP state block
PC = PC + 3 (no error)
PC + 2 (error)

Parameters: None

JPFLUSH writes the wide stack pointer and wide frame pointer to memory and fills in the JP state block for the processor issuing the instruction. The address for the JP state block is the same address that the processor was **JPLOAD**ed from. If no errors are detected, **JPFLUSH** skips the next sequential word.

Arguments

None

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 3 (Normal return) PC + 2 (Error return)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WBR	Use this instruction to branch to an error routine.

Exceptions

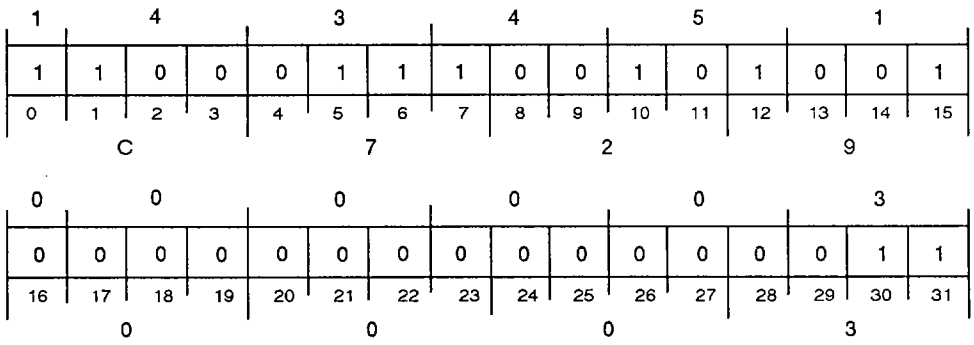
If no JP state block currently exists, **JPFLUSH** executes the next sequential word and returns error code 5 to AC1.

JPFLUSH will not write out information it knows has not changed since the last **JPLOAD** or **JPFLOAD**.

Return Processor ID JPID

Multiprocessor Instruction

JPID



Function: Processor ID --> AC0

Parameters: AC0 = ? --> Processor ID

JPID returns the processor ID of the processor that issued the instruction into AC0.

Arguments

None

Registers, Flags, and Stacks

AC0	After execution, contains 32-bit ID of processor.
AC1–AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

None

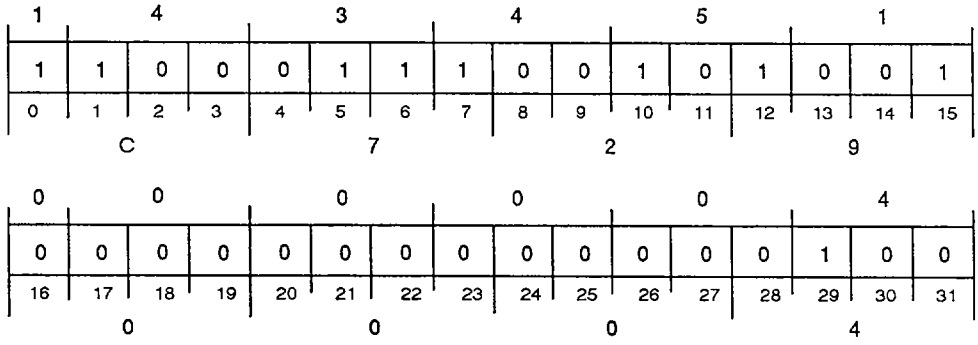
Exceptions

None

Load Control Store into JP JPLCS

Multiprocessor Instruction

JPLCS
(error return)
(normal return)



Function: Control Store --> Attached target processor
Perform LCS functions
PC = PC + 3 (no error)
PC + 2 (error)

Parameters: AC0 = Microcode options word/destination code --> unchanged
AC1 = Bit length word --> unchanged
AC2 = Data pointer --> unchanged
AC3 = Processor ID --> unchanged

NOTE: Possible errors returned to AC1 are the following: not stopped, LCS error, nonexistent processor, processor failure.

If loading control store causes an error, AC1 contains error code and the contents of AC0, AC2, and AC3 are undefined.

JPLCS loads writable control store into an attached processor. The target processor must be stopped. **JPLCS** works the same as the **LCS** instruction, using the same microcode blocks. For a complete description, refer to the appendix, "Load Control Store Instruction." The ECLIPSE MV/20000 Model 2 computer insures that microcode blocks will be 1 kiloword or less in length. **JPLCS** skips the next sequential word if no errors are detected during execution. **JPLCS** allows interrupts and is interrupt resumable.

Arguments

None

Registers, Flags, and Stacks

AC0 Before execution, contains double word as follows:

Bits	Contents	Description
0-15	Microcode Options	Specifies which microcode options to load. Bits 0-14 are reserved and should be set to 0. If bit 15 is set to 1, do NOT load microcode support for a hardware floating-point unit. Use this option when the system has an FPU that you do not want to use; for example, when you are running diagnostics.
16	Load/Verify Microcode	Specifies whether to load and verify microcode or verify only. 0 load and verify 1 verify only
17-31	Destination Code	Specifies where data is to be loaded.

After execution, contents unchanged. If an LCS error occurs, contains an LCS-specific error code, as follows:

Code	Meaning
1	Verify error
2	Illegal code
3	Unexpected (block type)
4	Illegal block length
5	Unknown destination
6	Illegal option

AC1(16-31) Before execution, contains bit length of code data.

After execution, contents unchanged. If an error occurs during execution, contains code indicating type of error.

AC2 Before execution, contains 31-bit pointer to first block of data (may be indirectable). Bit 0 must be set to zero.

After execution, contents unchanged. If an **LCS** error occurs, contains either an **LCS**-type, error-dependent code or is unchanged.

AC3 Before execution, contains 32-bit ID of processor to receive writable control store.

After execution, contents unchanged. If an **LCS** error occurs, contains a pointer to the erring block.

CARRY Unchanged

Overflow Unaffected

PC PC + 3 (Normal return)
PC + 2 (Error return)

PSR Unchanged

Stack Unchanged

Related Instructions

Load immediate
Use these instructions to place values into AC0, AC1, and AC3.

Load effective address
Use these instructions to place an address into AC2.

LCS Load Control Store

Exceptions

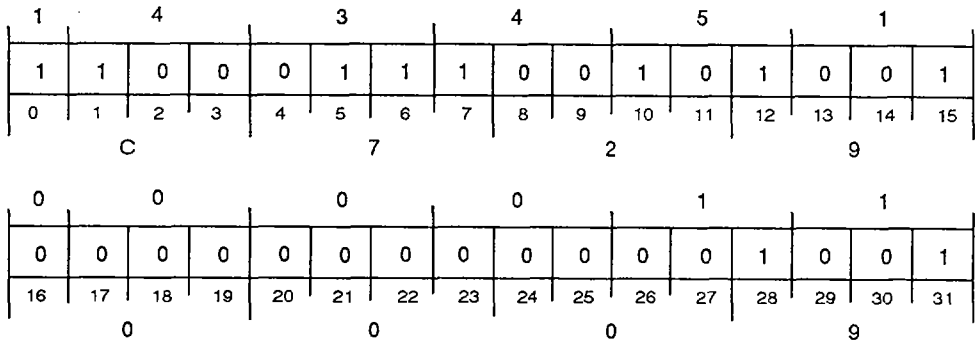
If one of the following conditions is true, **JPLCS** executes the next sequential word and returns an error code to AC1:

Condition	Code
Not stopped	1
LCS error	3 (see also Registers, Flags, and Stacks above)
Nonexistent processor	2
Processor failure	4

Load State Block JPLOAD

Multiprocessor Instruction

JPLOAD



Function: Load JP state block @ (AC0)
Load wide stack registers

Parameters: AC0 = pointer to state block --> unchanged

NOTE: **JPLOAD** does not load SBR0 from the state block.

JPLOAD loads the specified JP state block into the processor issuing the instruction and then loads the wide stack registers from the appropriate ring. The state block is stored in internal processor state and used by subsequent **JPFLUSH** instructions.

Arguments

None

Registers, Flags, and Stacks

AC0	Before execution, contains pointer to JP state block.
	After execution, contains AC0 from state block.
AC1–AC3	After execution, contain AC1–AC3 from state block.
CARRY	After execution, contains CARRY from state block.
Overflow	Unaffected.
PC	After execution, contains PC from state block.
PSR	After execution, contains PSR from state block.
Stack	Wide stack registers are loaded from the appropriate ring.

Related Instructions

Load effective address
Use these instructions to place an address in AC0.

JPFLOAD Load state block (without SBRs).

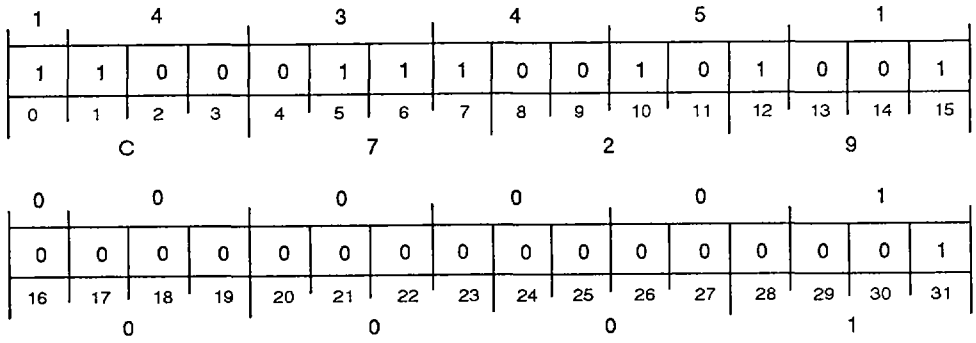
Exceptions

JPLOAD does not load SBR0 from the JP state block.

Start Another Processor — JPSTART

Multiprocessor Instruction

JPSTART
(error return)
(normal return)



Function: Start processor (AC0)
Load state @ (AC1)
PC = PC + 3 (no error)
PC + 2 (error)

Parameters: AC0 = Processor ID --> unchanged
AC1 = JP state block address --> unchanged (normal)
or code (error)

NOTE: The executing processor converts JP state block address to physical address. Possible errors returned to AC1 are the following: processor failure, not stopped, nonexistent processor.

JPSTART requests a processor to continue from a stopped state. The target processor loads its state from the JP state block pointed to by AC1. The **JPLOAD/JPFLUSH** address is unchanged.

Arguments

None

Registers, Flags, and Stacks

AC0	Before execution, contains 32-bit ID of processor to start.
	After execution, contents unchanged.
AC1	Before execution, contains address of JP state block that target processor will use to load its state.
	After execution, if no errors detected, contents unchanged. If error is detected, contains code.
AC2, AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 3 (Normal return) PC + 2 (Error return)

PSR	Unchanged
Stack	Unchanged

Related Instructions

JPSTATUS This instruction may be used to determine when the processor actually enters the running state.

Load effective address
Use these instructions to place an address into AC1.

Load immediate
Use these instructions to place a processor ID into AC0.

Exceptions

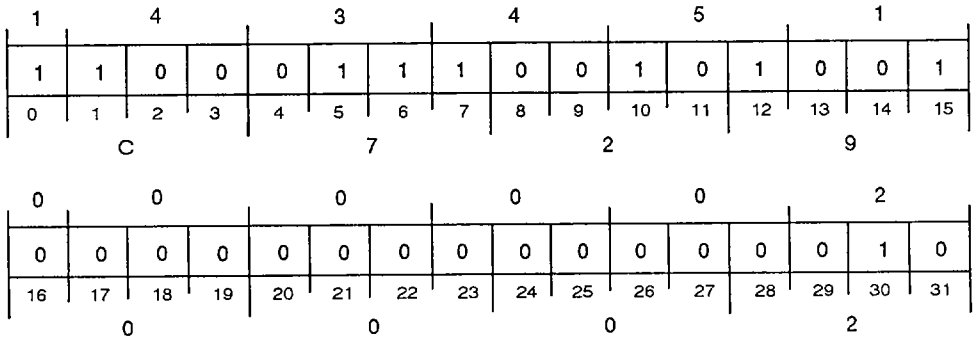
If one of the following conditions is true, **JPSTART** executes the next sequential word and returns an error code to AC1:

Condition	Code
Processor failure	4
Not stopped	1
Nonexistent processor	2

Return Processor Status JPSTATUS

Multiprocessor Instruction

JPSTATUS
(error return)
(normal return)



Function: Target processor data --> executing processor
PC = PC + 3 (no error)
PC + 2 (error)

Parameters: AC0 = Processor ID --> length/status
AC1 = ? --> Processor status (normal)
or code (error)
AC2 = ? --> Model/microcode
AC3 = ? --> JP state block pointer
or -1 (if no block)

NOTE: Possible errors returned to AC1 are the following: nonexistent processor, processor failure.

JPSTATUS returns information about a processor. The target processor does not need to be stopped. If the target processor has not had its writable control store loaded, the information will be approximate.

Arguments

None

Registers, Flags, and Stacks

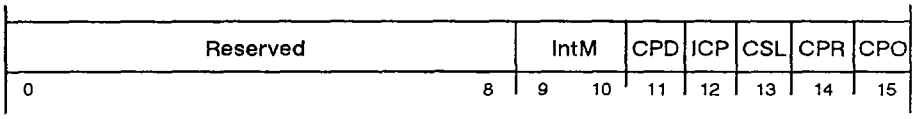
AC0 Before execution, contains 32-bit ID of target processor.

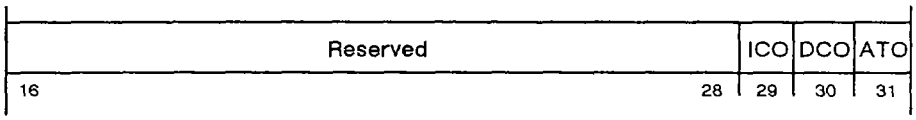
After execution, contains the length/status word of the target processor. Bits 0–15 contain the JP status block length; bits 16–31 contain the current target processor status. The following are valid status values:

Value	Status
0	Stopped
1	Starting
2	Running
3	Stopping
4	Waiting for writable control store

States 1 and 3 indicate to a faster processor that a particular implementation takes a longer time starting or stopping.

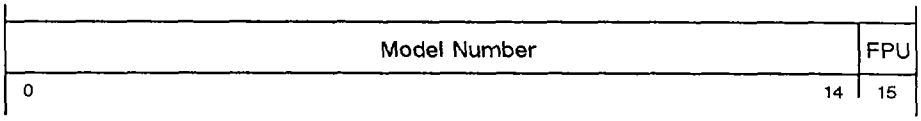
AC1 After execution, if error is detected, contains error code. If no errors detected, contains processor status word as follows:

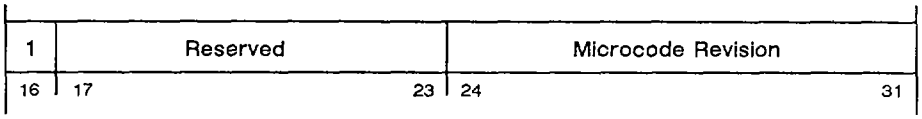




Bits	Mnemonic	Description (If bit is set to 1)
0-8	Reserved	Reserved for future use and is returned as zeros.
9, 10	IntM	Interrupt mode If 00, dedicated mode (Currently, this is the only acceptable value.)
11	CPD	The processor is degraded (system is 100 percent functional, but not totally healthy). *
12	ICP	This is the initial processor.
13	CSL	Control store is loaded.
14	CPR	The processor is running. This is equivalent to "processor is not in console."
15	CPO	The processor is OK.
16-28	Reserved	Reserved for future use and is returned as zeros.
29	ICO	The instruction cache is enabled.
30	DCO	The data cache is enabled.
31	ATO	The address translation cache is enabled.

AC2 After execution, contains the "Model/Microcode" double word of the target processor, as follows:





Bits	Mnemonic	Description (If bit is set to 1)
0-14	Model Number	Model number currently returned to AC0 by the NCLID instruction on the target processor.
15	FPU	There is a floating-point unit attached to the target processor.
16	1	Always set to one.
17-23	Reserved	Reserved for future use and is returned as zeros.
24-31	Microcode Revision	Microcode revision number currently returned to AC1 by the NCLID instruction on the target processor.

AC3 After execution, contains the physical address of the target processor's JP state block. If there is no JP state block, contains -1.

CARRY	Unchanged
Overflow	Unaffected
PC	PC + 3 (Normal return) PC + 2 (Error return)
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load immediate

Use these instructions to place a processor ID into AC0.

Exceptions

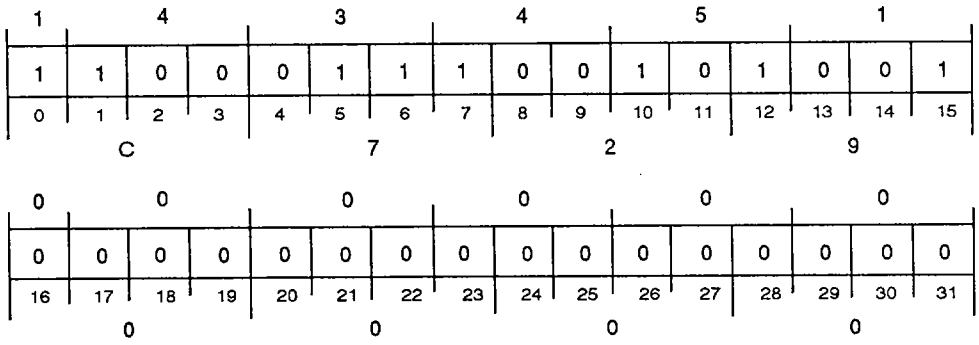
If one of the following conditions is true, **JPSTATUS** executes the next sequential word and returns an error code to AC1:

Condition	Code
Nonexistent processor	2
Processor failure	4

Stop Another Processor JPSTOP

Multiprocessor Instruction

JPSTOP
(error return)
(normal return)



Function: Halt processor (AC0)
Perform **JPFLUSH**
Write JP state block
PC = PC + 3 (no error)
PC + 2 (error)

Parameters: AC0 = Processor ID --> unchanged
AC1 = JP state block address --> unchanged (normal)
or code (error)

NOTE: Executing processor converts JP state block address to physical address.
Possible errors returned to AC1 are the following: processor failure, nonexistent processor, processor not running.

JPSTOP requests a processor to stop. A **JPFLUSH** function is performed, and the target processor will write a JP state block using the address in AC1. The **JPLOAD/JPFLUSH** address is unchanged. (Processors stop at interruptible points that are not masked by the ION flag.)

Arguments

None

Registers, Flags, and Stacks

AC0 — Before execution, contains 32-bit ID of processor to stop.

After execution, contents unchanged.

AC1 — Before execution, contains address that target processor will use to write its JP state block. Processor executing **JPSTOP** converts this to physical address.

After execution, if no errors detected, contents unchanged. If error is detected, contains error code.

AC2, AC3 — Unused

CARRY — Unchanged

Overflow	Unaffected
PC	PC + 3 (Normal return) PC + 2 (Error return)
PSR	Unchanged
Stack	Unchanged

Related Instructions

JPSTATUS This instruction may be used to determine when the processor actually stops.

Load effective address
Use these instructions to place an address into AC1.

Load immediate
Use these instructions to place a processor ID into AC0.

Exceptions

If one of the following conditions is true, **JPSTOP** executes the next sequential word and returns an error code to AC1:

Condition	Code
Processor failure	4
Nonexistent processor	2
Processor not running	7

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 details 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 13–31 of a segment base register specify a physical address of a page table in memory. Each segment is described by a page table, which occupies at least 2 Kbytes and begins on an integral 2 Kbyte boundary. A page table 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, page tables, 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 the chapter "Program Flow Management" for further information on protection fault handling.

The processor maintains one segment base register for each of the eight segments. The segment base register (SBR0 to SBR7) contains information which

- validates segment access.
- validates I/O access.
- specifies a one- or two-level page table.
- specifies for its segment the address of the first entry in the page table.

The segment base registers can be modified with the **LSBRA** and **LSBRS** privileged instructions, which load a block of double words from memory into the segment base registers.

NOTE: *Privileged instructions must be executed in segment 0; otherwise, a protection violation occurs.*

Figure 9–1 shows the format of a segment base register.

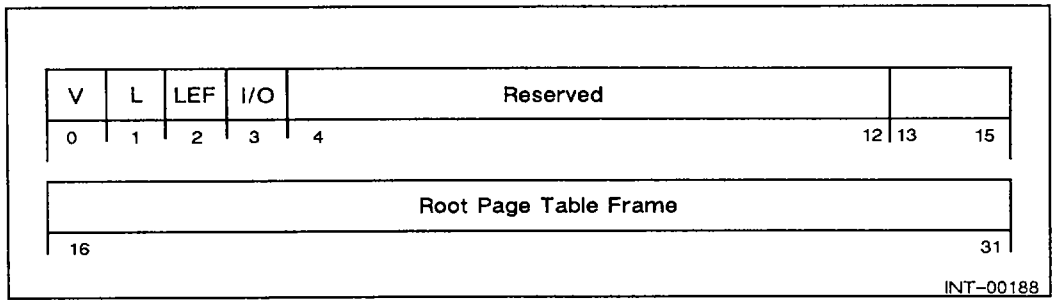


Figure 9-1 *Segment base register format*

In Figure 9-1,

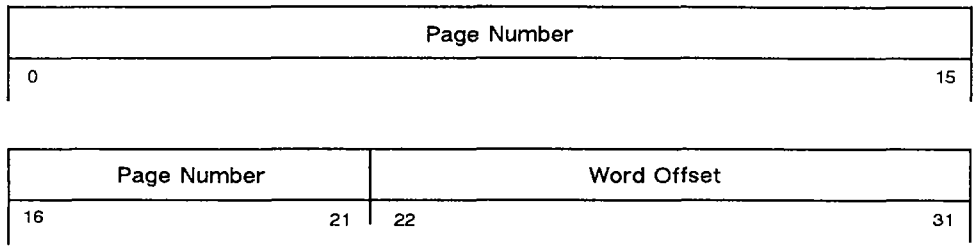
Bit	Name	Contents or Function
0	V	Segment-validity flag. The processor accesses the segment either to execute an instruction or to access data for an instruction that reads or writes data. The segment, however, must be a valid reference. The flag equals zero for an invalid segment. The processor aborts a memory reference instruction and services a protection violation when the logical address refers to an invalid segment. The flag equals one for 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 page table. The flag equals zero for a one-level page table. The processor can use a one-level page table with a program that requires 1 Mbyte or less of logical address space in the segment. A one-level page table entry contains the page table offset for the physical address translation. The flag equals one for a two-level page table. The processor must use a two-level page table with a program that requires from 1 Mbyte to 512 Mbytes of logical address space in the segment. A two-level page table entry contains the address of the second page table, which contains the page table offset for the physical address translation. Refer to the section "Page Table" for additional information on the page table. Refer to the section "Address Translation" for an example of using a segment base register and one or two page tables.
2	LEF	Mode flag. This flag controls the interpretation of the I/O or LEF opcode(s) (opcodes that begin with 011_2). The flag equals one for an LEF instruction interpretation. The processor executes the instruction as an LEF instruction. The flag equals zero for an I/O instruction interpretation. 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. When the flag equals zero, I/O operations are illegal from this segment. The processor aborts executing the I/O instruction and services the protection violation. The flag equals one for a legal I/O operation. The processor executes the I/O instruction.
4–12	Reserved	Reserved for internal DGC use.
13–31	Root Page Table Frame	Specifies the 19 most significant bits of the physical address for the root page table 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.

Pageframes

A pageframe address (or page number) is a page address shifted right ten bits.

A physical address has the format diagrammed below.



A page address is considered to be the page number with ten zeros following it. This page number is also the pageframe, 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}

Pageframe 1 corresponds to the page address 400_{16} and refers to the data words with addresses 400_{16} through $7FF_{16}$.

Page Tables

In each segment, the processor accesses a page table that specifies the status of the pages for the segment in memory. The page table manipulation instructions are Load Page Table Entry (**LPTE**) and Store Page Table Entry (**SPTE**). The page table contains an entry (PTE) for each page which

- indicates if a page is a valid access 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 page table entry.

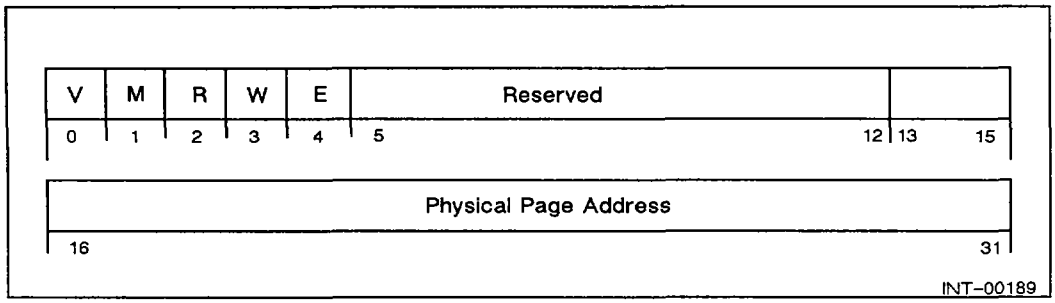


Figure 9–2 *Page table entry format*

In Figure 9–2,

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. The flag equals zero for an invalid page. The processor aborts the memory reference instruction and services the protection violation when the logical address refers to an invalid page. The flag equals one for 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 memory. The flag equals zero for a disk–resident page. 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. The flag equals one for a memory–resident page. The processor completes executing the memory reference instruction when the logical address refers to a memory–resident page.
2	R	Read–access flag. The flag equals zero for a page that the processor cannot access 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. The flag equals one for a page that the process can access 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 flag equals zero for a page that the processor cannot access 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. The flag equals one for a page that the processor can access for writing. Following the valid write access, the processor checks for a disk- or memory-resident page status.
4	E	Execute-access flag. The flag equals zero for a page that the processor cannot access 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. The flag equals one for a page that the processor can access for execution. Following the valid execute access, the processor checks for a disk- or memory-resident page status. NOTE: *The processor ignores the page access bits (bits 2-4) for a page table entry that addresses another page table, which occurs during a two-level page table translation.*
5-12	Reserved	Reserved for future use.
13-31	Physical Page Address	Identifies a page in memory. The physical page address refers to a page containing an instruction and/or data, or refers to a page containing the base of another page table, as determined by a one- or two-level page table 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 page table 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 zero, 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 one.

Figure 9-3 illustrates an indirect or an effective logical address for a one- and two-level page table. Refer to the chapter "System Overview" for an explanation of calculating an indirect or effective logical address.

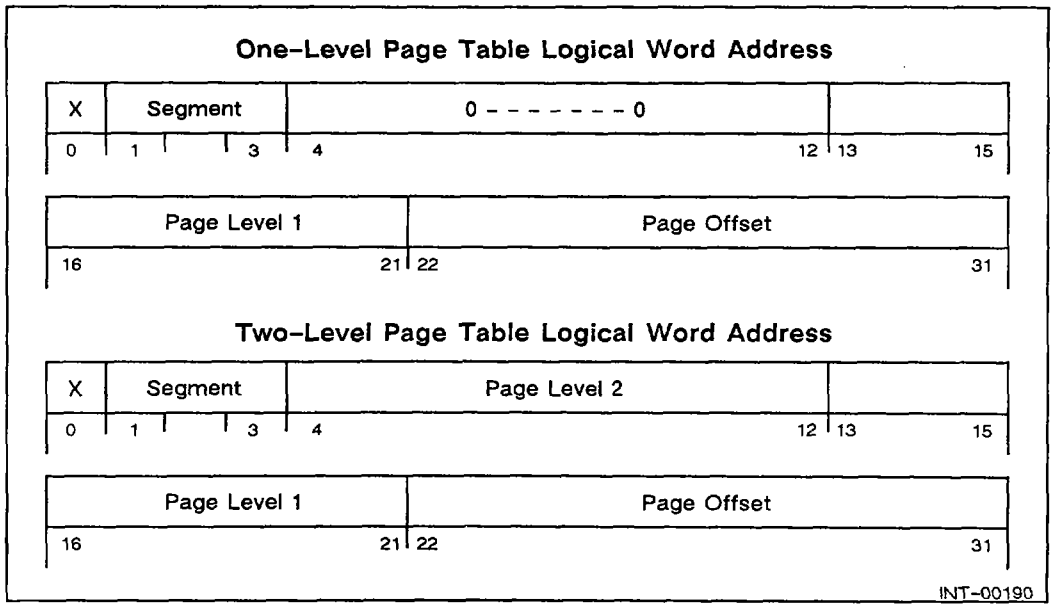


Figure 9-3 *Indirect and effective logical address formats*

In Figure 9-3,

X — The x bit (bit 0) is ignored by the processor when using direct addressing. The processor tests the x bit when using indirect addressing and continues to test it in subsequent indirect addressing until the bit equals zero.

Segment — The segment (bits 1-3) specifies one of eight segment base registers.

Page Level 2 — The page level 2 (bits 4-12) specifies an entry in the first of two page tables for a two-level page table translation. The page table entry contains the address of the second page table.

For a one-level page table translation, the page level 2 field (bits 4-12) must be all zeros. If they are not, the processor aborts the instruction and services a page table validity protection fault. Refer to the section "Protection Violations" in the chapter "Program Flow Management" for further information on protection fault handling.

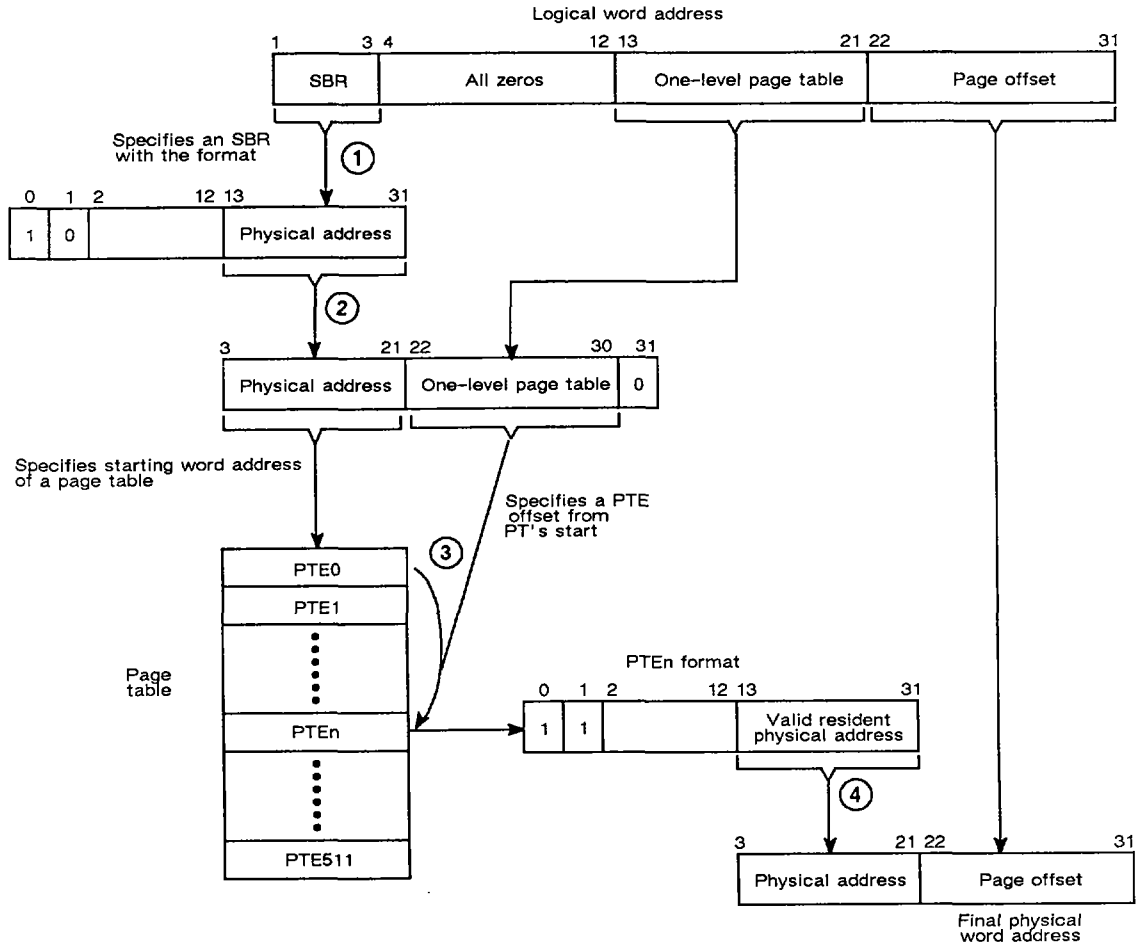
Page Level 1 — The page level 1 (bits 13-21) specifies an entry in a page table.

For a one- or two-level page table translation, the page table entry contains the address of the final page to be accessed for data or by an instruction.

Page Offset — The page offset (bits 22-31) specifies the final entry in the final page.

The page offset completes the address translation.

The section "Address Translation" presents examples of a one- and a two-level page table translation (Figures 9-4 and 9-5, respectively). The circled numbers labeling the accompanying paragraphs correspond to the circled numbers shown in Figures 9-4 and 9-5



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

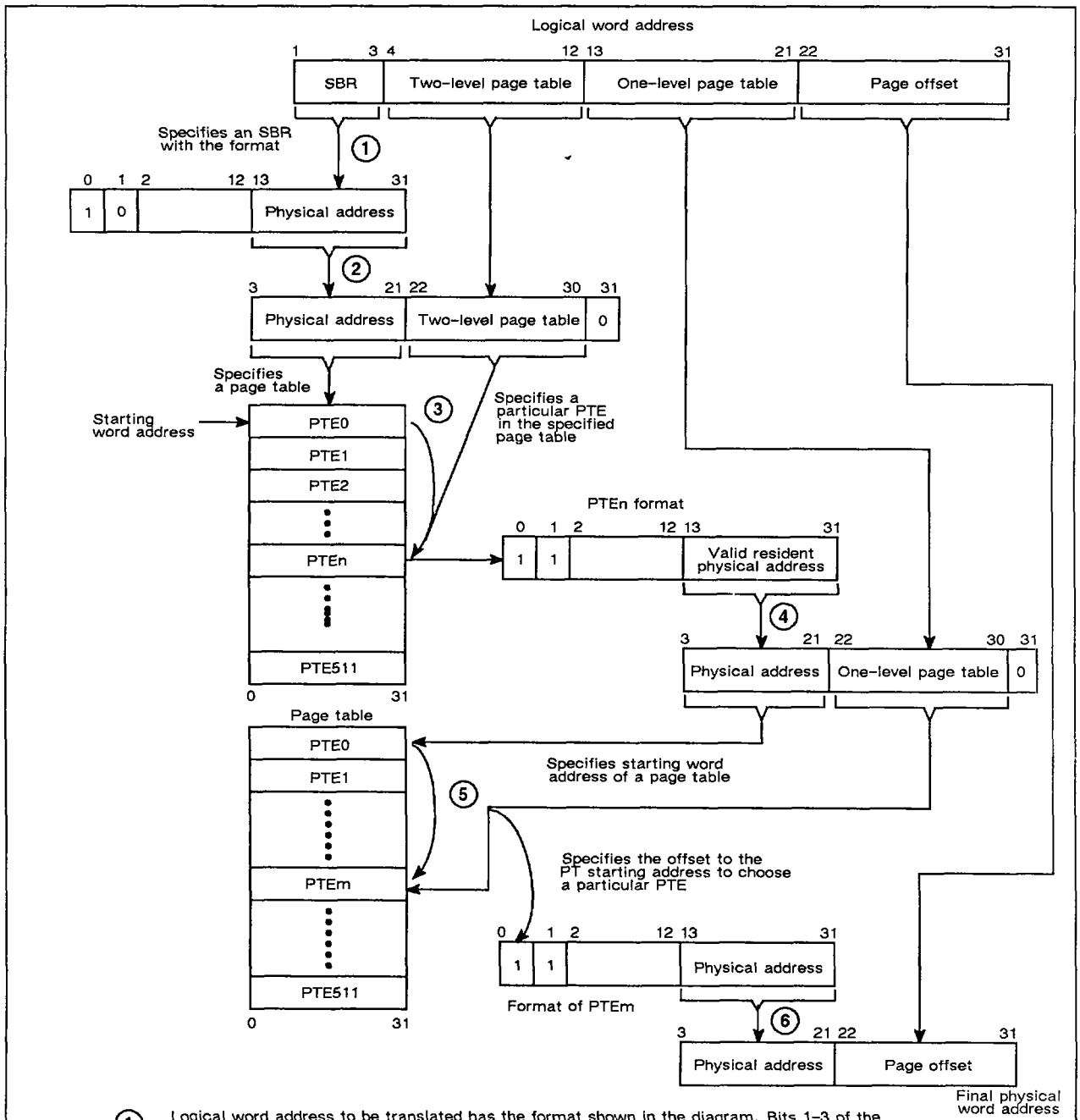
2. To form this physical page address, the processor begins with the physical address specified in bits 13–31 of the SBR. This address becomes bits 3–21 of the PTE address. Bits 13–21 of the logical word address becomes bits 22–30 of the PTE address. The processor appends a zero to the right of the PTE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

3. Bits 3–21 of the PTE address (unchanged in 2 above) specify the starting address of a page table. Bits 22–31 of the PTE address specify an offset from the start of the table to some PTE (labeled PTEn in the figure). This PTE specifies the starting address of a page of memory.

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

INT-00191

Figure 9-4 *One-level page table translation*



1. Logical word address to be translated has the format shown in the diagram. Bits 1–3 of the word address specify one of the eight segment base registers (SBRs). The processor uses the contents of this valid SBR to form the address of a PTE.
2. To form this address, the processor begins with the physical address specified in bits 13–31 of the SBR. This address becomes bits 3–21 of the PTE address. Bits 4–12 of the logical word address become bits 22–30 of the PTE address. The processor appends a zero to the right of the PTE address, making a 29-bit word address.
3. Bits 3–21 of the PTE address specify the starting address of a page table. Bits 22–31 of the PTE address specify an offset from the start of the table to some PTE (labeled PTEn in the figure). The PTE specifies the starting address of a page table.
4. The processor now constructs the address of a second PTE. The physical address specified in bits 13–31 of the first (PTEn) become bits 13–21 of the address of the second PTEm. Bits 13–21 of the logical word address become bits 22–30 of the second PTE's address. The processor appends a zero to the right of the second PTE address to make a 29-bit word address.
5. Bits 3–21 of the second PTE address specify the starting address of a second page table. Bits 22–31 of the second PTE address specify an offset from the start of the second table to some PTE (labeled PTEm in the figure). The second PTE specifies the starting address of a page. The page table containing PTEm can be paged itself. PTEn can indicate a nonresident page table.
6. The second PTEm's bits 13–31, the page address, become bits 3–21 of the physical address. The page offset specified in bits 22–31 of the logical word address becomes bits 22–31 of the physical address. This last value is the physical word address.

INT-00192

Figure 9–5 *Two-level page table 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 page table 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 (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 page table entry contain the access bits that specify any restriction.

If the reference to memory is for reading, the processor checks bit 2. A one in bit 2 indicates a valid read while a zero 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 one in bit 3 indicates a valid write, while a zero 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 one in bit 4 indicates a valid execute while a zero 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 on "Page Faults" in this chapter for detailed information.). If an instruction refers to a location that requires a two-level page table when only a one-level page table is allocated, then a protection violation occurs. Refer to the section "Protection Violations" in the chapter "Program Flow Management" for more information.

Referenced and Modified Bits

A referenced bit and a modified bit are associated with a physical page in memory. 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 one. A read or write operation occurs when the processor accesses memory without a protection fault occurring on a memory resident page.

NOTES: *A memory reference by the computer's I/O subsystem does not affect the flags' states.*

The referenced bit may or may not be set on page table pages when accessed by the processor to perform logical-to-physical address translations.

The referenced bit helps to determine which page in physical memory the page fault handler should replace with a new page from disk. The 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.

Table 9-1 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 9-1 *Instructions that manipulate referenced and modified bits*

Instruction	Operation
LMRF	Load modified and referenced bits
ORFB	OR referenced bits
RRFB	Reset referenced bits
SMRF	Store modified and referenced bits

Central Processor Identification

Central processor identification (CPUID) instructions store information about the processor parameters (such as the memory size and the microcode revision level) in one or more fixed-point accumulators. Table 9-2 lists the central processor identification instructions.

Table 9-2 *System identification instructions*

Instruction	Operation
ECLID, LCPID	Load CPU identification (AC0, bits 0-31)
NCLID	Narrow load CPU identification (AC0-AC2, bits 16-31)

Privileged Faults

Upon detection of a privileged fault, the address translator generates either a page or protection fault. A page fault occurs when the interpretation of the validity and appropriate access bits in a page table 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.

Page Faults

When a page fault occurs, the following actions result:

1. If the current segment is not 0, the processor 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. The processor 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 further details.
3. The processor initializes the segment 0 stack from page zero of segment 0.
4. The processor stores the fault code in AC1.

Fault Code	Explanation
0	Reserved
1	Reserved
2	Page table page fault
3	Reserved
4	Normal object reference

5. The processor disables interrupts for one instruction, jumps indirect through locations 30_8 and 31_8 of segment 0.
6. The processor 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 "Referenced and Modified Bits" section 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. The processor completes the memory reference and continues executing the instruction.

NOTES: *A page fault must not occur during steps 1 through 5; otherwise, the processor halts.*

Page zero of the current segment, and page zero of segment zero must always be resident. In addition, the context block and page fault handler must always be resident.

Figure 9-6 summarizes the actions taken upon detection of a page fault.

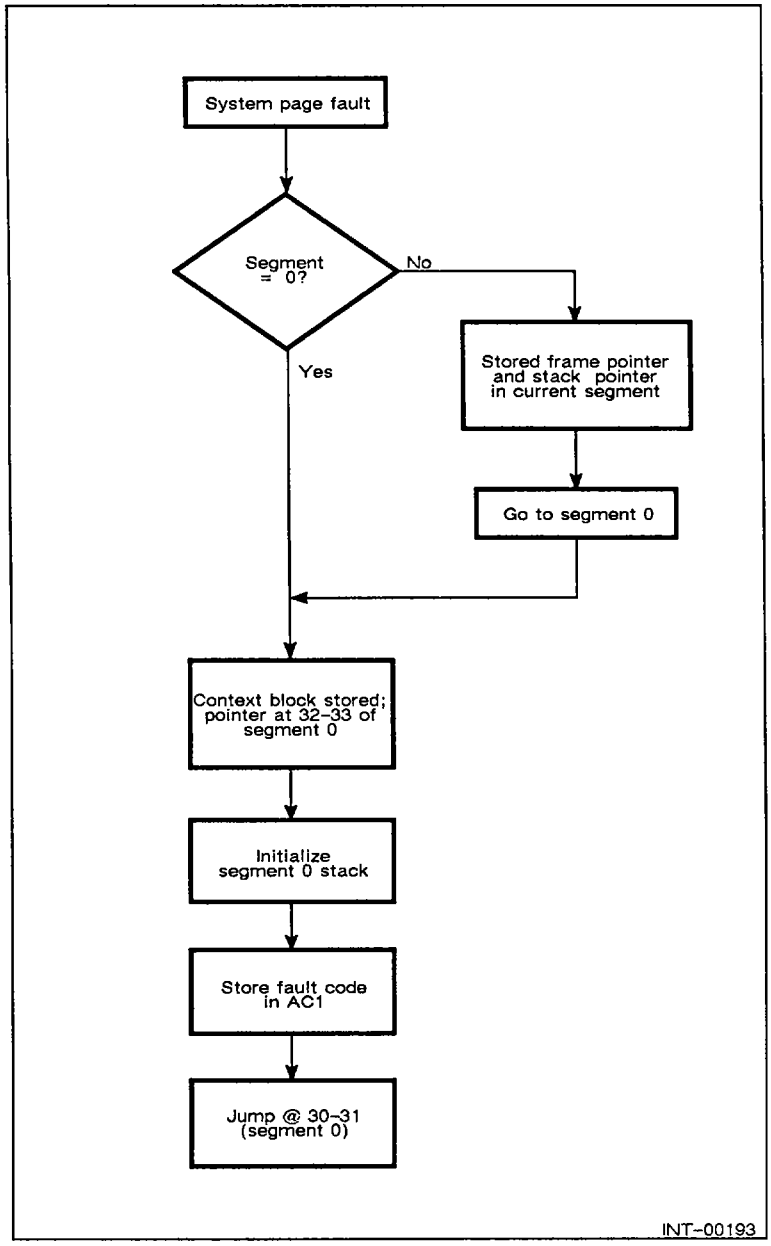


Figure 9-6 *Page fault sequence*

Address Protection Faults

When the address translator is enabled, the system checks for protection violation faults, listed in Table 9-3 below.

Table 9-3 *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	Page table validity violation
5	Read, write of execute access violation
6	Segment crossing violation

When a fault occurs, AC1 receives a code indicating the type of fault. (Refer to Fault Codes appendix.)

Reserved Memory

When a privileged/nonprivileged fault occurs, the processor transfers control to an appropriate fault handler. A reserved storage location in page zero of each segment contains the starting address of the fault handler.

The processor interprets segment 0, page zero locations differently from segments 1 through 7, page zero locations. For example, segment 0 contains pointers to privileged fault handlers, and segments 1 through 7 reserve these locations. Segment locations are described in the Reserved Memory appendix.

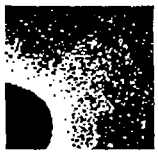
In addition, some ECLIPSE MV/Family computers require that a contiguous block of main memory be allocated to the processor for control purposes. This *state area* is available for use by the processor as hardware reserved memory. The operating system must consider the area as unusable memory.

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 operating system must execute **SSPT** at system initialization time, before the address translator is enabled. After execution, the state area is available for use by the processor.

SSPT places 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.

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 **SSPT**, as a final step.



10

ECLIPSE 16-Bit Programming

The 32-bit processor executes 16-bit processor instructions to provide upward program compatibility and to develop 16-bit programs (e.g., ECLIPSE C/350 processor). This chapter discusses both capabilities as well as machine-specific restrictions.

Programs that include ECLIPSE 16-bit memory- and stack-referenced instructions must meet certain requirements or restrictions explained in this chapter.

Refer to the *ECLIPSE C/350 Principles of Operation* manual for an explanation of C/350 instructions, terms, and conventions.

ECLIPSE 16-Bit Registers

The following ECLIPSE registers are implemented on MV/Family computers.

- Four 64-bit floating-point accumulators.

  The ECLIPSE floating-point accumulators are identical to the 64-bit processor floating-point accumulators.

- Four 16-bit fixed-point accumulators.

  The ECLIPSE fixed-point accumulator bits 0-15 correspond to the wide fixed-point accumulator bits 16-31. When it loads data into an accumulator, an ECLIPSE instruction alters bits 16-31, and leaves bits 0-15 undefined, unless otherwise noted. When using these instructions in a 32-bit environment, zero-extend (**ZEX**) or sign-extend (**SEX**) the results accordingly. **EJSR**, **ELEF**, **FRH**, **JSR** and **LEF** instructions are exceptions to this rule.

  An ECLIPSE instruction (e.g., a **CLM** instruction) does not alter the contents of an accumulator (bits 16-31) when it reads data from the accumulator.

  The ECLIPSE fixed-point accumulator bits 1-15 correspond to the wide accumulator bits 17-31 for accumulator-relative addressing.

- One 32-bit floating-point status register.

  The 32-bit floating-point status register (FPSR) corresponds to bits 0 through 15 and 49 through 63 of the FPSR in MV/Family systems.

- One 1-bit CARRY flag.

As executing ECLIPSE instructions does not generate fixed-point faults, they do not affect the processor status register. Certain ECLIPSE arithmetic instructions (**ADD**, **DIV**, etc.) set the state of the carry bit. To detect an appropriate fault, it is necessary to check the state of the carry bit upon completion of these instructions. A Carry from accumulator bit 16 affects MV/Family carry bits upon execution of these ECLIPSE instructions. The "Instruction Dictionary" that follows this chapter describes the ECLIPSE instruction set and those instructions which affect the carry bit.

The ECLIPSE program counter bits 1-15 correspond to the wide program counter bits 17-31. An ECLIPSE program flow instruction modifies bits 17-31, while the most significant bits are the current segment and zeros which are constrained to the first 64 Kbytes of memory (Figure 10-1). The following example demonstrates the use of a program flow instruction in both the lower 64 Kbytes and above that limit:

Lower 64 Kbytes: **JMP** .+1 (no-op)
Above 64 Kbytes: **JMP** .+1 (Resolves to some point in lower 64K memory)

NOTE: *Normal program flow and conditional skips are not constrained to the first 64 Kbytes of memory, such as:* **MOV# 0,0,SNR** *which produces a valid skip/no skip anywhere in memory.*

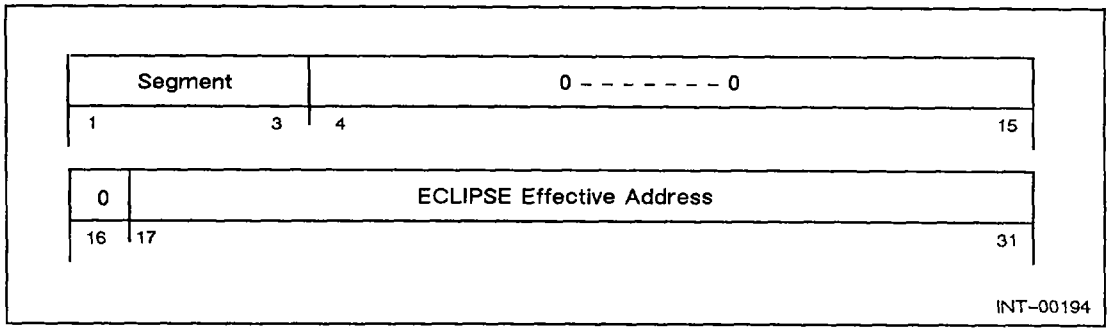


Figure 10-1 *ECLIPSE 16-bit program counter format*

In Figure 10-1,

Segment	Is the number of the current segment.
ECLIPSE Effective Address	Remains within the first 64 Kbytes of the segment.

The ECLIPSE instructions do not affect the processor status register.

ECLIPSE instructions function with the narrow stack, and thus use reserved memory locations in page zero for stack management without affecting stack management registers in MV/Family systems.

Register fields are illustrated in the "Register Fields" appendix.

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 memory) to define and control the narrow stack.

1. The narrow stack limit defines the upper limit of the narrow stack.

   Although specifying a 16-bit word, the narrow stack limit functions like the wide stack limit.

2. The narrow stack pointer initially defines the lower limit of the narrow stack.

   To enable narrow stack underflow, initialize the narrow stack pointer to 400_8 and start the narrow stack area at location 401_8

   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.)

3. The narrow frame pointer 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.

The ECLIPSE (or narrow) return block normally consists of five words: the contents of the least significant 16 bits of the four accumulators, the least significant 15 bits of the program counter or the frame pointer, and the carry in bit 0 of the last word pushed. 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.

ECLIPSE Faults and Interrupts

The 32-bit processor services (with the same pointers and fault handlers) the 16- and 32-bit floating-point and decimal/ASCII faults. For floating-point faults, the processor pushes a return block onto either the narrow or the wide stack, depending on the first instruction of the floating-point fault handler (a 16- or 32-bit instruction). For decimal/ASCII faults, the processor pushes a return block onto either the narrow or the wide stack, depending on bit 16 (32-bit accumulator) of the fault code in AC1. (Bit 16 equals one for ECLIPSE faults.) Thus, you can upgrade a program written for the 16-bit processor to incorporate 32-bit processor enhancements. Refer to the chapter "Program Flow Management" for more information on the fault handlers.

The 32-bit processor services (with the same pointer and interrupt handler) the 16- and 32-bit I/O interrupts. The processor pushes a return block onto either the narrow or the wide stack, depending on the first instruction of the I/O interrupt handler (a 16- or 32-bit instruction). Thus, you can upgrade a program written for the 16-bit processor to incorporate 32-bit processor enhancements. Refer to the Device Management chapter for more information on the interrupt handler.

The 32-bit processor and the 16-bit processor use different methods to flag an interrupted and resumable EDIT instruction. While the 16-bit processor sets AC0 to minus one (177777_8), the 32-bit processor sets a resume flag (IRES) in the PSR and checks the flag after completing the interrupt. For compatibility, the 32-bit processor also sets AC0 to minus one.

Expanding an ECLIPSE Program

A 16-bit program can be expanded by using a specific set of 32-bit instructions to

- Expand the program beyond 64 Kbytes.
- Use expanded data areas, such as large arrays.
- Utilize the 32-bit fixed-point arithmetic instructions.

Several methods are available to expand a 16-bit program beyond 64 Kbytes. The most reliable is to rewrite one of the subroutines so it contains 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 32-bit memory-reference instructions.

In addition, to expand data areas for large arrays or buffers, the processor must perform address calculations with 32-bit fixed integer arithmetic, and it must reference data with the 32-bit memory-referenced 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 reference 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 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 MV/32-bit routine using the changes listed in Table 10-1.

Table 10-1 *Alterations to ECLIPSE subroutines*

Changes to ECLIPSE Subroutine	Reason for Change
Replace **SAVE** and **RTN** with **WSSVS** or **WSSVR** and **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 instructions (such as **ADD**) are presented with the other 32-bit processor instructions.

The "Instruction Dictionary" and *ECLIPSE MV/Family Instruction Reference Booklet* identify the ECLIPSE instructions supported on the 32-bit processors.

Note that **NIO**[*f*] *ac*,**CPU** instructions (where ac <> 0) are reserved or assigned an 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 instructions supported on the 32-bit processors.

MV/Family Instruction Compatibility

In the MV/Family systems, ECLIPSE program flow instructions are limited to an address range of 64 Kbytes.

ECLIPSE instructions that load AC3 with the address of the next instruction (*jump to subroutine*) or push the address of the next instruction onto the narrow stack (*push and jump*) calculate effective addresses within the lower 64 Kbytes of the present segment.

The processor recognizes wide instructions supported by this machine via instruction opcodes. The instructions are a result of the ECLIPSE ALC no load-always skip opcode and the ECLIPSE **XOP** and **XOP1** opcodes (replaced by **X0PO**). ECLIPSE programs that contain these instructions cannot be used. The processor interprets these instructions as wide instructions, not as ECLIPSE instructions.

In addition, the 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**

ECLIPSE Memory Reference Instructions

The processor considers the ECLIPSE memory reference instructions to be 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. (See Figure 10-2.)

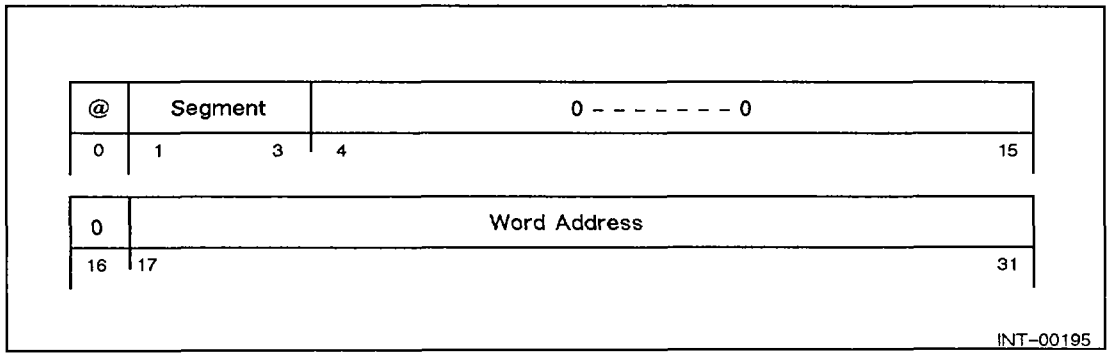


Figure 10-2 *ECLIPSE word addressing format*

In Figure 10-2,

@	Is an indirect bit in bit 0 which forces indirect addressing (when set to one) through a single word pointer.
Segment	Is the number of the current segment.
Word Address	Identifies a 16-bit word in the first 64 Kbytes of the current segment.

Figure 10-3 illustrates ECLIPSE effective addressing.

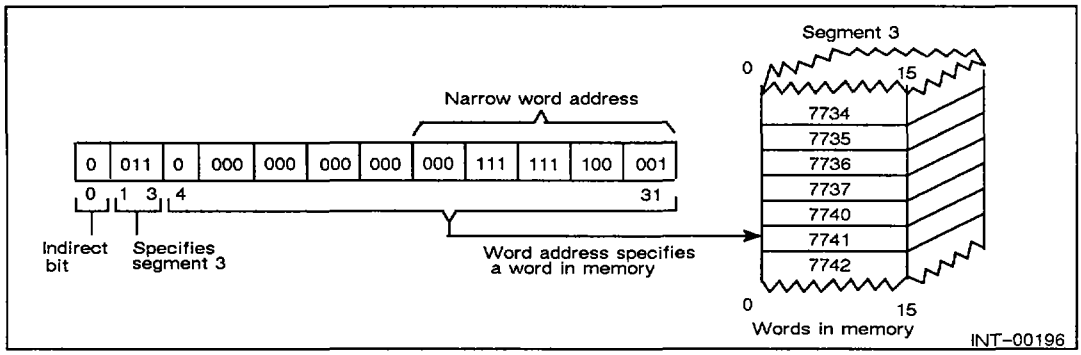


Figure 10-3 *ECLIPSE effective addressing*

To refer to a byte with an ECLIPSE memory reference instruction, the processor forms a byte address. (See Figure 10-4.)

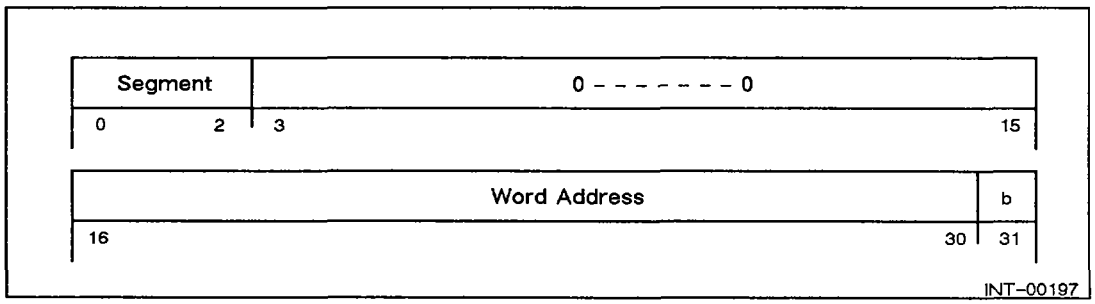


Figure 10-4 *ECLIPSE byte addressing format*

In Figure 10-4,

Segment	Is the number (in bits 0-2) of the current segment.
Word Address	Identifies a 16-bit word in the first 64 Kbytes of the current segment.
b	Is the byte (b) indicator in bit 31 which specifies the byte; set to one, b specifies least significant byte (bits 8-15 of a word).

Figure 10-5 illustrates ECLIPSE byte addressing.

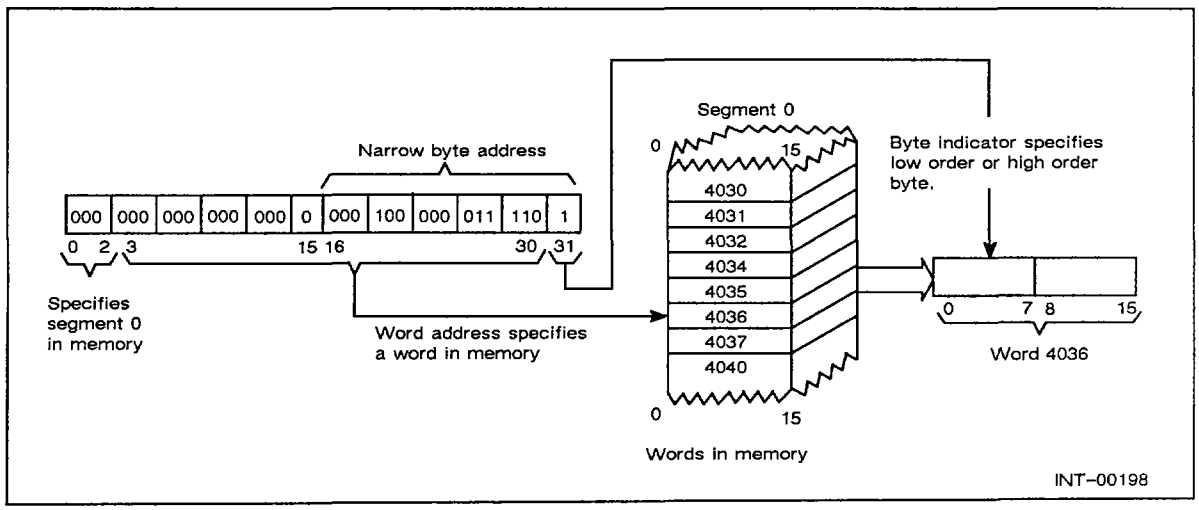


Figure 10-5 *ECLIPSE byte addressing*

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. 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.

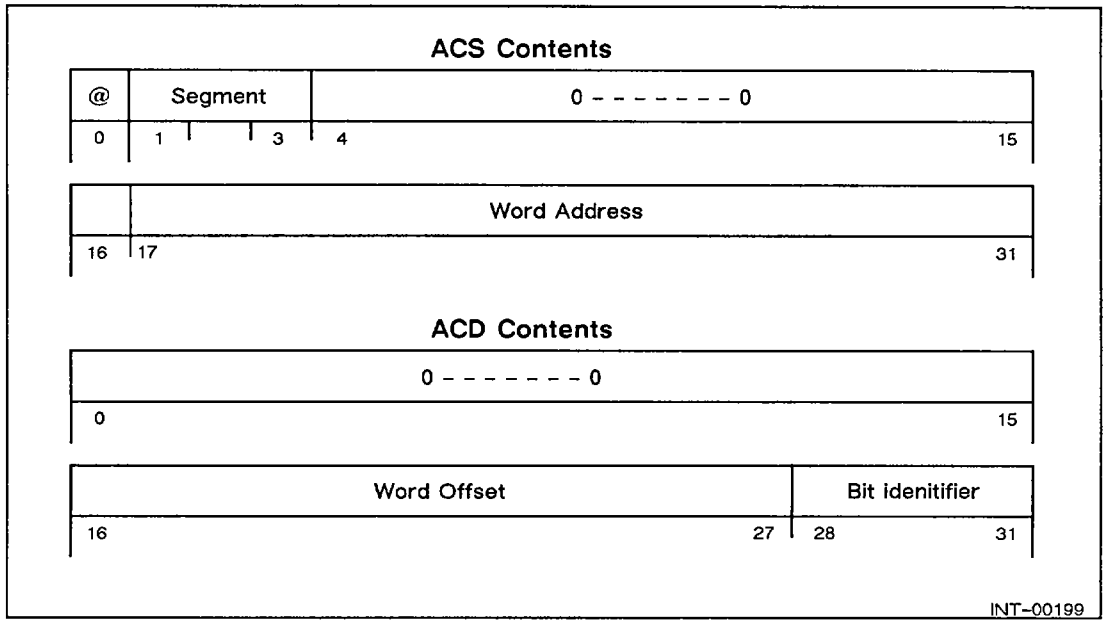


Figure 10-6 *ECLIPSE bit addressing format*

In Figure 10-6,

@	Is an indirect bit in bit 0 of the ACS accumulator which forces indirect addressing (when set to one) through a single word pointer.
Segment	Is the number of the current segment.
Word Address	Identifies a 16-bit word in the current segment.
Word Offset	The processor adds the word offset bits, an unsigned integer, to the effective address to determine a final word address.
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** and **BTZ** instructions, the processor limits effective addressing to the first 64 Kbytes of the current segment. If a bit instruction specifies the two accumulators as the same accumulator, then the effective address is zero 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 bit position (0-15) in the final word.

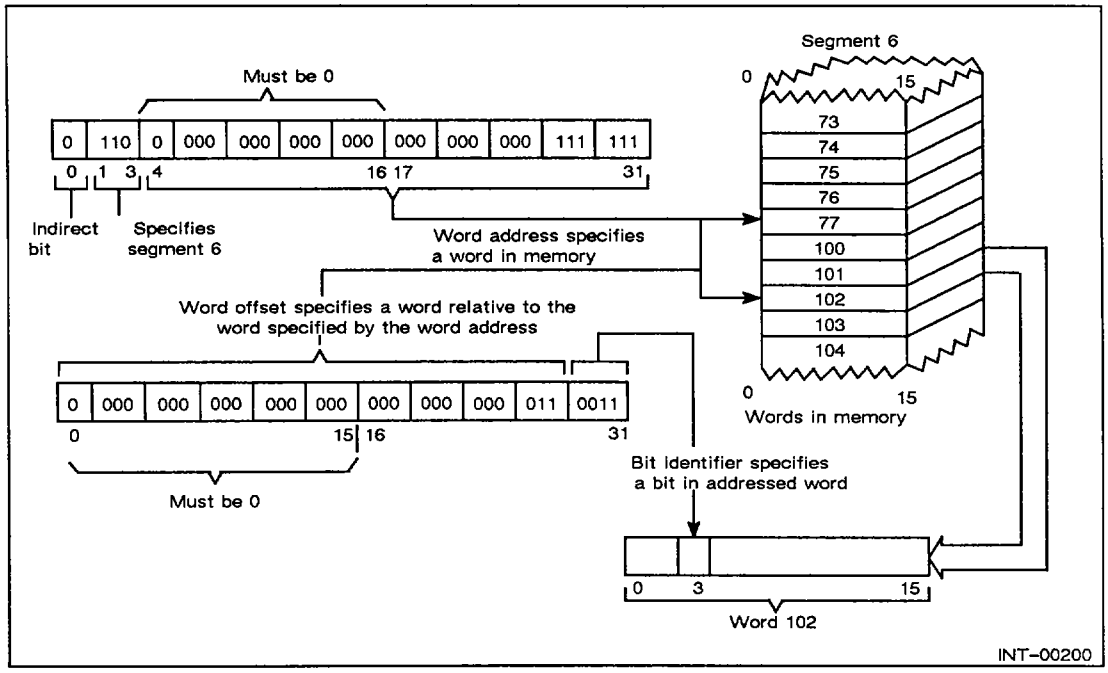


Figure 10-7 BTO, BTZ, SNB, SZB, *and* SZBO *bit addressing*

Fixed-Point Instructions

Table 10-2 lists the ECLIPSE fixed-point instructions that refer to memory. The table also shows an equivalent 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 32-bit processor equivalent instruction use identical

- Single- or double-word instruction length
- Argument string
- Data access for writing and for reading (register or memory)
- Data precision of 16 bits

An equivalent 32-bit processor instruction, however, uses a double word indirect pointer, while the ECLIPSE instruction uses a single word indirect pointer.

Table 10-2 *ECLIPSE fixed-point computing instructions*

ECLIPSE Instruction	Operation	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 32-bit processor equivalent instruction requires two words.*

Floating-Point Instructions

Table 10-3 lists the ECLIPSE floating-point instructions that refer to memory. The table also shows an equivalent 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 32-bit processor equivalent instruction use identical

- Single- or double-word instruction length
- Argument string
- Data access for writing and for reading (register or memory)
- Data precision of 16, 32, or 64 bits

An equivalent 32-bit processor instruction, however, uses a double word 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 MOF 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 10-3 *ECLIPSE floating-point computing instructions*

ECLIPSE Instruction	Operation	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** *instruction use a 32-bit accumulator, while the equivalent ECLIPSE instruction uses two memory words.*

** *The* **LFLST** *or* **LFSST** *instruction is a triple word instruction, while the ECLIPSE instruction is a double word instruction.*

Floating-Point Numerical Algorithms

The ECLIPSE floating-point loads (**FLDS, FLDD**) do not correct impure zero input. All loads simply move the memory operand to the specified FPAC. No normalization and correction to true zero is performed. The *Z* and *N* bits of the FPSR are set to reflect the loaded operand only if the operand is normalized. The *Z* and *N* flags are undefined if the operand is 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 of the FPSR is zero, the results of the floating-point computation performed on the 32-bit processor are identical to those obtained on the ECLIPSE.

Program Flow Instructions

Table 10-4 lists ECLIPSE program flow instructions that refer to memory. The table also lists an equivalent 32-bit processor instruction that can be substituted to expand (within the segment) the memory address range and to use the wide stack.

Unless otherwise stated, the ECLIPSE instruction and the 32-bit processor equivalent instruction use identical

- Single- or double-word instruction length
- Argument string
- Data access for writing and for reading (register or memory)

The data precision changes from 16 to 32 bits.

An equivalent 32-bit processor instruction, however, uses a double-word indirect pointer, while the ECLIPSE instruction uses a single-word indirect pointer.

Table 10-4 *ECLIPSE program flow management instructions*

ECLIPSE Instruction	Operation	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 OVK fixed-point overflow mask and use a return block of six double words.*

** *The* XVCT *and* WRSTR *instructions use the wide stack and are equivalent to* RSTR *and mode* E *of the* VCT *instruction.*

*** *The* XOP0 *and* WXOP *instructions are double-word instructions.*

Stack Instructions

Table 10-5 lists ECLIPSE stack instructions that refer to memory. The table also lists an equivalent 32-bit processor instruction that can be substituted to expand the memory address range (within the segment).

Unless otherwise stated, the ECLIPSE instruction and the 32-bit processor equivalent instruction use identical

- Single- or double-word instruction length
- Argument string
- Data access for writing and for reading (register or memory)

The data precision changes from 16 to 32 bits.

An equivalent 32-bit processor instruction, however, uses a double-word indirect pointer, while the ECLIPSE instruction uses a single-word indirect pointer.

Table 10-5 *ECLIPSE stack management instructions*

ECLIPSE Instruction	Operation	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**, *or* **WSSVR** *instructions modify the OVK fixed-point overflow mask and use a return block of six double words.*

** *The* **XVCT** *and* **WRSTR** *instructions use the wide stack and are equivalent to* **RSTR** *and mode* E *of the* **VCT** *instruction.*

*** *The* **XOP0** *and* **WXOP** *instructions are double-word instructions.*

Program Flow

The program counter governs program flow management as described in the chapter "Program Flow Management."

For any ECLIPSE program executing on an 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 wraparound to 0. The PC increments to the next value (such as 100000_8), and the processor executes the instruction at this location.

When using ECLIPSE **BAM, BLM, CMP, CMT, CMV, CTR,** and **EDIT** instructions, 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 the instructions listed in the previous paragraph move data in descending addresses and cross a segment boundary, a protection fault occurs. 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. Bits 1 through 3 remain unchanged; bits 4 through 16 are set to 0.

PC contents are illustrated in the "Register Fields" appendix.

Fault Handling

Fault handling is identical to the handling of MV/Family system nonprivileged faults as described earlier in this manual, and in the "Fault Codes" appendix. All faults which occur with the execution of ECLIPSE instructions use the narrow stack.

In addition, the 32-bit processor responds to floating-point traps upon completion of the floating-point instruction that caused the fault. In the ECLIPSE C/350, the response to a floating-point trap occurs when the next floating-point instruction is encountered. In either case, the value of the floating-point PC (FPPC) contains the address of the first floating-point instruction that caused a fault.

Note that an ECLIPSE commercial fault loads different information in 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.

When ECLIPSE **DIVS** or **DIVX** instructions produce a result of -32,768, the 32-bit processor sets CARRY to zero (meaning no overflow). The processor sets CARRY to one (meaning overflow) when these instructions are used on the ECLIPSE C/350. Wide divide instruction set overflow to zero when -32,768 results.

The "Fault Codes" appendix lists the error codes returned to AC1 when a decimal/ASCII fault occurs and denotes the type of fault generated.

Reserved Memory

MV/Family computers do not implement ECLIPSE auto-increment and auto-decrement locations 20_8 through 37_8. Processors in MV/Family computers reserve these locations to store certain system parameters.

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 "Register Fields" appendix.

11

Instruction Dictionary

The Instruction Dictionary presents the ECLIPSE MV/Family instruction set. The instructions appear in alphabetical order by the Assembler instruction mnemonic.

Instructions, which specify signed or unsigned values, assume the inputs are in the appropriate formats; otherwise results may be undefined.

The Instruction Dictionary uses the following abbreviations and symbols.

[]	The square brackets indicate an optional argument. Omit the square brackets when you include an optional argument with an Assembler statement. Square brackets also indicate a data indicator in shorthand "function and parameters" descriptions.
ac	Accumulator.
acs	Source accumulator.
acd	Destination accumulator.
fpac	Floating-point accumulator.
facs	Source floating-point accumulator.
facd	Destination floating-point accumulator.

lowercase *italic*	Lowercase *italic* characters indicate a variable argument in an Assembler statement. When you include the argument with an Assembler statement, substitute a literal value for the variable argument.
UPPERCASE **boldface**	UPPERCASE **boldface** characters indicate a literal argument in an Assembler statement. When you include a literal argument with an Assembler statement, use the exact form.
#	Unsigned value.
&	Treat these items as one.
(##-##)	Bit ## to ## of indicated word.
(E)	Contents of E.
*	Multiplication.
**	Raise to a power.
+	Addition or positive.
-	Subtraction or negative.
-->	Returns result to.
≠>	Does not return result.
/	Division.
2#	Signed integer.
<	Less than.
<-->	Swap contents.
=	Equal to.
≠	Not equal to.
?=?	Comparison
>	Greater than.
?	Undefined.
@	Indirection specifier.
@(AC#)	At address specified in AC#.
AND	Logical AND.
CRY	CARRY.
dec#	Decimal number.
displacement	Signed 8-, 15- or 32-bit integer.

E	Effective address.
fp#	Floating-point number. (fp#s=single precision, fp#d=double precision)
FPSR()	Floating-point status register (flags)
i	16-/32-bit signed or unsigned integer
index	Used by processor to determine if instruction specifies an absolute or relative addressing mode.
n	Integer from 1 to 4.
norm	Normalized floating-point number.
OR	Logical inclusive OR.
OVR	Overflow bit (PSR).
PC	Program counter.
PSR	Processor status register.
PTE	Page table entry.
R/W	Read or write.
SBR	Segment base register.
skip	If condition is true, skip the next word.
stack	Wide or narrow stack (instruction dependent).
wfp/fp	Wide/narrow frame pointer.
wsb	Wide stack base.
wsl/sl	Wide/narrow stack limit
wsp/sp	Wide/narrow stack pointer.
x	Unknown.
XOR	Logical exclusive OR.
$\overline{xxx}$	One's complement.

The following example and description presents the standard format for each instruction description.

(1) # Do Nothing Twice (Extended Displacement) XDUMME (2)

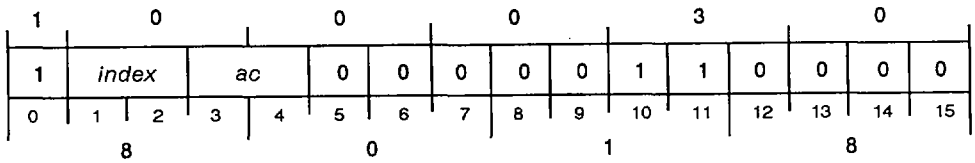
(3) Privileged Instruction

(4) **XDUMME** *ac,[@]displacement[,index]*
(exception return)

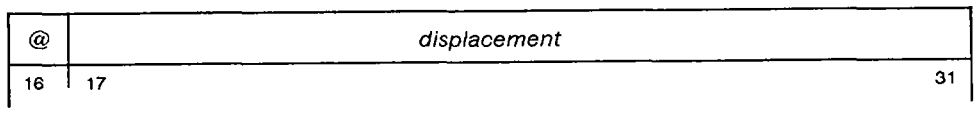
(4a) (normal return)

(5)

(6)



(7)



(8) Function: *ac* - 1 --> (E); (E) --> *ac*; *ac* + 1 --> *ac*
ALU CRY -> CRY

Parameters: None

NOTE: This instruction does not do much of anything.

(9) **XDUMME** subtracts one$_8$ from the value in the specified accumulator, calculates the effective address E, and stores the integer contained in *ac* at the location specified by E. **XDUMME** then loads the value at E back into *ac*, adding one$_8$ to this value.

(10) ## Arguments

ac Before execution, contains some value.
After execution, contains result of operation.

(10a) [@]*displacement*[,*index*]
Effective address generated by instruction is confined to current segment.

(11) ## Registers, Flags, and Stacks

AC0–AC3	Can be individually specified for *ac*; otherwise not used.
CARRY	Set with value of ALU CARRY
Overflow	1 if ALU overflow
PC	PC + 2 (exception return) PC + 3 (normal return)
PSR	Unchanged
Stack	Unchanged

(12) ## Related Instructions

LDA **LDA** loads a word from a memory location into an accumulator.

(13) ## Exceptions

If **XDUMME** does anything, a protection fault is generated.

(14) ## Example

```
LKBUSY: XDUMME              ;The execution of this routine produces an
        WBR LKBUSY          ;infinite loop.
```

(1) English translation of **Instruction**.

(2) Instruction **MNEMONIC** (**BOLDFACE CAPITAL** letters are used for the **MNEMONIC** throughout the instruction description).

(3) Special category of instruction.

Within this manual there are some special categories of instructions designated for ECLIPSE MV/computers:

Privileged Instruction -- 32-bit ECLIPSE MV/Family instructions executable only in segment 0.

ECLIPSE Instruction -- 16-bit (ECLIPSE C/350) compatible instructions.

Edit Sub-opcode -- Logically executable only within an Edit sub-program initiated by the **EDIT** or **WEDIT** instructions.

Intrinsic Instruction -- 32-bit ECLIPSE MV/Family instruction belonging to the optional Instrinsic Instruction Set (IIS).

Graphics Instruction -- 32-bit ECLIPSE MV/Family instruction belonging to the optional Graphics Instruction Set (GIS).

All other instructions are 32-bit ECLIPSE MV/Family instructions executable in any segment.

(4) Instruction **MNEMONIC** with *arguments* (*arguments* are *lowercase italic* throughout the instruction description).

(4a) Instructions capable of skipping the next 16-bit word indicate the conditions which update the program counter. Instructions which either load another value into the program counter or continue execution with the next sequential word omit this notation.

(5) Octal coding of the bit pattern.

(6) Bit pattern (*argument* bits are treated as zeros for the octal and hexadecimal codings).

(7) Hexadecimal coding of the bit pattern.

NOTE: *Bit boxes are presented as 16 bits per line regardless of the length of the instruction.*

Additional bit boxes may or may not include the octal/hexadecimal codings, dependent on their contents (such as, bit boxes which contain only a *displacement* have a default coding of 0_8 and 0_{16}).

(8) Abbreviated descriptions for Function, Parameters, and Notes. These are similar to the descriptions in the "ECLIPSE MV/Family Instruction Reference Booklet."

(9) Brief paragraph(s) describing the instruction action.

(10) Listing of *arguments* with a description of their contents before and after execution.

Instructions which use only certain portions of an argument utilize a modified format, such as, bits 16–31 of an accumulator are designated as *ac*(16–31). This holds true even if the result returned is greater than the initial value, e.g. *ac*(16–31) originally contains a 16-bit value, yet the final result is 32 bits.

In general, the result of an instruction execution retains the initial precision (32-bit floating-point addition produces a 32-bit floating-point result). Exceptions will be given in the instruction description.

Undefined after execution, or because of a fault condition, means that no information can be inferred from the results.

Accumulator arguments (*ac, acs, acd, fpac, fpacs, fpacd*) can be specified from AC0, AC1, AC2, or AC3 for fixed-point and from FPAC0, FPAC1, FPAC2, or FPAC3 for floating-point, unless specified otherwise.

(10a) The argument, [@]*displacement*[,*index*], is treated as a single entity within the descriptions. Upon instruction completion, memory locations remain unchanged unless specifically noted within the instruction description.

(11) Registers, Flags, and Stacks contains a description of the items affected by the execution of the instruction.

The term "Unused" indicates that an item is not used by the instruction and remains unchanged upon instruction execution.

The term "Unchanged" indicates that the contents of an item are used by the instruction and remain the same after instruction execution.

The term "Unaffected" is used to describe a temporary state (e.g., *Overflow*) that is not affected by the instruction.

The four fixed-point accumulators (AC0–AC3) and the four floating-point accumulators (FPAC0–FPAC3) are described individually only if they are affected by the instruction.

Though sometimes indicated as "Unused", the two fixed-point accumulators, which may be indexed for ac-relative addressing (AC2 and AC3), are available for use by those instructions which implement *index* as an argument.

The results of operations that complement CARRY for fixed-point ECLIPSE C/350-compatible instructions are:

Unsigned:	0 => integer > 65,535
Signed:	-32,768 > integer > 32,767

Overflow is a temporary condition (refer to the Fixed-Point Computing chapter). The results of operations that create an overflow condition for fixed-point, signed MV-specific instructions are:

Narrow:	-32,768 > integer > 32,767
Wide:	-2,147,483,648 > integer > 2,147,483,647

An *Overflow* condition may be reflected in another status bit, such as CARRY or the PSR(OVR) bit. In general, instructions which operate on unsigned values leave *Overflow* 0 or unaffected; instructions which operate on signed values affect *Overflow* from the ALU.

The Program Counter (PC) is always updated --

The value of the PC following instruction execution is calculated as the previous value plus the instruction length in 16-bit words.

Additional PC values are listed for possible variances, such as an interrupt value or a conditional skip value.

The Processor Status Register (PSR) and Floating-Point Status Register (FPSR) descriptions list only those bits affected by the instruction.

Stack usage is dependent on the type of instruction executing. ECLIPSE 16-bit instructions affect the narrow stack; 32-bit ECLIPSE MV/Family instructions affect the wide stack.

(12) Related Instructions lists those instructions in the following categories:

Generic -- helpful to implementing the present instruction (such as, load effective address-type instructions for those instructions requiring an address in an accumulator),

Specific Necessity -- deemed necessary for successful execution of a routine containing the present instruction (a Wide Do Until Greater Than instruction should be terminated with a Wide Branch instruction), or

Specific Related -- perform approximately the same function in a different manner or to a varying degree (the Skip on Valid Word Pointer instruction and the Skip on Valid Byte Pointer instruction are related.)

(13) Exceptions describes possible ways in which this instruction may produce a fault, may return some value to a register or flag, or should avoid being coded.

General exceptions not included for each instruction are:

Faults taken upon exception conditions, such as a protection fault for invalid addresses.

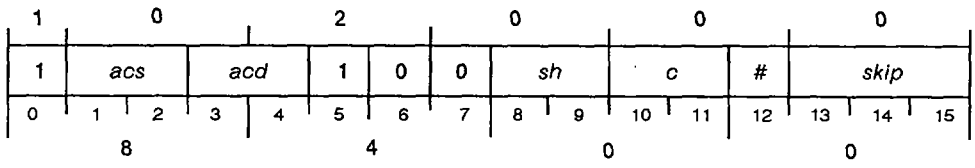
The majority of floating-point instructions do not check for unnormalized, or ill-formed, data. Operations on this data may produce erroneous results. Use **FNOM** to normalize data.

(14) Examples for each instruction range from a single line of code to complete subroutines.

Add Complement ADC

ECLIPSE Instruction

ADC[*c*][*sh*][#] *acs*,*acd*[,*skip*]
(*skip* false return)
(*skip* true return)



Function: $\overline{acs}$ + *acd* -> *acd*

Parameters: None

ADC initializes CARRY to the specified value, adds the logical complement of the unsigned 16-bit integer in *acs* to the unsigned 16-bit integer in *acd*, and places the result in the shifter. The instruction then performs the specified shift operation and loads the result of the shift into *acd* if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

Arguments

[*c*] Processor determines effect of CARRY flag (*c*) on initial value of CARRY before performing operation (opcode). Following table gives values of *c* and bits 10 and 11 and specifies operation.

Symbol [*c*]	Bits 10-11	Operation
omitted	0 0	Leave CARRY unchanged
Z	0 1	Initialize CARRY to 0
O	1 0	Initialize CARRY to 1
C	1 1	Complement CARRY

[*sh*] Processor shifts CARRY flag and 16 data bits after performing operation. Processor can shift bits left or right one bit position or can swap the two bytes. Following table gives values of *sh* and bits 8 and 9 and specifies shift operation.

Symbol [*sh*]	Bits 8-9	Shift Operation
omitted	0 0	Do not shift the result
L	0 1	Shift left
R	1 0	Shift right
S	1 1	Swap the two 8-bit bytes

[#] Except with no-load option (#), processor loads result of shift operation into destination accumulator. No-load option can test result of operation without destroying destination accumulator contents. Following table gives values of no-load option and bit 12 and specifies operation.

Symbol [#]	Bit 12	Operation
omitted	0	Load result into *acd*
#	1	Do not load result; restore initial CARRY flag

acs(16–31) Before execution, contains unsigned 16-bit integer.

After execution, contents unchanged unless *acs* and *acd* are same accumulator.

acd(16–31) Before execution, contains unsigned 16-bit integer.

After execution, contains result if no-load bit (#) is 0.

[*skip*] Processor skips next instruction if condition test true. Following table gives test conditions as specified by bits 13 to 15 and specifies operation.

Symbol [*skip*]	Bits 13–15	Operation
omitted	0 0 0	Never skip
SKP	0 0 1	Always skip
SZC	0 1 0	Skip if CARRY is 0
SNC	0 1 1	Skip if CARRY is not 0
SZR	1 0 0	Skip if result is 0
SNR	1 0 1	Skip if result is not 0
SEZ	1 1 0	Skip if either CARRY or result is 0
SBN	1 1 1	Skip if both CARRY and result are not 0

Skip omits next sequential 16-bit word. Make sure that *skip* does not transfer control to point within 32-bit or longer instruction.

Registers, Flags, and Stacks

AC0–AC3 Can be individually specified as *acs* and *acd*; otherwise unused.

CARRY If sum of two numbers being added is greater than 65,535, **ADC** complements initial CARRY. Then if left or right shift occurs, final resulting CARRY is bit shifted into CARRY.

Overflow 0

PC PC + 1 (false exit)
PC + 2 (true exit)

PSR Unchanged

Stacks Unchanged

Related Instructions

WADC Wide Add Complement

Exceptions

If result > 65,535, **ADC** complements CARRY. (See also CARRY)

Do not specify **ADC** with no-load option (#) in combination with either never skip or always skip option. Instruction may not end in 1000_2 or 1001_2 (reserved for other instructions).

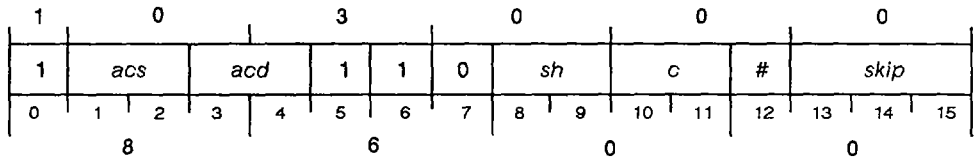
Example

```
ADCZ 1,2,SNR        ;Sets CARRY to zero, adds complement
                    ;of AC1 to AC2,
WBR ZEROANS         ;placing the result into AC2. If the result of
INC 2,2             ;the addition is not zero, next word is skipped.
```

Add ADD

ECLIPSE Instruction

ADD[*c*][*sh*][*#*] *acs,acd*[,*skip*]
(*skip* false return)
(*skip* true return)



Function: *acs + acd -> acd*

Parameters: None

ADD initializes CARRY to the specified value, adds the unsigned 16-bit integer in *acs* to the unsigned 16-bit integer in *acd*, and places the result in the shifter. The instruction then performs the specified shift operation and places the result of the shift in *acd* if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

Arguments

[*c*] Processor determines effect of CARRY flag (*c*) on initial value of CARRY before performing operation (opcode). Following table gives values of *c* and bits 10 and 11 and specifies operation.

Symbol [*c*]	Bits 10-11	Operation
omitted	0 0	Leave CARRY unchanged
Z	0 1	Initialize CARRY to 0
O	1 0	Initialize CARRY to 1
C	1 1	Complement CARRY

[*sh*] Processor shifts CARRY flag and 16 data bits after performing operation. Processor can shift bits left or right one bit position or can swap the two bytes. Following table gives values of *sh* and bits 8 and 9 and specifies shift operation.

Symbol [*sh*]	Bits 8-9	Shift Operation
omitted	0 0	Do not shift the result
L	0 1	Shift left
R	1 0	Shift right
S	1 1	Swap the two 8-bit bytes

[*#*] Except with no-load option (#), processor loads result of shift operation into destination accumulator. No-load option can test result of operation without destroying destination accumulator contents. Following table gives values of no-load option and bit 12 and specifies operation.

Symbol [#]	Bit 12	Operation
omitted	0	Load result into *acd*
#	1	Do not load result; restore initial CARRY flag

acs(16-31) Before execution, contains unsigned 16-bit integer.

After execution, contents unchanged unless *acs* and *acd* are same accumulator.

acd(16-31) Before execution, contains unsigned 16-bit integer.

After execution, contains result if no-load bit (#) is 0.

[*skip*] Processor skips next instruction if condition test true. Following table gives test conditions as specified by bits 13 to 15 and specifies operation.

Symbol [*skip*]	Bits 13-15	Operation
omitted	0 0 0	Never skip
SKP	0 0 1	Always skip
SZC	0 1 0	Skip if CARRY is 0
SNC	0 1 1	Skip if CARRY is not 0
SZR	1 0 0	Skip if result is 0
SNR	1 0 1	Skip if result is not 0
SEZ	1 1 0	Skip if either CARRY or result is 0
SBN	1 1 1	Skip if both CARRY and result are not 0

Skip omits next sequential 16-bit word. Make sure that *skip* does not transfer control to point within 32-bit or longer instruction.

Registers, Flags, and Stacks

AC0-AC3 Can be individually specified as *acs* and *acd*; otherwise unused.

CARRY If the sum of the two numbers being added is greater than 65,535, **ADD** complements initial CARRY. Then, if left or right shift occurs, final resulting CARRY is bit shifted into CARRY.

Overflow 0

PC PC + 1 (false exit)
PC + 2 (true exit)

PSR Unchanged

Stack Unchanged

Related Instructions

NADD Narrow Add

WADD Wide Add

Exceptions

If result > 65,535 **ADD** complements CARRY. (See also CARRY)

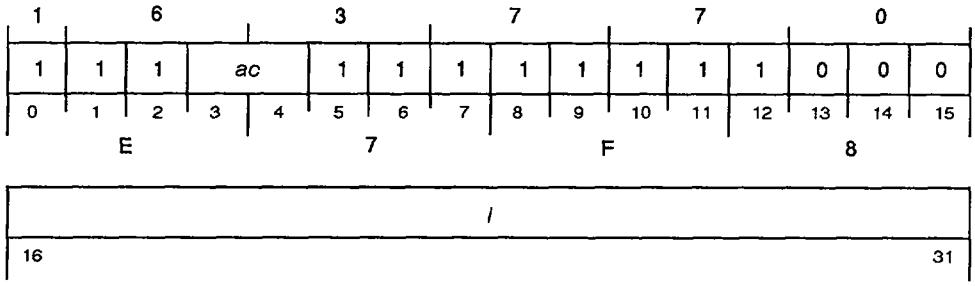
Do not specify **ADD** with no-load option (#) in combination with either never skip or always skip option. Instruction may not end in 1000_2 or 1001_2 (reserved for other instructions).

Example

```
ADDL 2,0,SNC        ;Adds the contents of AC2 to AC0,
                    ;placing the result in AC0.
WBR ERROR           ;Shifts the result left one bit, then skips
....                ;the next 16-bit word if CARRY is not zero.
```

Extended Add Immediate ADDI

ADDI *i,ac*



Function: *i + ac -> ac*

Parameters: None

ADDI adds a signed 16-bit integer in the range of -32,768 to +32,767 from the immediate field to the contents of an accumulator.

Arguments

i	Contains signed 16-bit integer.
ac(16-31)	Before execution, contains signed 16-bit integer.
	After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

ADI, NADI, WADI Add an immediate value to an accumulator.

NADDI, WADDI, WNADI Add a signed 16- or 32-bit immediate value to an accumulator.

Exceptions

None

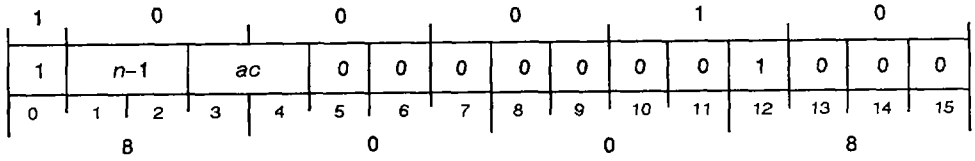
Example

```
    LDA   0,TWELVE      ;Load AC0 with 12.
    ADDI  5,0           ;AC0[16-31] now has 17. Bits 0-15 undefined.
    . . .
TWELVE:  .WORD 12.
```

Add Immediate ADI

ECLIPSE Instruction

ADI *n,ac*



Function: *n + ac -> ac*

Parameters: None

ADI adds an integer in the range 1–4 to the unsigned 16-bit integer contained in the specified accumulator.

Arguments

n	Integer in range 1–4. Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field. Thus, programmer should code exact value that is to be added.
ac(16–31)	Before execution, contains unsigned 16-bit integer. After execution, contains result.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

ADDI, NADDI, WADDI, WNADI	Add a signed 16- or 32-bit immediate value to an accumulator.
NADI, WADI	Add an immediate value to an accumulator.

Exceptions

None

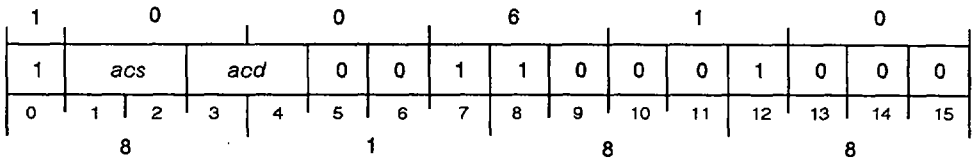
Example

```
ADI 4,2    ;Assume that AC2 contains 177775₈. After the instruction
           ;is executed, AC2 contains 000001₈ ;and CARRY is unchanged.
```

AND with Complemented Source ANC

ECLIPSE Instruction

ANC *acs,acd*



Function: $\overline{acs}$ AND *acd* -> *acd*

Parameters: None

ANC forms the logical AND of the logical complement of the contents of *acs* and the contents of *acd* and places the result in *acd*. The instruction sets a bit position in the result to 1 if the corresponding bits in *acs* contain 0 and *acd* contain 1.

Arguments

acs(16–31)	Before execution, contains 16-bit value.
	After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd(16–31)	Before execution, contains 16-bit value.
	After execution, contains result of operation.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

AND	AND
WANC	Wide AND with Complemented Source
WAND	Wide AND

Exceptions

None

Example

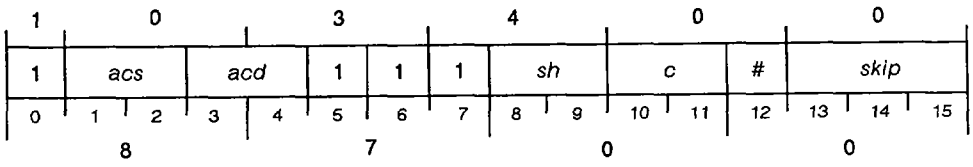
```
ANC 2,0    ;ANDs the logical complement of AC2 with the
....       ;contents of AC0 and returns the result to AC0.
```

AND

AND

ECLIPSE Instruction

AND[*c*][*sh*][#] *acs*,*acd*[,*skip*]
(*skip* false return)
(*skip* true return)



Function: *acs* AND *acd* -> *acd*

Parameters: None

AND initializes CARRY to the specified value and places the logical AND of *acs* and *acd* in the shifter. Each bit placed in the shifter is 1 only if the corresponding bit in both *acs* and *acd* is 1; otherwise the resulting bit is 0. The instruction then performs the specified shift operation and places the result in *acd* if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

Arguments

[*c*] Processor determines effect of CARRY flag (*c*) on initial value of CARRY before performing operation (opcode). Following table gives values of *c* and bits 10 and 11 and specifies operation.

Symbol [*c*]	Bits 10-11	Operation
omitted	0 0	Leave CARRY unchanged
Z	0 1	Initialize CARRY to 0
O	1 0	Initialize CARRY to 1
C	1 1	Complement CARRY

[*sh*] Processor shifts CARRY flag and 16 data bits after performing operation. Processor can shift bits left or right one bit position or can swap the two bytes. Following table gives values of *sh* and bits 8 and 9 and specifies shift operation.

Symbol [*sh*]	Bits 8-9	Shift Operation
omitted	0 0	Do not shift the result
L	0 1	Shift left
R	1 0	Shift right
S	1 1	Swap the two 8-bit bytes

[#] Except with no-load option (#), processor loads result of shift operation into destination accumulator. No-load option can test result of operation without destroying destination accumulator contents. Following table gives values of no-load option and bit 12 and specifies operation.

Symbol [#]	Bit 12	Operation
omitted	0	Load result into *acd*
#	1	Do not load result; restore initial CARRY flag

acs(16-31) Before execution, contains 16-bit value.

After execution, contents unchanged unless *acs* and *acd* are same accumulator.

acd(16-31) Before execution, contains 16-bit value.

After execution, contains result if no-load bit (#) is 0.

[*skip*] Processor skips next instruction if condition test true. Following table gives test conditions as specified by bits 13 to 15 and specifies operation.

Symbol [*skip*]	Bits 13-15	Operation
omitted	0 0 0	Never skip
SKP	0 0 1	Always skip
SZC	0 1 0	Skip if CARRY is 0
SNC	0 1 1	Skip if CARRY is not 0
SZR	1 0 0	Skip if result is 0
SNR	1 0 1	Skip if result is not 0
SEZ	1 1 0	Skip if either CARRY or result is 0
SBN	1 1 1	Skip if both CARRY and result are not 0

Skip omits next sequential 16-bit word. Make sure that *skip* does not transfer control to point within 32-bit or longer instruction.

Registers, Flags, and Stacks

AC0-AC3 Can be individually specified as *acs* and *acd*; otherwise unused.

CARRY Logical operation leaves initial CARRY unchanged unless [*c*] option specified. If left or right shift occurs, final resulting CARRY is bit shifted into CARRY.

Overflow 0

PC PC + 1 (false exit)
PC + 2 (true exit)

PSR Unchanged

Stacks Unchanged

Related Instructions

ANC, WANC AND with complemented source

WAND Wide AND

Exceptions

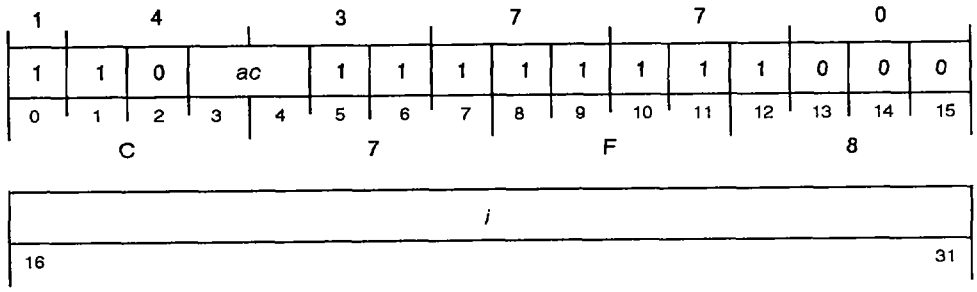
Do not specify **AND** with no-load option (#) in combination with either never skip or always skip option. Instruction may not end in 1000_2 or 1001_2 (reserved for other instructions).

Example

```
ANDOS 2,1 ;Initializes CARRY to one, ANDs the contents
...       ;of AC2 and AC1, swaps the two lower bytes of
...       ;the result, and places this result in AC1.
```

AND Immediate ANDI

ANDI *i,ac*



Function: *i* AND *ac* -> *ac*

Parameters: None

ANDI logically ANDs the contents of the immediate field and the contents of the specified accumulator, placing the result in the specified accumulator.

Arguments

i	16-bit value
ac(16-31)	Before execution, contains 16-bit value.
	After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

WANDI	Wide AND Immediate

Exceptions

None

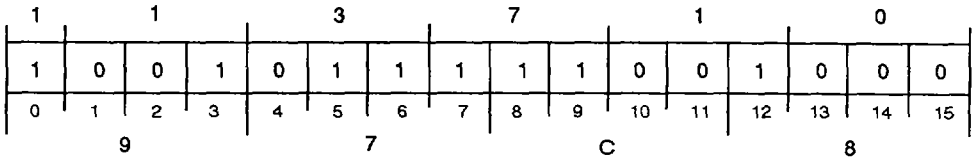
Example

```
ANDI 123,0          ;Logically ANDs 123₈ with the contents of
....                ;AC0, placing the result into AC0.
```

Block Add and Move BAM

ECLIPSE Instruction

BAM



Function: source @(AC2) + AC0 -> destination @(AC3)

Parameters:
AC0 = #(addend) -> unchanged
AC1 = #(number of words) -> 0
AC2 = source E -> last E + 1
AC3 = destination E -> last E + 1

BAM moves words in consecutive, ascending order from one memory location to another, adding a constant to each one. After fetching a word from the source location, the instruction adds the contents of AC0 and stores the result in the destination location. The source and destination fields may overlap in any way.

The resolved effective addresses are confined to the first 64 Kbytes of the current segment.

Arguments

None

Registers, Flags, and Stacks

AC0(16–31)	Before execution, contains unsigned 16-bit addend.
	After execution, content remains unchanged.
AC1(16–31)	Before execution, contains unsigned 16-bit integer specifying number of words to be moved. With each word moved, number decrements by 1.
	After execution, content is 0.
AC2(16–31)	Before execution, contains source location word address. With each word moved, value increments by 1.
	After execution, points to last location in string plus 1.
AC3(16–31)	Before execution, contains destination location word address. With each word moved, value increments by 1.
	After execution, points to last location in string plus 1.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

BLM	Block Move
LEF, ELEF	Use these instructions to place addresses into AC2 and AC3.
Load immediate	Use these instructions to place values into AC0 and AC1.
WBLM	Wide Block Move

Exceptions

Because **BAM** may have a long execution time, it is interruptible. If the instruction is interrupted, the program counter is decremented by one before being saved, thus it points to the interrupted instruction. Because addresses and the word count are updated after each word is stored, any interrupt service routine returning control via the saved program counter will correctly restart the instruction.

The initial value of AC1 must be greater than 0 and less than or equal to 32,768. If the value is outside these bounds, no data is moved and the accumulators remain unchanged.

If the specified addresses in AC2 and AC3 are outside the user's address space, a protection fault occurs with error code 2 returned in AC1, even if no words are to be moved. AC2 and AC3 will contain the last valid indirect address.

When updating the source and destination addresses, the instruction forces bit 16 of the result to 0. This ensures that upon a return, the instruction will not try to resolve an indirect address in either AC2 or AC3. Note that the next address after 77777_8 is 0.

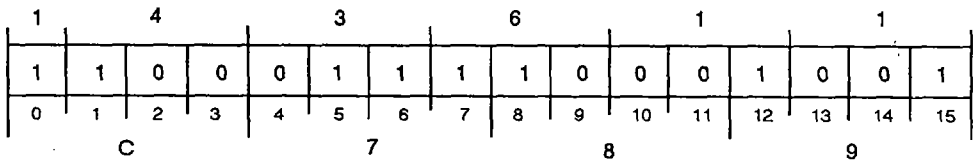
If the result of any add operation is greater than $32{,}768_8$, no indication is given.

Example

```
ELEF   0,3 ;Addend = 3.
LDA    1,SIZE       ;Load size of array.
ELEF   2,ARRAY      ;Load source address.
MOV    2,3 ;Destination address equals source address.
BAM        ;Add 3 to every element in array.
.
.
.
SIZE:      .WORD 20
ARRAY:     ...
```

Breakpoint BKPT

BKPT



Function: 6 double words -> wide stack (wide return block)
0 -> PSR
page zero[locations 10-11] -> PC

Parameters: None

NOTE: PC on stack = (**BKPT**)

BKPT pushes a wide-return block onto the stack and transfers program control to the breakpoint handler. Before executing **BKPT**, first store 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.

Arguments

None

Registers, Flags, and Stacks

AC0-AC3	Unused
CARRY	Indeterminate
Overflow	Indeterminate
PC	After pushing return block, contains effective address of wide jump indirect through breakpoint handler in current segment (locations 10-11 in page zero). If no stack overflow, control transfers to breakpoint handler.
PSR	Set to 0 after pushing return block.
Stack	Wide stack contains wide-return block (value of PC in return block is address of **BKPT**.)

Related Instructions

PBX	Pop Block and Execute returns from the breakpoint handler if you do not remove **BKPT**.
WPOPB	Wide Pop Block Block returns from the breakpoint handler if you do remove **BKPT**.

Exceptions

If a stack overflow fault occurs, the processor services the stack fault and AC1 contains the code 0.

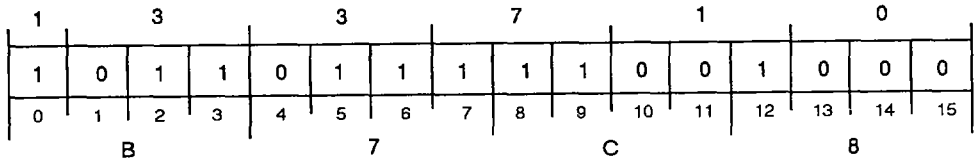
Example

```
BKPT      ;Transfers program control to breakpoint handler.
```

Block Move BLM

ECLIPSE Instruction

BLM



Function: source @(AC2) -> destination @(AC3)

Parameters: AC1 = # words # -> 0
AC2 = source E -> last E + 1
AC3 = destination E -> last E + 1

BLM moves a specified number of memory words in consecutive ascending order from a source location to a destination location. The source and destination fields may overlap in any way.

The resolved effective addresses are confined to the first 64 Kbytes of the current segment.

Arguments

None

Registers, Flags, and Stacks

AC0	Unused
AC1(16–31)	Before execution, contains unsigned 16–bit integer specifying number of words to be moved. With each word moved, the number decrements by 1. After execution, contains 0.
AC2(16–31)	Before execution, contains source location word address. With each word moved, the value increments by 1. After execution, points to last word in string plus 1.
AC3(16–31)	Before execution, contains destination location word address. With each word moved, the value increments by 1. After execution, points to last word in string plus 1.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

BAM	Block Add and Move
LEF, ELEF	Use these instructions to place addresses into AC2 and AC3.
Load immediate	Use these instructions to place values into AC1.
WBLM	Wide Block Move

Exceptions

Because **BLM** may have a long execution time, it is interruptible. If the instruction is interrupted, the program counter is decremented by one before being saved, thus it points to the interrupted instruction. Because addresses and the word count are updated after each word is stored, any interrupt service routine returning control via the saved program counter will correctly restart the instruction.

When updating the source and destination addresses, the instruction forces bit 16 of the result to 0. This ensures that upon a return, the instruction will not try to resolve an indirect address in either AC2 or AC3. Note that the next address after 77777_8 is 0.

If the addresses in AC2 and AC3 are outside the user's address space, a protection fault occurs with error code 2 returned in AC1, even if no words are to be moved.

The number in AC1 must be greater than 0 and less than or equal to $32{,}768_8$; if number is outside these bounds, no data is moved and the contents of the accumulators remain unchanged.

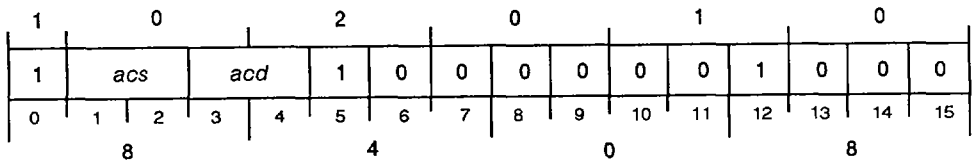
Example

```
LDA   1,SIZE       ;Number of words to move.
ELEF  2,AARRAY     ;Source address.
ELEF  3,BARRAY     ;Destination address.
BLM                ;Copy AARRAY to BARRAY.
.
.
.
SIZE:  .WORD 20
AARRAY:    ....
BARRAY:    ....
```

Set Bit to One — BTO

ECLIPSE Instruction

BTO *acs,acd*



Function: 1 -> @(*acs* & *acd*)bit

Parameters: *acs* = base word pointer -> unchanged
acd = word offset & bit pointer -> unchanged

NOTE: If *acs* is *acd*, base word identifier = 0 (of the current segment)

BTO sets the specified bit in the specified memory location to 1. **BTO** is an indivisible instruction. The effective address generated by this instruction is confined to the first 64 Kbytes of the current segment.

Arguments

acs(16–31) Provides high-order 16 bits of 32-bit bit pointer. If same accumulator is specified as for *acd*, instruction treats accumulator content as low-order 16 bits of bit pointer and assumes high-order 16 bits are 0.
After execution, contents unchanged.

acd(16–31) Provides low-order 16 bits of 32-bit bit pointer.
After execution, contents unchanged.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as either *acs* or *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stacks	Unchanged

Related Instructions

BTZ, WBTZ	Set bit to zero
WBTO	Wide Set Bit to One

Exceptions

None

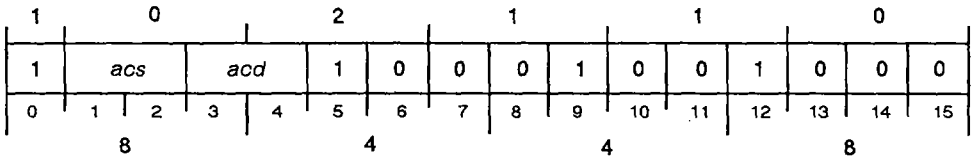
Example

```
ELEF   0,FLAGS      ;Get address of word containing flags.
SUB    1,1          ;Get a zero in AC1.
ADDI   5,1          ;Get a 5 in AC1.
BTO    0,1          ;Set bit 5 of flags word to 1.
FLAGS: .WORD 0      ;Flags word (initially all zero).
```

Set Bit to Zero BTZ

ECLIPSE Instruction

BTZ *acs,acd*



Function: 0 -> @(*acs* & *acd*)bit

Parameters: *acs* = base word pointer -> unchanged
acd = word offset & bit pointer -> unchanged

NOTE: If *acs* is *acd*, base word identifier = 0 (of the current segment)

BTZ sets the specified bit of the specified memory location to 0. **BTZ** is an indivisible instruction. The effective address generated by this instruction is confined to the first 64 Kbytes of the current segment.

Arguments

acs(16–31) Provides high-order bits of 32-bit bit pointer. If same accumulator is specified as for *acd*, instruction treats accumulator content as low-order 16-bits of the bit pointer and assumes the high-order 16 bits are 0.

After execution, contents unchanged.

acd(16–31) Provides low-order bits of 32-bit bit pointer.

After execution, contents unchanged.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as either *acs* or *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

BTO, WBTO Set bit to one

WBTZ Wide Set Bit to Zero

Exceptions

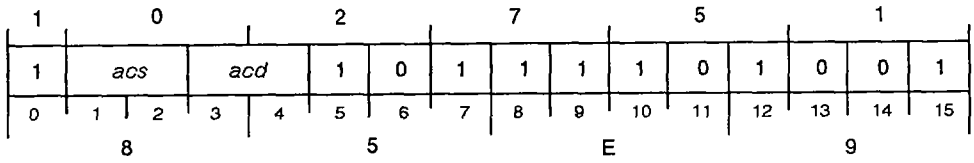
None

Example

```
    ELEF   0,FLAGS      ;Get address of word containing flags.
    SUB    1,1          ;Get a zero in AC1.
    ADDI   6,1          ;Get a 6 in AC1.
    BTZ    0,1          ;Set bit 6 of flags word to 0.
FLAGS:   .WORD -1       ;Flags word (initially all ones).
```

Command I/O CIO

CIO *acs,acd*



Function: R/W --> I/O system bus

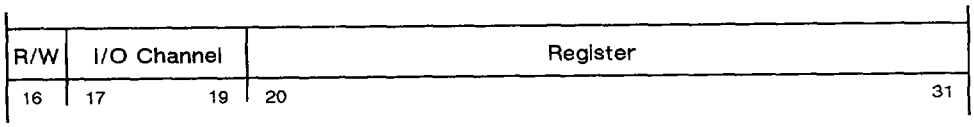
Parameters:
acs(16-31) = #(command) --> unchanged
If *acs*(16): 1 = write; *acd*(16-31) = data
0 = read; result --> *acd*(16-31)
acs(17-19) = I/O channel number
acs(20-31) = Register address
acd(16-31) = data

CIO asserts a read or write data command from an accumulator onto the I/O channel bus. Used to read DCH/BMC map status registers, to write to DCH/BMC map definition and mask registers, and to load DCH/BMC map slots from the accumulators.

The channel sets its BUSY flag to 1 when a write or read operation is in progress.

Arguments

acs(16-31) Before execution, contains the I/O command to be transmitted. Command is formatted as follows:



Bits	Name	Contents or Function
16	Read/Write	0 specifies a read operation1 specifies a write operation
17-19	I/O Channel	Specifies I/O channel. Channel numbers range from 0 to 7. Default is 0. Refer to Device Management chapter for definition of I/O channels on a specific system.
20-31	Register	Specifies address of a DCH/BMC Address register in range from 0000 to 7777_8 . Refer to Device Management chapter for definition of DCH/BMC registers.

After execution, contents unchanged.

acd(16-31) Temporary data storage. On a read command, receives data from specified address; on a write command, sends data to specified address.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as either *acs* or *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

CIOI	Command I/O Immediate

Exceptions

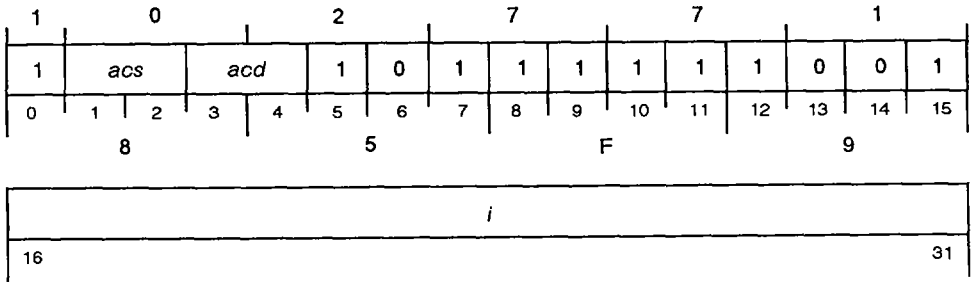
None

Example

```
NLDAI  126000,0     ;Specify write to register 6000₈ of IOC 2.
NLDAI  2,1          ;Data to enable DCH mapping.
CIO    0,1          ;Write to register 6000₈ of IOC 2.
```

Command I/O Immediate CIOI

CIOI *i,acs,acd*



Function: R/W -> I/O system bus

Parameters: If *acs* = *acd*
Then, *i* = command
Else, *i* OR *acs*(16-31) = command

acs = command --> unchanged
i = command --> unchanged

If command(16): 1 = write; *acd*(16-31) = data
0 = read; result --> *acd*(16-31)

CIOI asserts a read or write data command onto the I/O channel bus. Used to read DCH/BMC map status registers, to write to DCH/BMC map definition and mask registers, and to load DCH/BMC map slots from the accumulators.

The channel sets its BUSY flag to 1 when a write or read operation is in progress.

Arguments

acs(16-31) Before execution,

If same accumulator specified as for *acd*, *i* contains command.

If specification is different from *acd*, contents logically ORed with *i* to form command to be transmitted.

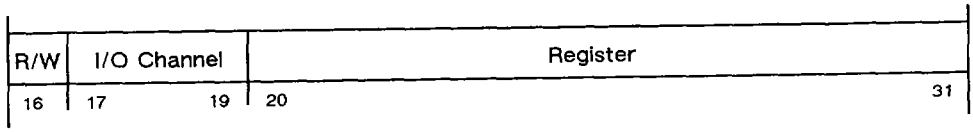
After execution, contents unchanged.

acd(16-31) Temporary data storage.

On a read command, receives data from specified address.

On a write command, contains data to be transmitted.

i Immediate definition of the command or is logically ORed with *acs* to derive the command. Command is formatted as follows:



Bits	Name	Contents or Function
16	Read/Write	0 specifies a read operation1 specifies a write operation
17-19	I/O Channel	Specifies I/O channel. Channel numbers range from 0 to 7. Default is 0. For definition of I/O channels, refer to Device Management chapter.
20-31	Register Address	Specifies address of a DCH/BMC Map register or slot in range from 0000 to 7777_8 . For definition of DCH/BMC Map registers and slots, refer to Device Management chapter.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as either *acs* or *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

CIO	Command I/O

Exceptions

None

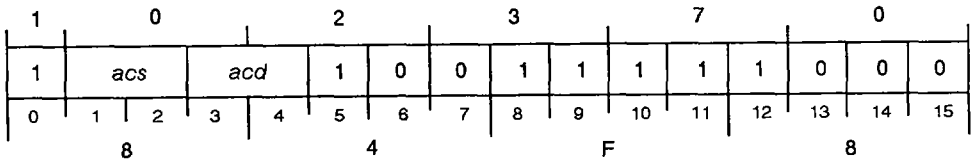
Example

```
NLDAI  106000,0     ;Specify write to register 6000₈ .
NLDAI  2,1          ;Data to enable DCH mapping.
CIOI   10000,0,1    ;Enable DCH mapping on IOC 1.
CIOI   20000,0,1    ;Enable DCH mapping on IOC 2.
CIOI   30000,0,1    ;Enable DCH mapping on IOC 3.
```

Compare to Limits CLM

ECLIPSE Instruction

CLM *acs,acd*
(*acs* ≠ *acd*; *acs* not within limits return)
(*acs* ≠ *acd*; *acs* within limits return)
(*acs* = *acd*; *acs* not within limits return)
(*acs* = *acd*; *acs* within limits return)



Function: L < = *acs* < = H then skip

Parameters: *acs* = 2# -> unchanged

If *acs* is not *acd*: (*acd*) = L
(*acd* + 1) = H

If *acs* is *acd*: (CLM + 1) = L
(CLM + 2) = H

CLM compares a signed 16-bit integer stored in *acs* with two other signed 16-bit integers (lower limit L and higher limit H) and skips the next sequential instruction if the integer is equal to or between L and H. If the integer in *acs* is less than L or greater than H, the next sequential instruction is executed. The specification of *acd* determines the location of L and H.

When the location of the instruction is sequential with L and H, the instruction can be placed anywhere in the address space.

The effective addresses generated are confined to the first 64 Kbytes of the current segment.

Arguments

acs(16-31) Contains signed 16-bit integer to be compared.

acd(16-31) If specification is different from *acs*, bits 17-31 contain address of the lower limit value L; the higher limit value H is contained in the next location following L.

If specified the same as *acs*, then limits L and H are in the next two respective locations following this instruction.

The values of L and H are interpreted as signed 16-bit integers.

Registers, Flags, and Stacks

AC0-AC3 Can be individually specified as either *acs* or *acd*; otherwise unused.

CARRY Unchanged

Overflow 0

PC	PC + 1 (*acs* ≠ *acd*; *acs* not within limits) PC + 2 (*acs* ≠ *acd*; *acs* within limits) PC + 3 (*acs* = *acd*; *acs* not within limits) PC + 4 (*acs* = *acd*; *acs* within limits)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WCLM	Wide Compare to Limits

Exceptions

None

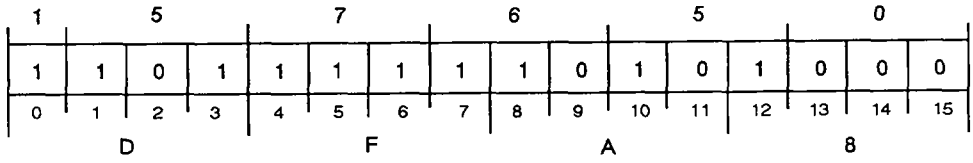
Example

```
    CLM    0,0          ;Is the value of AC0 greater than or equal
    .WORD  5.           ;to 5 and less than or equal to 10?
    .WORD  10.          ;
    JMP        OUT_LMT  ;Value of AC0 is not within the limits.
IN_LMT:    . . .        ;Value of AC0 is between 5 and 10, inclusive.
```

Character Compare CMP

ECLIPSE Instruction

CMP



Function: string 1 ?=? string 2
Result returned -> AC1

Parameters: AC0 = string 2 #bytes[+ = ascending,- = descending] ->
0 or uncompared bytes
AC1 = string 1 #bytes[+ = ascending,- = descending] ->
Result -1 (string 1 < string 2)
0 (string 1 = string 2)
+1 (string 1 > string 2)
AC2 = str2 byte pointer -> last byte pointer $\neq$1 or failing byte
AC3 = str1 bp -> last bp $\neq$ 1 or failing byte

NOTE: When shorter string exhausted, comparison continues using spaces.

CMP compares two strings of bytes, a byte at a time from each string. **CMP** terminates if two bytes are not equal, or the specified number of bytes have been compared, and returns a code reflecting the result. Each byte is treated as an unsigned 8-bit integer in the range $0\text{–}255_{10}$. At completion of the instruction, both strings remain unchanged.

The strings may overlap in any way; the overlap does not affect execution of the instruction. If two strings are unequal in length, upon completion of shorter string, comparisons continue using space characters $<040_8>$ for comparison with remaining bytes of longer string.

The effective addresses generated are confined within the first 64 Kbytes of the current segment.

Arguments

None

Registers, Flags, and Stacks

AC0(16–31) Before execution, contains length and direction of comparison for string 2.

If string is compared from its lowest memory location to its highest, contains unsigned value of number of bytes in string 2.

If string is compared from its highest memory location to its lowest, contains negative value of number of bytes in string 2.

After execution, contains signed or unsigned number of bytes left to compare in string 2.

AC1(16–31) Before execution, contains length and direction of comparison for string 1.

If string is compared from its lowest memory location to its highest, contains unsigned value of number of bytes in string 1.

If string is compared from its highest memory location to its lowest, contains negative value of number of bytes in string 1.

After execution, contains a return code as follows:

Code	Meaning
-1	string 1 < string 2
0	string 1 = string 2
+1	string 1 > string 2
2	invalid pointer (protection fault error)

AC2(16-31) Before execution, contains byte address for first byte to be compared in string 2. When string is to be compared in ascending order, points to lowest byte. When string is compared in descending order, points to highest byte.

With each successful comparison of a byte in string, address is incremented or decremented, depending on direction of compare.

After execution, contains byte address either of failing byte, if a mismatch is found, or of next byte following last byte in string, if both strings compare.

AC3(16-31) Before execution, contains byte address for first byte to be compared in string 1. When string is to be compared in ascending order, points to lowest byte. When string is compared in descending order, points to highest byte.

With each successful comparison of a byte in string, address is incremented or decremented, depending on direction of comparison.

After execution, contains byte address either of failing byte, if a mismatch is found, or to next byte following last byte in string, if both strings compare.

CARRY Unchanged

Overflow 0

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

WCMP Wide Character Compare

Exceptions

If both of the strings are defined with a zero length, no comparisons are made and the result returned to AC1 is 0.

If the specified addresses are outside the user's address space, a protection fault occurs (code 2 returned in AC1), even if no bytes are to be compared.

Because **CMP** may have a long execution time, it is interruptible. If the instruction is interrupted, the program counter is decremented by one before being saved; thus it points to the interrupted instruction. Because addresses are updated after each byte is compared, any interrupt service routine returning control via the saved program counter will correctly restart the instruction.

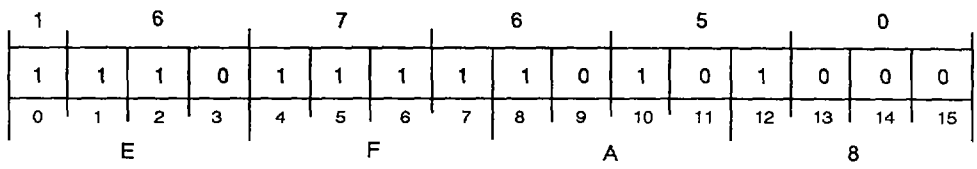
Example

```
    ;This program Compares "SUPERCALIFRAGILISTICEXPIALIDOCIOUS" with
    ;another string in memory. It places a 1 in the location "FOUND"
    ;if the string is equal, otherwise it places a zero in FOUND.
    ;
    ;       AC0: String 2 Length and Direction = 34.
    ;       AC1: String 1 Length and Direction
    ;       AC2: String 2 Byte Pointer -> SUPER...
    ;       AC3: String 1 Byte Pointer
START:          LEF     1,34.,0         ;Put 34 (Length of"SUPER...") into AC1
                MOV     1,0             ;Put 34 into AC0. Search only 34 chars
                                        ;of the string in memory.
                ELEF    3,2*SUPER,0     ;Put Byte Pointer to "SUPER..." into
                                        ;AC3.
                ELEF    2,2*OTHER,0     ;Put byte pointer to OTHER string in
                                        ;AC2.
                CMP                     ;Compare them.
                MOV#    1,1,SNR         ;If AC1 <> 0 then strings are not
                                        ;equal.
                JMP     EQ
NEQ:            SUB     0,0             ;RESULT not equal.
                STA     0,FOUND         ;Put a 0 in FOUND,
                JMP     EXIT            ;and EXIT.
EQ:             SUBZL   0,0             ;Create a 1.
                STA     0,FOUND         ;Set FOUND to 1.
EXIT:           SUB     2,2             ;End of Program - RETURN TO CALLER.
                ?RETURN
                                        ;
    ;           VARIABLES
    ;
SUPER:          .TXT "SUPERCALIFRAGILISTICEXPIALIDOCIOUS"
FOUND:          .0
OTHER:          .BLK    100.            ;The other string is placed here.
                .END START
```

Character Move Until True CMT

ECLIPSE Instruction

CMT



Function: source @(AC3) -> destination @(AC2)
If byte = delimiter, instruction terminates; byte not moved

Parameters: AC0 = delimiter table address -> E(delimiter table)
AC1 = # bytes [+ = ascending, - = descending] -> 0 or # unmoved bytes
AC2 = destination bp -> last byte pointer $\neq$ 1
AC3 = source byte pointer -> last byte pointer $\neq$ 1

NOTE: If AC2 = AC3, then no bytes written, but string is scanned for delimiter.

CMT moves a string of bytes, one at a time, from one area of memory to another until either a table-specified delimiter character is encountered or the specified number of bytes has been transferred.

Before each byte is moved, its value (an unsigned 8-bit integer in the range $0\text{-}255_{10}$) is used as a bit index into a 256-bit delimiter table.

If the indexed bit in the delimiter table is 0, the byte is not a delimiter and it is copied from the source string into the destination string.

If the indexed bit in the delimiter table is 1, the byte is a delimiter; the byte does not get copied and the instruction terminates with AC3 containing the address of the delimiter.

Both strings are processed in the same direction, either from the lowest specified memory locations upward (ascending order), or from the highest specified memory locations downward (descending order). The source and destination strings may overlap in any way; however, since one byte is moved at a time, certain types of overlap may produce undesired results.

The effective addresses generated are confined to the first 64 Kbytes of the current segment.

Arguments

None

Registers, Flags, and Stacks

AC0(16-31) Before execution, contains word address of start of 256-bit (16-word) delimiter table.

After execution, contains resolved address of delimiter table.

AC1(16–31)	Before execution, contains total number of bytes in source string to be moved and direction in memory in which strings are to be processed. If processing is to occur in ascending order, contains unsigned value of total number of bytes to be moved. If processing is to occur in descending order, contains negative number of bytes in source string. After execution: if no delimiter found, contains 0. if delimiter found, contains number of bytes (or two's complement of number of bytes) not moved.
AC2(16–31)	Before execution, contains byte address for first byte in destination string. When string is accessed in ascending order, points to lowest byte. When string is accessed in descending order, points to highest byte. With each byte stored, address is incremented or decremented, depending on direction of process. After written execution, contains address of next byte following last byte in string. (If AC2 equals AC3 before execution, they are also equal after execution, even though no writes are performed.)
AC3(16–31)	Before execution, contains byte address for first byte in source string. When source string is accessed in ascending order, points to lowest byte. When string is accessed in descending order, points to highest byte. With each byte stored, address is incremented or decremented, depending on direction of process. After execution, contains address of delimiter or, if none found, next byte following last byte in string.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load byte address	Use these instructions to place addresses into AC2 and AC3.
Load effective address	Use these instruction to place a word address into AC0.
Load with immediate	Use these instructions to place a value into AC1.
WCMT	Wide Character Move Until True

Exceptions

Because **CMT** may have a long execution time, it is interruptible. If the instruction is interrupted, the program counter is decremented by one before being saved, thus it points to the interrupted instruction. Because addresses and byte count are updated after each move, any interrupt service routine returning control via the saved program counter will correctly restart the instruction.

The original contents of AC0, AC2, and AC3 must be valid pointers to some area in the user's address space. If the addresses are invalid, a protection fault may occur (even if no bytes are to be moved), and fault code 2 gets stored in AC1.

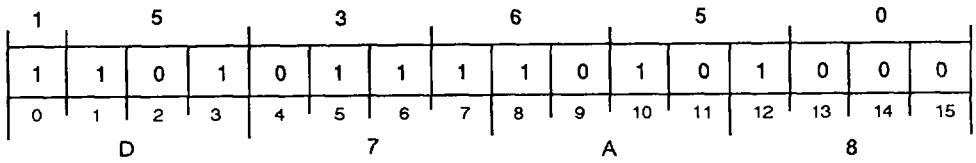
Example

```
    .ENT  START, DESSTR, SRCSTR, DELIMS
    ;This program moves a string of up to 80 characters, or until
    ;it encounters one of the following termination characters:
    ;       NUL <000>     NULL or ZERO character
    ;       CR  <010>     CARRIAGE RETURN
    ;       NL  <012>     NEW LINE
    ;       FF  <014>     FORM FEED
    ;       AC0: WORD address of bit addressable delimiter table
    ;       AC1: Source length and Direction
    ;       AC2: Destination string Byte Pointer
    ;       AC3: Source string Byte Pointer
START:    LEF 1,80.,0  ;Put 80 (Max length) into AC1.
    LEF   0,DELIMS     ;Point AC0 at DELIMITER TABLE.
    ELEF 2,2*DESSTR,0 ;Put dest. buffer BYTE POINTER into AC2.
    ELEF 3,2*SRCSTR,0  ;Point to first CHAR of SRCSTR and
    CMT                ;move it until we hit delimiter or 80th char.
    SUB   2,2          ;End of Program - RETURN TO CALLER
    ?RETURN
    ;       -- DELIMITER TABLE --
            .ENABLE UWORD
            .RDX         2
    ;       NUL CR    NL FF
    ;       /      \  /  /
DELIMS:     1000000010101000          ;0 - 17 octal
            0000000000000000          ;20 - 37
    ...
            1000000010101000          ;200-217  Note: 200, 210, 212, 214 are
            0000000000000000          ;220-237 NUL,CR,NL,FF with HI bit set
            0000000000000...          ;240 -...
            0000000000000000          ;360 - 377
            .RDX         8
    ;
    ;       VARIABLES
    ;
SRCSTR:     .BLK 100                  ;Who knows how long source will be.
DESSTR:     .BLK         40.          ;Can't be longer than 80 chars!
    ;IF SRCSTR = "ABCDE<12>FGHIJK"
    0000000000000000   ;260 - 277
            0000000000000000          ;300 - 317
            0000000000000000          ;320 - 337
            0000000000000000          ;340 - 357;\
    ;New Line
    ;Then only ABCDE will be moved to DESSTR. The DELIMITER CHARACTER
    ;itself is NOT moved.
```

Character Move CMV

ECLIPSE Instruction

CMV



Function: source @(AC3) -> destination @(AC2)
relative length -> CRY

Parameters: AC0 = destination #bytes [+ = ascending,- = descending] -> 0
AC1 = source #bytes[+ = ascending,- = descending] -> 0 or unmoved bytes
AC2 = destination byte pointer -> last byte pointer +/-1
AC3 = source byte pointer -> last byte pointer +/-1
CRY = x -> relative length:
1 (source > destination)
0 (source = <destination)

NOTE: If source < destination, remainder of destination is filled with spaces.

CMV moves a string of bytes, one at a time, from one area of memory to another and returns a value in CARRY reflecting the relative lengths of the source and destination strings. The strings may overlap in any way; the overlap does not affect execution of the instruction.

The effective addresses generated are confined to the first 64 Kbytes of the current segment.

Arguments

None

Registers, Flags, and Stacks

AC0(16-31) Before execution, contains length and direction of accessing for destination string.

If string is accessed from its lowest memory location to highest, contains positive value of number of bytes in string.

If string is accessed from its highest memory location to its lowest, contains negative value of number of bytes in string.

After execution, contains zeros.

AC1(16-31) Before execution, contains length and direction of accessing for source string.

If string is to be accessed from its lowest memory location to its highest, contains positive value of number of bytes in string.

If string is to be accessed from its highest memory location to its lowest, contains negative value of number of bytes in string.

After execution, reflects number, or two's complement, of bytes left unmoved in source string.

AC2(16-31)	Before execution, contains memory address for first byte to be accessed in destination string. When string is to be accessed in ascending order, points to lowest byte. When string is accessed in descending order, points to highest byte. With each byte moved, address is incremented or decremented, depending on direction of accessing. After execution, contains address for next byte following string.
AC3(16-31)	Before execution, contains memory address for first byte to be accessed in source string. When string is to be accessed in ascending order, points to lowest byte. When string is accessed in descending order, points to highest byte. With each byte moved, address is incremented or decremented, depending on direction of accessing. After execution, contains address for next byte following last byte fetched.
CARRY	If 0, source number bytes is =< destination number bytes. If 1, source number bytes is > destination number bytes.
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load byte address	Use these instructions to place addresses into AC2 and AC3.
Load with immediate	Use these instructions to place values into AC0 and AC1.
WCMV	Wide Character Move

Exceptions

Because **CMV** may have a long execution time, it is interruptible. If the instruction is interrupted, the program counter is decremented by one before being saved, thus it points to the interrupted instruction. Because addresses and byte counts are updated after each move, any interrupt service routine returning control via the saved program counter will correctly restart the instruction.

If both strings are defined with a zero length, no moves are made and the result returned in CARRY is 0. If the destination string is longer than the source string, upon completion of the source string, the remaining locations of the destination string are filled with space characters.

If the initial value of AC0 is 0, no bytes are fetched and none are stored. If the initial value of AC1 is 0, no bytes are fetched and the destination field is filled with spaces.

If the contents of AC2 and AC3 are not valid pointers to some area in the user's address space, a protection fault may occur (even if no bytes are to be moved), and fault code 2 gets stored in AC1.

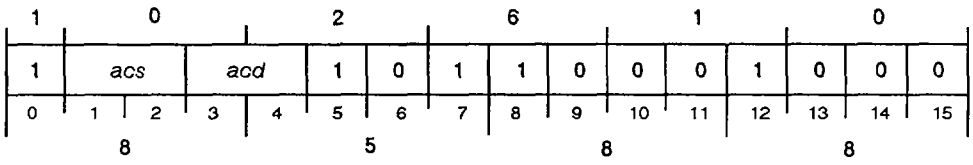
Example

```
    ;This program Reverses the string "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    ;
    ;       AC0: Destination Length and Direction
    ;       AC1: Source length and Direction
    ;       AC2: Destination Byte Pointer
    ;       AC3: Source Byte Pointer
START:          LEF     0,26.,0         ;Put 26 (Length of Dest) into AC0.
                NEG     0,1             ;Put -26 (length of Source) into AC1.
                                        ;Negative length means backward move!
                ELEF    2,2*DESSTR,0    ;Put destination buffer BYTE POINTER
                                        ;into AC2.
                ELEF    3,2*SRCSTR      ;
                         +25.,0         ;Point to the last CHAR of SRCSTR
                                        ;("Z") and
                CMV                     ;move it.
                SUB     2,2             ;End of Program - RETURN TO CALLER.
                ?RETURN
    ;
    ;       VARIABLES
    ;
SRCSTR:     .TXT "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
DESSTR:     .BLK        14
            .END START
```

Count Bits COB

ECLIPSE Instruction

COB *acs,acd*



Function: *acs*(# of 1s) + *acd* -> *acd*

Parameters: None

COB counts the number of ones in a source accumulator and adds the number to a destination accumulator.

Arguments

acs(16–31) Before execution, contains source word for the bit count.

After execution, contents unchanged, unless *acs* is specified same as *acd*, then the bit count is added to the source word.

acd(16–31) Destination for bit count. If initialized with a value, must be expressed as a signed 16-bit integer.

Registers, Flags, and Stacks

AC0–AC3 Can be individually specified as either *acs* or *acd*; otherwise unused.

CARRY Unchanged

Overflow 0

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

WCOB Wide Count Bits

Exceptions

None

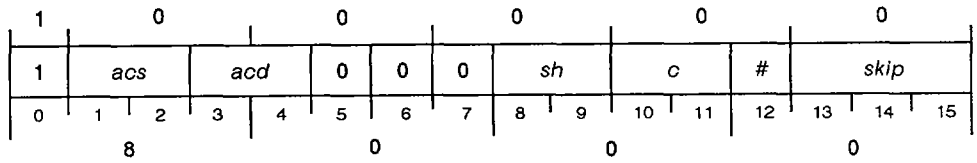
Example

```
SUB   0,0          ;Start with 0 in AC0.
ADC   1,1          ;Set AC1 to all ones.
COB   1,0          ;Adds 16 to AC0. New value is 16.
```

Complement COM

ECLIPSE Instruction

COM[*c*][*sh*][#] *acs*,*acd*[,*skip*]
(*skip* false return)
(*skip* true return)



Function: $\overline{acs}$ -> *acd*

Parameters: None

COM initializes CARRY to the specified value, forms the logical complement of the value in *acs*, and performs the specified shift operation. The instruction then places the result in *acd* if the no-load bit is 0. If the skip condition is true, the next sequential word is skipped.

Arguments

[*c*] Processor determines effect of CARRY flag (*c*) on initial value of CARRY before performing operation (opcode). Following table gives values of *c* and bits 10 and 11 and specifies operation.

Symbol [*c*]	Bits 10-11	Operation
omitted	0,0	Leave CARRY unchanged
Z	0,1	Initialize CARRY to 0
O	1,0	Initialize CARRY to 1
C	1,1	Complement CARRY

[*sh*] Processor shifts CARRY flag and 16 data bits after performing operation. Processor can shift bits left or right one bit position or can swap the two bytes. Following table gives values of *sh* and bits 8 and 9 and specifies shift operation.

Symbol [*sh*]	Bits 8-9	Shift Operation
omitted	0,0	Do not shift the result
L	0,1	Shift left
R	1,0	Shift right
S	1,1	Swap the two 8-bit bytes

[#] Except with no-load option (#), processor loads result of shift operation into destination accumulator. No-load option can test result of operation without destroying destination accumulator contents. Following table gives values of no-load option and bit 12 and specifies operation.

Symbol [#]	Bit 12	Operation
omitted	0	Load result into *acd*
#	1	Do not load result; restore initial CARRY flag

acs(16-31) Before execution, contains 16-bit value.

After execution, contents unchanged unless *acs* and *acd* are same accumulator.

acd(16-31) Before execution, contains 16-bit value.

After execution, contains result if no-load bit (#) is 0.

[*skip*] Processor skips next instruction if condition test true. Following table gives test conditions as specified by bits 13 to 15 and specifies operation.

Symbol [*skip*]	Bits 13-15	Operation
omitted	0 0 0	Never skip
SKP	0 0 1	Always skip
SZC	0 1 0	Skip if CARRY is 0
SNC	0 1 1	Skip if CARRY is not 0
SZR	1 0 0	Skip if result is 0
SNR	1 0 1	Skip if result is not 0
SEZ	1 1 0	Skip if either CARRY or result is 0
SBN	1 1 1	Skip if both CARRY and result are not 0

Skip omits next sequential 16-bit word. Make sure that *skip* does not transfer control to within a 32-bit or longer instruction.

Registers, Flags, and Stacks

AC0-AC3 Can be individually specified as *acs* and *acd*; otherwise unused.

CARRY Logical operation leaves initial CARRY unchanged, unless [*c*] option specified. If left or right shift occurs, final resulting CARRY is bit shifted into CARRY.

Overflow 0

PC PC + 1 (false exit)
PC + 2 (true exit)

PSR Unchanged

Stacks Unchanged

Related Instructions

WCOM Wide Complement

Exceptions

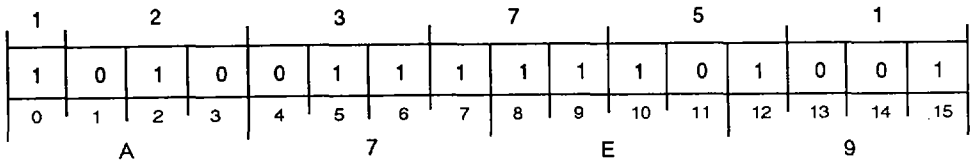
Do not specify **COM** with no-load option (#) in combination with either never skip or always skip option. Instruction may not end in 1000_2 or 1001_2 (reserved for other instructions).

Example

```
COMC 0,1,SBN        ;Complements CARRY, complements the
                    ;value in AC0,
WBR ZERERR          ;places the result in AC1, then skips
                    ;the next word if both CARRY and the
      ....          ;result are not zero.
```

Complement CARRY CRYTC

CRYTC



Function: $\overline{CRY}$ -> CRY

Parameters: None

CRYTC complements CARRY.

Arguments

None

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Complemented
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

CRYTO	Set CARRY to One
CRYTZ	Set CARRY to Zero

Exceptions

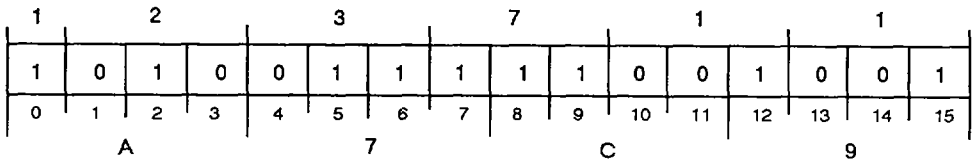
None

Example

```
CRYTC     ;CARRY is complemented.
```

Set Carry to One — CRYTO

CRYTO



Function: 1 -> CRY

Parameters: None

CRYTO unconditionally sets CARRY to 1.

Arguments

None

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Set to 1
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

CRYTC	Complement CARRY
CRYTZ	Set CARRY to Zero

Exceptions

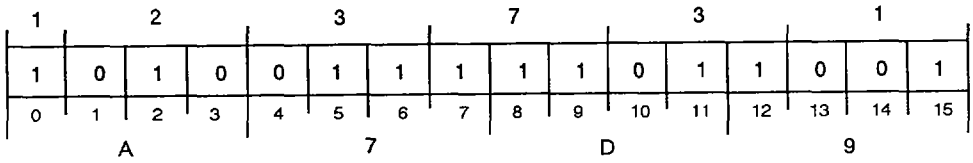
None

Example

```
CRYTO     ;CARRY set to 1.
```

Set Carry to Zero — CRYTZ

CRYTZ



Function: 0 -> CRY

Parameters: None

CRYTZ unconditionally sets CARRY to 0.

Arguments

None

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Set to 0
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

CRYTC	Complement CARRY
CRYTO	Set CARRY to One

Exceptions

None

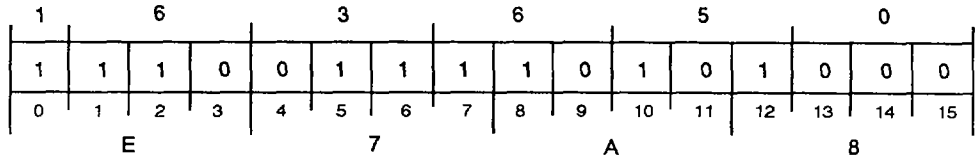
Example

```
CRYTZ      ;CARRY set to 0.
```

Character Translate CTR

ECLIPSE Instruction

CTR



Function: source @(AC3) (translated) -> destination @(AC2)

Parameters: AC0 = word address of translation table byte pointer -> unchanged
AC1 = # bytes [-2#] -> 0
AC2 = destination byte pointer -> last byte pointer + 1
AC3 = source byte pointer -> last byte pointer + 1

-OR-

Function: source @(AC3) (translated) ?=? destination @(AC2)
result code -> AC1

Parameters: AC0 = word address of translation table byte pointer -> unchanged
AC1 = # bytes [+#] -> result
-1 (S < D)
0 (S = D)
+1 (S > D)
AC2 = destination byte pointer -> last byte pointer + 1 or to failing byte
AC3 = source byte pointer -> last byte pointer + 1 or to failing byte

CTR has two different operating modes: *translate and move*, or *translate and compare*.

In the *translate and move* mode, **CTR** translates a string of bytes, one at a time, from one data representation to another and moves the translated results into a corresponding string located in another area of memory.

In the *translate and compare* mode, **CTR** translates bytes from two strings, a byte at a time from each string, and compares the translated results. If the two translated bytes are not equal, the instruction ends and returns a result code in AC1. If the two translated bytes are equal, the next two bytes are compared. This continues until either the specified number of bytes are processed, or two bytes fail the comparison.

Translation is performed by using each byte as an unsigned 8-bit integer index into a 256-byte translation table. The byte addressed by the index then becomes the translated value. Regardless of operating mode, the source strings for the translations remain unchanged upon completion of the instruction.

In both modes, the strings are processed from the specified starting addresses upward in single-byte increments. The source and destination strings may overlap in any way; however, certain overlaps may produce undesired results.

The effective addresses generated are confined within the first 64 Kbytes of the current segment.

Arguments

None

Registers, Flags, and Stacks

AC0(16–31) Before execution, contains word address, direct or indirect, of a memory location containing byte pointer to first byte of a 256-byte translation table.

After execution, contains address of word containing byte pointer to translation table.

AC1(16–31) Before execution, contains total number of bytes in each string and defines operating mode. If byte value is negative, *translate and move* mode is selected. If value is positive, *translate and compare* mode is selected.

With each byte of source string processed, value is either incremented or decremented, depending on operating mode.

After execution, value equals 0 or defines which byte is larger value (if two bytes fail to compare in *translate and compare* mode):

Code	Meaning
-1	String 1 byte < string 2 byte
0	String 1 byte = string 2 byte
+1	String 1 byte > string 2 byte
2	invalid address (protection fault error)

AC2(16–31) Before execution, contains starting byte address for destination string (string 2). With each byte accessed, address is incremented by 1.

After execution, contains either address of string 2 (destination) byte that failed to compare or, if strings processed without fault, address of byte following last byte in string.

AC3(16–31) Before execution, contains starting byte address for source string (string 1). With each byte accessed, address is incremented by 1.

After execution, contains either address of string 1 (source) byte that failed to compare or, if strings processed without fault, address of byte following last byte in string.

CARRY Unchanged

Overflow 0

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

Load byte address — Use these instructions to place addresses into AC2 and AC3.

Load effective address — Use these instructions to place a word address into AC0.

Load with immediate — Use these instructions to place a value into AC1.

WCTR Wide Character Translate

Exceptions

If the specified string length (number of bytes) is 0, the instruction is essentially a No-Op. (In *translate and compare* mode, AC1 receives a 0.)

If the contents of AC0, AC2, and AC3 are not valid addresses to some area in the user's address space, a protection fault may occur, and fault code 2 gets stored in AC1.

Because **CTR** may have a long execution time, it is interruptible. If the instruction is interrupted, the program counter is decremented by one before being saved, thus it points to the interrupted instruction. Because addresses and byte count are updated after each move, any interrupt service routine returning control via the saved program counter will correctly restart the instruction.

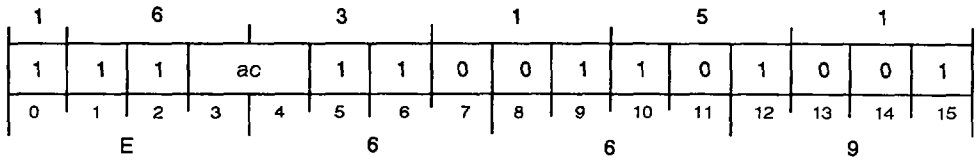
Example

```
    .ENT  START, DESSTR, SRCSTR, XLATAB
    ;This program TRANSLATES LOWERCASE characters to UPPERCASE
    ;and characters with high bit set (<200> - <377>) to normal range
    ;
    ;AC0: WORD address of byte pointer to first byte of TRANSLATION table
    ;AC1: Source length & operation MODE
    ;      (MOVE = neg, COMPARE =positive)
    ;AC2: Destination string Byte Pointer
    ;AC3: Source string Byte Pointer
START:    ELDA        1,SRLEN      ;Get length of string
          NEG         1,1          ;and negate it to set MOVE mode
          ELEF        0,XLTPTR,0   ;address of BYTE POINTER to
                                   ;TRANSLATION TABLE.
          ELEF        2,2*DESSTR,0 ;Put destination buffer BYTE POINTER
                                   ;into AC2.
          ELEF        3,2*SRCSTR,0 ;Point to first CHAR of SRCSTR,
          CTR                      ;move it until we hit delimiter or
                                   ;80th character.
          SUB         2,2          ;End of Program - RETURN TO CALLER
          ?RETURN
    ;
    ;     -- TRANSLATION TABLE --
    ;
              .TXTN   1            ;set no "<000>" at end of TXT
                                   ;string mode
              .ENABLE SWORD        ;>> set assembler to use single words
                                   ;by default <<
XLTPTR:       2*XLATAB             ;Byte pointer to XLATAB
XLATAB:   .TXT "<000><001><002><003><004><005><006><007>";0 - 7
          .TXT "<010><011><012><013><014><015><016><017>";10 - 17
          .TXT "<020><021><022><023><024><025><026><027>";20 - 27
          .TXT "<030><031><032><033><034><035><036><037>";30 - 37
          .TXT "<040>!<042>#$%&'"                        ;40 - 47
          .TXT "()*+,-./01234567"                        ;50 - 67
          .TXT "89:<073><074>=<075>?"                    ;70 - 77
          .TXT "@ABCDEFGHIJKLMNO"                        ;100 - 117
          .TXT "PQRSTUVWXYZ[\]^_"                        ;120 - 137
          .TXT "ABCDEFGHIJKLMNO"                         ;140 - 157
          .TXT "PQRSTUVWXYZ{|}~<177>"                    ;160 - 177
          .TXT "<000><001><002><003><004><005><006><007>";200 - 207
          .TXT "<010><011><012><013><014><015><016><017>";210 - 217
```

```
          .TXT "<020><021><022><023><024><025><026><027>";220 - 227
          .TXT "<030><031><032><033><034><035><036><037>";230 - 237
          .TXT "<040>!<042>#$%&'"                        ;240 - 247
          .TXT "()*+,-./01234567"                        ;250 - 267
          .TXT "89:<073><074>=<075>?"                    ;270 - 277
          .TXT "@ABCDEFGHIJKLMNO"                        ;300 - 317
          .TXT "PQRSTUVWXYZ[\]^_"                        ;320 - 337
          .TXT "ABCDEFGHIJKLMNO"                         ;340 - 357
          .TXT "PQRSTUVWXYZ{|}~<177>"                    ;360 - 377
          ;
    ;     VARIABLES
    ;
SRLEN:        13.
SRCSTR:   .TXT "ABcde<012>fg<310><311><312>KL"
DESSTR:   .BLK 100.    ;Resulting Translated string
    ;
    ;IF SRCSTR = "ABcdE<12>fg<310><311><312>KL"
    ;                     \     \    |    |
    ;                   New Line H   I    J (with HI bit set)
    ;
    ;Then the result string will be:
    ;
    ;"ABCDE<12>FGHIJKL"
    .END START
```

Convert to 16-Bit Integer CVWN

CVWN *ac*



Function: *ac*(32 bit #) -> *ac*(16 bit # [bit 16 extended])

Parameters: None

NOTE: If *ac*(bits 0–16) $\neq$ all ones or all zeros before conversion, overflow = 1

CVWN converts a 32-bit integer to a 16-bit integer.

Arguments

ac	Before execution, contains signed 32-bit integer.
	After execution, *ac*(16–31) contains signed 16-bit integer; *ac*(0–15) truncated and set to same value as bit 16.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	1, if before performing conversion, most significant 17 bits do not contain either all ones or all zeros.
PC	PC + 1
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

SEX	Sign Extend
ZEX	Zero Extend

Exceptions

None

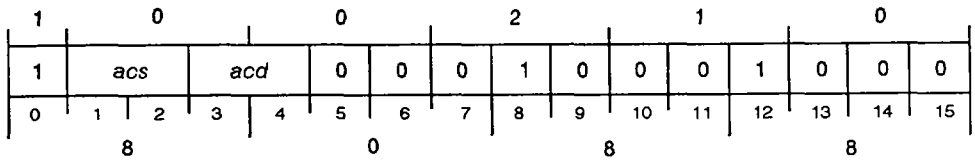
Example

```
CVWN  3   ;The processor converts AC3 to a 16-bit integer.
```

Decimal Add DAD

ECLIPSE Instruction

DAD *acs,acd*



Function: *acs*[bcd] + *acd*[bcd] + CRY -> *acd*
Decimal carry -> CRY

Parameters: None

DAD adds the decimal digit contained in *acs* to the decimal digit contained in *acd*, and adds CARRY to this result. The instruction then places the decimal units' position of the final result in *acd* and the decimal carry in CARRY.

Arguments

acs(28–31) Before execution, contains unsigned, 4-bit binary coded decimal (BCD) digit.

After execution, contents unchanged unless *acs* and *acd* are same accumulator.

acd(28–31) Before execution, contains unsigned, 4-bit binary coded decimal (BCD) digit.

After execution, contains decimal unit position of result.

Registers, Flags, and Stacks

AC0–AC3 Can be individually specified for *acs* and *acd*; otherwise unused.

CARRY Contains decimal carry

Overflow 0

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

DSB Decimal Subtract

Exceptions

No validation of the input digits is performed. Therefore, if bits 28–31 of either *acs* or *acd* contain a number greater than 9(10), the results will be unpredictable.

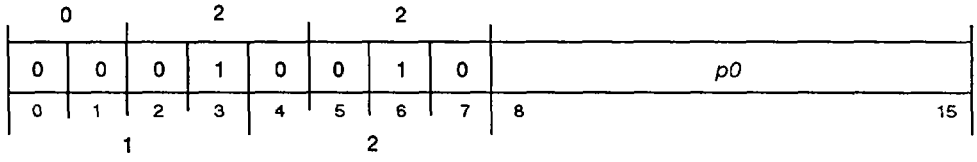
Example

```
DAD 2,3     ;Assume that bits 28-31 of AC2 contain 9, bits 28-31 of AC3
....        ;contain 7, and CARRY is 0. After this instruction is
            ;executed, AC2 remains the same; bits 28-31 of AC3 contain
            ;6 and CARRY is 1, indicating a decimal carry.
```

Add to DI DADI

Edit Subopcode

DADI *p0*



Function: DI + *p0*[2#] -> DI

Parameters: None

DADI adds the integer specified by *p0* to the Destination Indicator (DI).

Arguments

p0	Signed 8-bit integer

Registers, Flags, and Stacks

DI	DI + *p0*
Overflow	Unaffected
P	P + 2
SI	Unused

Related Instructions

DASI	Add to SI
DAPU	Add to P

Exceptions

None

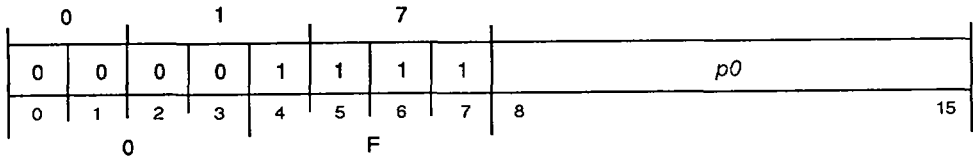
Example

```
DADI 2     ;Increments DI by 2.
```

Add to P Depending on S — DAPS

Edit Subopcode

DAPS *p0*



Function: If S = 0, P + *p0*[2#] -> P

Parameters: None

If S is 0, **DAPS** adds the integer specified by *p0* to the opcode Pointer (P). Before the addition, P is pointing to the byte containing the **DAPS** opcode.

Arguments

p0	Signed 8-bit integer

Registers, Flags, and Stacks

DI	Unused
Overflow	Unaffected
P	P + *p0* (if S = 0) P + 2 (if S = 1)
SI	Unused

Related Instructions

DAPT	Add to P Depending on T
DAPU	Add to P

Exceptions

None

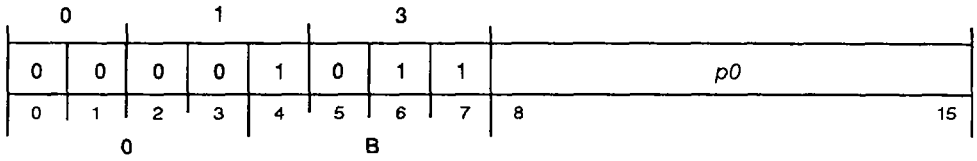
Example

```
DAPS -4    ;Decrements P by 4 if the source integer is positive.
```

Add to P Depending on T — DAPT

Edit Subopcode

DAPT *p0*



Function: If T = 1, P + *p0*[2#] -> P

Parameters: None

If T is one, **DAPT** adds the integer specified by *p0* to the opcode Pointer (P). Before the addition, P is pointing to the byte containing the **DAPT** opcode.

Arguments

p0	Signed 8-bit integer

Registers, Flags, and Stacks

Overflow	Unaffected
P	P + *p0* (if T = 1) P + 2 (if T = 0)
T	Unchanged

Related Instructions

DAPS	Add to P Depending on S
DAPU	Add to P

Exceptions

None

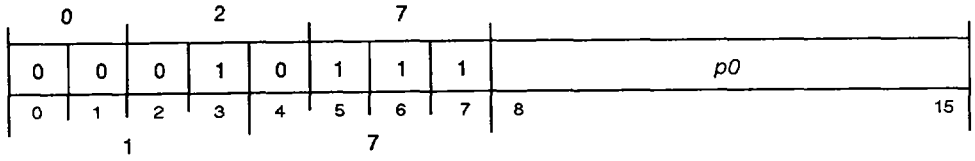
Example

```
DAPT 4     ;Increment P by 4 if the previous digit was nonzero.
```

Add to P — DAPU

Edit Subopcode

DAPU *p0*



Function: P + *p0*[2#] -> P

Parameters: None

DAPU adds the integer specified by *p0* to the opcode Pointer (P). Before the add is performed, P points to the byte containing the **DAPU** opcode.

Arguments

p0	Signed 8-bit integer

Registers, Flags, and Stacks

DI	Unused
Overflow	Unaffected
P	P + *p0*
SI	Unused

Related Instructions

DAPS	Add to P Depending on S
DAPT	Add to P Depending on T

Exceptions

None

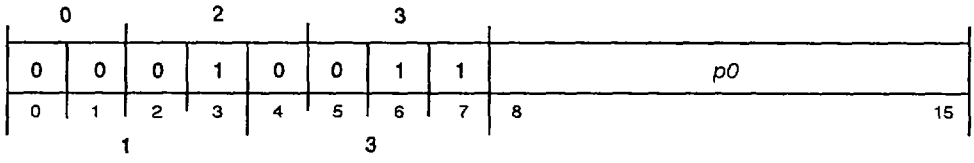
Example

```
DAPU 0     ;Loops on this instruction.
```

Add to SI DASI

Edit Subopcode

DASI *p0*



Function: SI + *p0*[2#] -> SI

Parameters: None

DASI adds the integer specified by *p0* to the Source Indicator (SI).

Arguments

p0	Signed 8-bit integer

Registers, Flags, and Stacks

DI	Unused
Overflow	Unaffected
P	P + 2
SI	SI + *p0*

Related Instructions

DADI	Add to DI

Exceptions

If data type is 5 (packed decimal), *p0* is treated as a digit quantity, rather than a byte quantity.

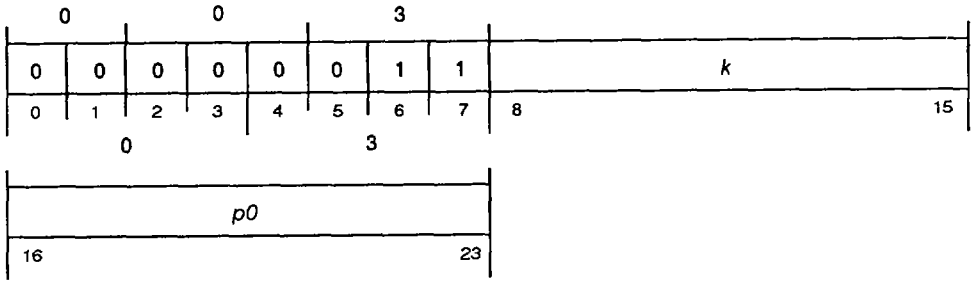
Example

```
DASI -2    ;Decrement SI by 2.
```

Decrement and Jump if Nonzero — DDTK

Edit Subopcode

DDTK *k,p0*



Function: (*k*) [#] - 1 -> (*k*)
If (*k*) ≠ 0, P + *p0*[2#] -> P

Parameters: None

DDTK decrements a value in the stack by one. If the decremented value is nonzero, **DDTK** adds the integer specified by *p0* to the opcode Pointer (P). Before the addition, P is pointing to the byte containing the **DDTK** opcode.

Arguments

k	Signed 8-bit integer
p0	Signed 8-bit integer

Registers, Flags, and Stacks

Overflow	Unaffected
P	P + 3 (stack word = 0) P + *p0* (stack word ≠0)
Stack	For **EDIT**: if integer specified by *k* negative, word decremented is at address: narrow stack pointer + 1 + *k*. If *k* positive, word decremented is at address: narrow frame pointer + 1 + *k*. For **WEDIT**: if integer specified by *k* negative, double word decremented is at address: WSP + 2 + 2**k*. If *k* positive, double word decremented is at address: WFP + 2 + 2**k*.

Related Instructions

DSTK	Store in Stack

Exceptions

None

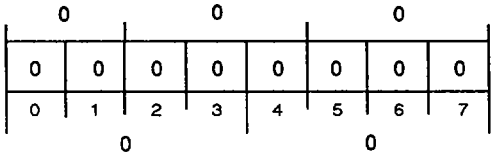
Example

```
WEDIT               ;Call Edit subprogram.
                    ;Subprogram.
      ...
DDTK -2,2           ;Decrement the double word at the wide stack
                    ;pointer + 6. if this value is not 0, add 2 to P.
```

End Edit DEND

Edit Subopcode

DEND



Function: Stop Edit(ing)

Parameters: None

DEND terminates the Edit subprogram.

Arguments

None

Registers, Flags, and Stacks

Overflow	Unaffected
P	P + 1
Stack	Unchanged

Related Instructions

EDIT, WEDIT

Exceptions

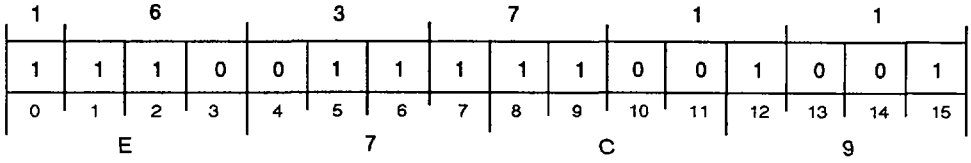
None

Example

```
DEND   ;Terminates Edit subprogram.
```

Dequeue a Queue Data Element DEQUE

DEQUE
(deque last element or from empty queue return)
(deque from queue with two or more elements return)



Function: Queue - element --> Queue

Parameters: AC0 = E(Q descriptor) --> unchanged
AC1 = E(data element) --> unchanged

NOTE: If AC1 = -1, first element at queue head is dequeued and AC1 is updated.

DEQUE dequeues a data element. The instruction checks all reads and writes of links in data elements and queue descriptors against the current segment. Segment numbers of the link addresses must be greater than or equal to number of the current segment.

DEQUE cannot be interrupted. The entire operation completes before any interrupts are enabled. If the **DEQUE** instruction is successful, entry into the next instruction is guaranteed to occur without an intervening interruption.

The instruction requires up to nine pages to be resident in memory. (The worst case occurs when the element to be dequeued is between two other elements and when all of the elements and the queue header have one of their affected links on a page boundary. Then eight pages are required, in addition to page zero of the current segment.) The instruction first reads all links required to complete the dequeing operation. If a page fault occurs, **DEQUE** starts again reading all the links until no page fault occurs. When all required pages are resident, the instruction attempts to dequeue the data element.

Arguments

None

Registers, Flags, and Stacks

AC0 Before execution, contains effective address of queue descriptor.

After execution, contents unchaned.

If dequeing element at head or tail of queue, queue descriptor updated with effective address of data element that becomes new head or tail of queue. If dequeing last element in queue, this means both links in the queue descriptor will be set to -1.

AC1 Before execution, specifies element to be dequeued.

Instruction dequeues an element by writing -1 in forward and backward links of the element being dequeued. The forward link of the element preceding the element being dequeued is updated to the value in the forward link of the element being dequeued. Similarly, the backward link of the element following the element being dequeued is set to the value of the backward link of the element being dequeued.

If AC1 does not contain −1, accumulator contains effective address of data element to be deleted. If AC1 contains −1, accumulator deletes head of queue, obtained from queue descriptor pointed to by effective address in AC0.

After execution, contains address of element dequeued.

If dequeuing from empty queue, unchanged.

If dequeuing from queue with one or more data elements, AC1 updated with address of dequeued data element.

AC2–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1 (dequeuing last data element or dequeuing from an empty queue)
	PC + 2 (dequeuing from queue with two or more data elements)
PSR	Unchanged
Stack	Unchanged

Related Instructions

ENQH	Use this instruction to enqueue a new data element before another element.
ENQT	Use this instruction to enqueue a new data element behind another element.

Exceptions

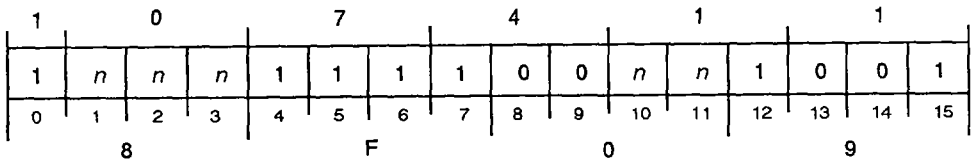
None

Example

```
        ;This subroutine deques 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
              ;    <return>
              ;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 dequeueing element 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!
```

Detected Error DERR

DERR *nn*



Function: PC(**DERR**) -> stack
5-bit error code(zero extended) -> stack
If Page Zero Pointer (bit 0) = 1, then **DERR** checks for stack overflow
If Page Zero Pointer (bit 0) = 0, then **DERR** ignores any stack overflow
CRY = ? -> unchanged

Parameters: PC = **DERR** -> user-supplied error (or trap) handler

DERR pushes onto the wide stack the PC of the **DERR** instruction and a 5-bit code. The instruction then jumps to a user-supplied error handler through a pointer in location 47(8) of page zero of the current segment. Based on the value of bit 0 of this pointer, the instruction either checks for or ignores any stack overflow.

Arguments

nn	Contains five-bit error code. Processor zero extends code to 32 bits before pushing it onto stack.

Registers, Flags, and Stacks

AC0-AC3	Unused
CARRY	Unchanged
Overflow	0
Page Zero Location 47	Nonindirectable 16-bit pointer to error handler in first 64 Kbytes of current segment. If bit 0 of pointer is one, instruction checks for stack overflow. If bit 0 is zero, instruction ignores stack overflow.
PC	Address of error handler
PSR	Unchanged
Stack	Two top double words of wide stack contain address of **DERR** instruction followed by zero-extended error code. Updated WSP points to error code.

Related Instructions

WPOP	Use this instruction to pop the error code from the wide stack.

Exceptions

Stack overflow	Action dependent on bit 0 of Page Zero pointer.

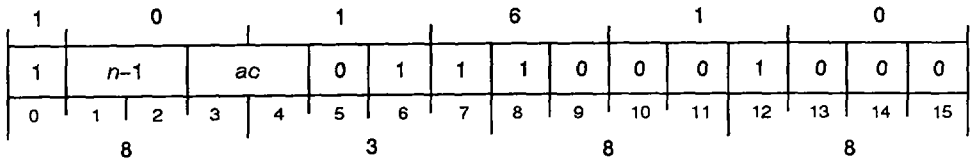
Example

```
...
XNLDA 0,ARG0        ;
XNLDA 1,ARG1        ;Compare the contents of accumulators,
WSNE  0,1           ;conditionally skipping the next word.
DERR  12            ;If the DERR instruction is executed,
                    ;the PC of the DERR instruction and
...                 ;the error code are pushed onto the
                    ;wide stack. The pushed
WSEQ  1,2           ;code may then be used by the error
DERR  13            ;handling routine to indicate where
...                 ;in the program the error occurred.
WUSGE 1,3           ;
DERR  14            ;
...       ;
```

Double Hex Shift Left DHXL

ECLIPSE Instruction

DHXL *n,ac*



Function: shift *ac*&(*ac*+1) left (*n**4) -> *ac*&(*ac*+1)

Parameters: *ac* = high-order -> high-order result
ac+1 = low-order -> low-order result

DHXL shifts the 32-bit value contained in *ac* and *ac*+1 left 1 to 4 hex digits depending upon the immediate field *n*. Bits shifted out are lost, and the vacated bit positions are filled with zeros.

Arguments

n	Integer in range 1-4. If *n* equals 4, contents of *ac*+1(16-31) placed in *ac*(16-31) and *ac*+1(16-31) filled with zeros. Assembler takes coded value of *n* and subtracts one from it before placing it in immediate field. Thus, programmer should code exact number of hex digits to be shifted.
ac(16-31)	Before execution, contains 16-bit value. After execution, contains result of operation.
ac+1(16-31)	Before execution, contains 16-bit value. If *ac* specified as AC3, then *ac*+1 is AC0. After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

DHXR	Double Hex Shift Right
HXL	Hex Shift Left
HXR	Hex Shift Right

Exceptions

None

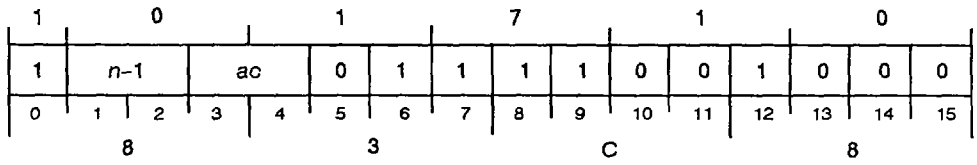
Example

```
SUB     0,0     ;Start with zeros in AC0.
ADC     1,1     ;Start with ones in AC1.
DHXL    3,0     ;Do the left shift.
                ;AC0[16-31] now has 007777(8).
                ;AC1[16-31] now has 170000(8).
```

Double Hex Shift Right DHXR

ECLIPSE Instruction

DHXR *n,ac*



Function: shift *ac*&(*ac*+1) right (*n**4) -> *ac*&(*ac*+1)

Parameters: *ac* = high-order -> high-order result
ac+1 = low-order -> low-order result

DHXR shifts the value contained in *ac* and *ac*+1 to the right 1 to 4 hex digits, depending upon the immediate field *n*. Bits shifted out are lost, and the vacated bit positions are filled with zeros.

Arguments

n	Integer in range 1 to 4. If *n* equals 4, the contents of *ac*(16–31) placed in *ac*+1(16–31), and *ac*(16–31) filled with zeros. Assembler takes coded value of *n* and subtracts one from it before placing it in immediate field. Thus, programmer should code exact number of hex digits to be shifted.
ac(16–31)	Before execution, contains 16-bit value. After execution, contains result of operation.
ac+1(16–31)	Before execution, contains 16-bit value. If *ac* specified as AC3, then *ac*+1 is AC0. After execution, contains result.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

DHXL	Double Hex Shift Left
HXL	Hex Shift Left
HXR	Hex Shift Right

Exceptions

None

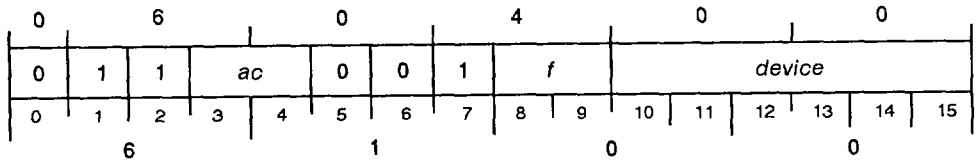
Example

```
ADC     0,0     ;Start with ones in AC0.
SUB     1,1     ;Start with zeros in AC1.
DHXR    3,0     ;Do the right shift.
                ;AC0[16-31] now has 000017(8).
                ;AC1[16-31] now has 177760(8).
```

Data In A Buffer DIA

ECLIPSE Instruction

DIA[*f*] *ac,device*



Function: *device* (A buffer) -> *ac*
[*f*] -> BUSY, DONE flags

Parameters: None

DIA transfers data from the A buffer of specified I/O device on the default I/O channel (IOC) to specified accumulator. After the data transfer, the BUSY and DONE flags are set according to the function specified by *f*.

Arguments

f — Specify from **S**, **C**, and **P** for desired I/O device flag control as follows:

f	BUSY	DONE
(option omitted)	No effect	No effect
S	Set to a 1	Set to a 0
C	Set to a 0	Set to a 0
P	Pulses a special I/O bus control line	

ac(16-31) — After execution, contains data from A buffer of specified I/O device. Number of data bits moved depends upon size of buffer and mode of operation of device. Bits in accumulator that do not receive data are set to 0.

device — Specify either mnemonic or device code for desired I/O device.

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

DIB, DIC	Transfer data from buffer of I/O device to an accumulator.
DOA, DOB, DOC	Transfer data from an accumulator to the buffer of an I/O device.

Exceptions

None

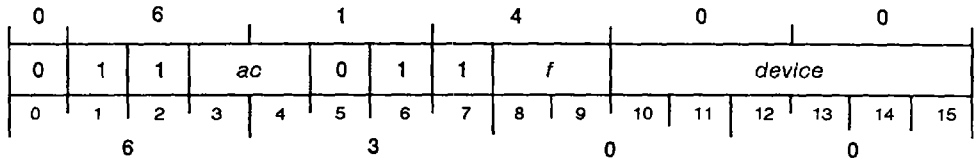
Example

```
DIAC  0,27        ;Move into AC0 the contents of the A
                  ;buffer of device 27 on the default
                  ;IOC, and clear the device's BUSY and
                  ;DONE flags.
```

Data In B Buffer DIB

ECLIPSE Instruction

DIB[*f*] *ac,device*



Function: device (B buffer) -> *ac*
[*f*] -> BUSY, DONE flags

Parameters: None

DIB transfers data from the B buffer of specified I/O device on the default I/O channel (IOC) to specified accumulator. After the data transfer, the BUSY and DONE flags are set according to the function specified by *f*.

Arguments

f — Specify from **S**, **C**, and **P** for desired I/O device flag control as follows:

f	BUSY	DONE
(option omitted)	No effect	No effect
S	Set to a 1	Set to a 0
C	Set to a 0	Set to a 0
P	Pulses a special I/O bus control line	

ac(16–31) — After execution, contains data from B buffer of specified I/O device. Number of data bits moved depends upon size of buffer and mode of operation of device. Bits in accumulator that do not receive data are set to 0.

device — Specify either mnemonic or device code for desired I/O device.

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, DIC	Transfer data from buffer of I/O device to an accumulator.
DOA, DOB, DOC	Transfer data from an accumulator to the buffer of an I/O device.

Exceptions

None

Example

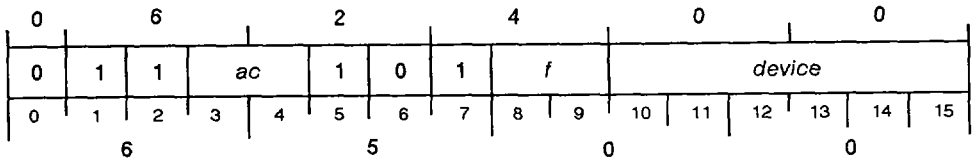
```
DIB    2,27        ;Read into AC2 the contents of the B buffer
                   ;of device 27 on the default IOC and do not
                   ;modify the device's BUSY and DONE flags.
```

Data In C Buffer

DIC

ECLIPSE Instruction

DIC[*f*] *ac,device*



Function: *device* (C buffer) -> *ac*
[*f*] -> BUSY, DONE flags

Parameters: None

DIC transfers data from the C buffer of specified I/O device on the default I/O channel (IOC) to specified accumulator. After the data transfer, the BUSY and DONE flags are set according to the function specified by *f*.

Arguments

f — Specify from **S**, **C**, and **P** for desired I/O device flag control as follows:

f	BUSY	DONE
(option omitted)	No effect	No effect
S	Set to a 1	Set to a 0
C	Set to a 0	Set to a 0
P	Pulses a special I/O bus control line	

ac(16–31) — After execution, contains data from C buffer of specified I/O device. Number of data bits moved depends upon size of buffer and mode of operation of device. Bits in accumulator that do not receive data are set to 0.

device — Specify either mnemonic or device code for desired I/O device.

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, DIB	Transfer data from buffer of I/O device to an accumulator.
DOA, DOB, DOC	Transfer data from an accumulator to the buffer of an I/O device.

Exceptions

The assembler reserves the **DICC 0,CPU (IORST)** instruction for resetting I/O functions.

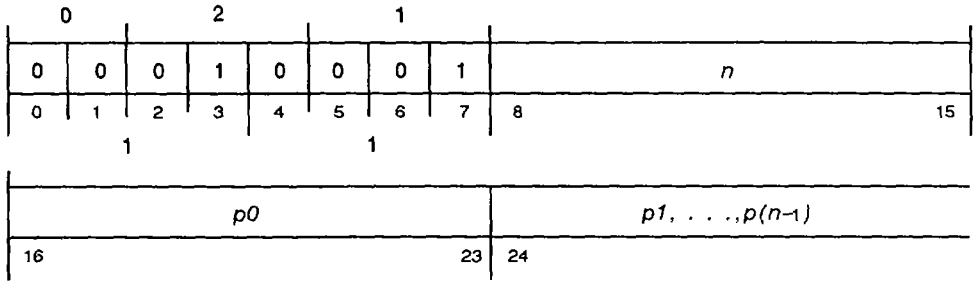
Example

```
DICS  1,27        ;Read into AC1 the contents of the C buffer
                  ;of device 27 on the default IOC, and start a
                  ;new operation on the device by clearing its
                  ;DONE flag and setting its BUSY flag.
```

Insert Characters Immediate DICI

Edit Subopcode

DICI *n,p0[,p1,...,p(n-1)]*



Function:
p0[char] -> (DI)
p1[char] -> (DI + 1)
...
p(n-1)[char] -> (DI + *n*-1)
P + *n* + 2 -> P
DI + *n* -> DI

Parameters: n = #(0-377(8))

DICI inserts *n* characters from the opcode stream into the destination field, beginning at the position specified by DI. It then increases P by *n* + 2, and increases DI by *n*.

Arguments

n Unsigned 8-bit integer

p0[,p1,...,p(n-1)]

8-bit characters

Registers, Flags, and Stacks

DI	DI + *n*
Overflow	Unaffected
P	P + *n* + 2
SI	Unused

Related Instructions

DIMC	Insert Character J Times
DINC	Insert Character Once
DIMT	Insert Character Suppress

Exceptions

None

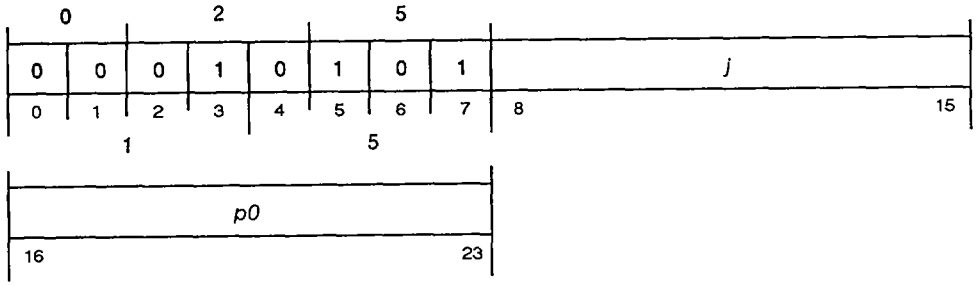
Example

```
DICI   3,JWB        ;Insert "J" at DI, "W" at DI+1, "B" at DI+2.
                    ;After execution, DI = DI+3, P = P+5.
```

Insert Character J Times — DIMC

Edit Subopcode

DIMC *j,p0*



Function: *p0*[char] -> (DI)
...
p0 -> (DI + *j*-1)
DI + *j* -> DI

Parameters: None

DIMC inserts a character, specified by *p0*, into the destination field a number of times equal to *j*. It then increases DI by *j*.

Arguments

j	8-bit value
p0	8-bit character

Registers, Flags, and Stacks

DI	Before execution, contains byte pointer to first byte of destination field.
	After execution, DI + *j*
Overflow	Unaffected
P	P + 3
SI	Unused

Related Instructions

DICI	Insert Characters Immediate
DINC	Insert Character J Times
DINT	Insert Character Suppress

Exceptions

None

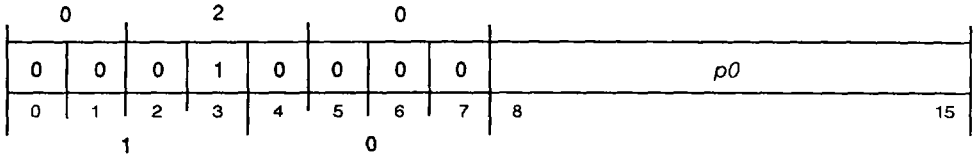
Example

```
DIMC 3,J  ;Insert the character""J" at DI, DI+1, and
          ;DI+2. After execution, DI = DI+3.
```

Insert Character Once DINC

Edit Subopcode

DINC *p0*



Function: *p0*[char] -> (DI)
DI + 1 -> DI

Parameters: None

DINC inserts the character specified by *p0* into the destination field at the position specified by DI. It then increments DI by one.

Arguments

p0	8-bit character

Registers, Flags, and Stacks

DI	Before execution, contains byte pointer to first byte of destination field.
	After execution, DI + 1.
Overflow	Unaffected
P	P + 2
SI	Unused

Related Instructions

DICI	Insert Characters Immediate
DIMC	Insert Character J Times
DINT	Insert Character Suppress

Exceptions

None

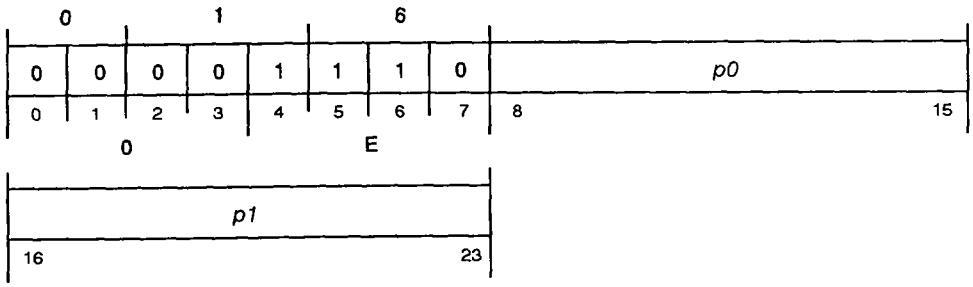
Example

```
DINC J     ;Insert "J" at DI.
```

Insert Sign DINS

Edit Subopcode

DINS *p0,p1*



Function: If S = 0, *p0*[char] -> (DI)
If S = 1, *p1*[char] -> (DI)
DI + 1 -> DI

Parameters: None

DINS inserts a character into the destination field. The value of the Sign flag (S) determines which character to insert: if S = 0, insert *p0*; if S = 1, insert *p1*.

Arguments

p0,p1	8-bit characters

Registers, Flags, and Stacks

DI	Before execution, contains byte pointer to first byte of destination field.
	After execution, DI + 1.
Overflow	Unaffected
P	P + 3
SI	Unused

Related Instructions

DSSO	Set S to One
DSSZ	Set S to Zero

Exceptions

None

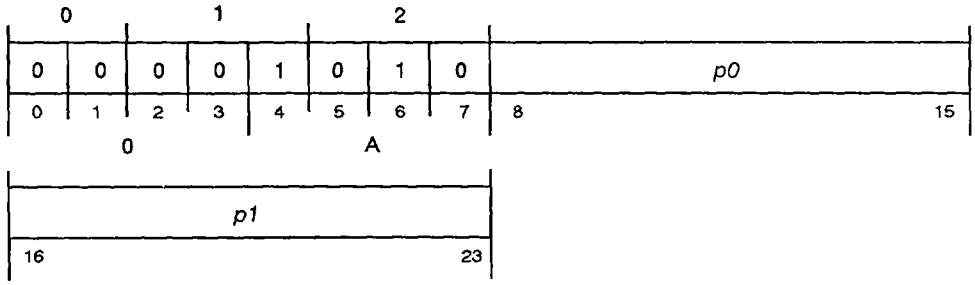
Example

```
DINS A,Z  ;If S = 0, insert "A" at DI; otherwise insert "Z."
```

Insert Character Suppress DINT

Edit Subopcode

DINT *p0,p1*



Function: If T = 0, *p0*[char] -> (DI)
If T = 1, *p1*[char] -> (DI)
DI + 1 -> DI

Parameters: None

DINT inserts a character into the destination field at the position specified by DI. The character inserted depends on the value of T. If T = 0, insert *p0*; if T = 1, insert *p1*.

Arguments

p0,p1	8-bit characters

Registers, Flags, and Stacks

DI	DI + 1
Overflow	Unaffected
P	P + 3
SI	Unused

Related Instructions

DICI	Insert Characters Immediate
DIMC	Insert Character J Times
DINC	Insert Character Once

Exceptions

None

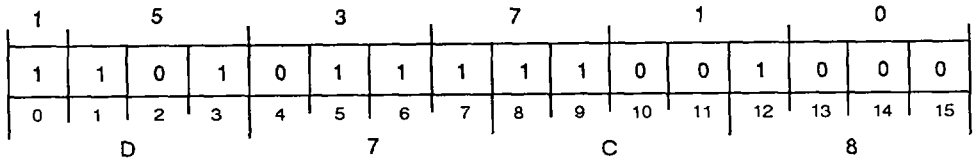
Example

```
DINT Z,A  ;If T = 0, insert "Z" at DI; otherwise insert "A."
```

Unsigned Divide DIV

ECLIPSE Instruction

DIV



Function: AC0&AC1 / AC2 -> AC1(quotient)&AC0(remainder)
0 -> CRY

Parameters: AC0 = high dividend -> remainder
AC1 = low dividend -> quotient
AC2 = divisor -> unchanged

NOTE: If AC0>=AC2 or if AC2=0, then: 1->CRY, AC0&AC1 = unchanged, instruction terminates

DIV divides the unsigned 32-bit integer contained in AC0 and AC1 by the unsigned 16-bit integer in AC2. The quotient and remainder are placed in AC1 and AC0, respectively.

Arguments

None

Registers, Flags, and Stacks

AC0(16-31)	Before execution, contains high half of unsigned 32-bit dividend.
	After execution, contains unsigned 16-bit remainder.
AC1(16-31)	Before execution, contains low half of unsigned 32-bit dividend.
	After execution, contains unsigned 16-bit quotient.
AC2(16-31)	Contains unsigned 16-bit divisor.
	After execution, contents unchanged.
CARRY	Set to 1 if AC0 >= AC2 or if AC2 = 0; otherwise 0.
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

DIVS	Signed Divide
DIVX	Sign Extend and Divide
NDIV	Narrow Divide
WDIV	Wide Divide
WDIVS	Wide Signed Divide

Exceptions

If AC0 is equal to or greater than AC2, or if AC2 initially contains 0, then operation terminates, CARRY contains 1, and AC0 and AC1 are unchanged.

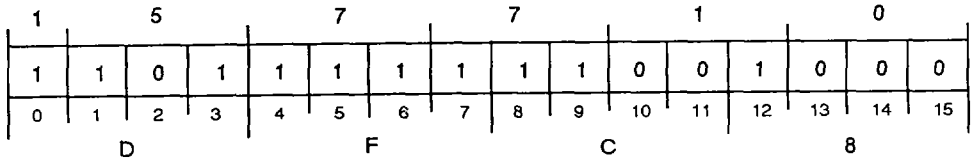
Example

```
DIV                 ;If the unsigned integer in AC0 and AC1 is
MOV#    0,0,SZR     ;evenly divisible by the unsigned integer
JMP     NOTEVEN     ;in AC2, jump to YESEVEN. Otherwise, jump
JMP     YESEVEN     ;to NOTEVEN.
```

Signed Divide DIVS

ECLIPSE Instruction

DIVS



Function: AC0&AC1 / AC2 -> AC1(quotient)&AC0(remainder)
0 -> CRY

Parameters: AC0 = high dividend -> remainder
AC1 = low dividend -> quotient
AC2 = divisor -> unchanged

NOTE: If quotient overflows: 1->CRY, AC0&AC1 = ?, instruction terminates

DIVS divides the signed 32-bit integer in AC0 and AC1 by the signed 16-bit integer in AC2. The quotient and remainder are placed in AC1 and AC0, respectively.

Arguments

None

Registers, Flags, and Stacks

AC0(16-31)	Before execution, contains high half of signed 32-bit dividend.
	After execution, contains signed 16-bit remainder. Sign of remainder is always same as sign of dividend, except that zero quotient or zero remainder always positive.
AC1(16-31)	Before execution, contains low half of signed 32-bit dividend.
	After execution, contains signed 16-bit quotient. Sign of quotient determined by rules of algebra.
AC2(16-31)	Contains signed 16-bit divisor.
	After execution, contents unchanged.
AC3	Unused
CARRY	Set to 1, if AC2=0 or if quotient too large; otherwise 0.
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

DIV	Unsigned Divide
DIVX	Sign Extend and Divide
NDIV	Narrow Divide
WDIV	Wide Divide
WDIVS	Wide Signed Divide

Exceptions

If quotient too large to fit into bits 16–31 of AC1, CARRY set to 1 and operation terminates. Contents of AC0 and AC1 unpredictable.

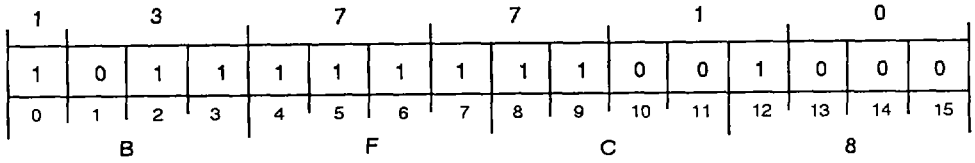
Example

```
DIVS                ;If the signed integer in AC0 and AC1 is
ZEX    0,0          ;evenly divisible by the signed integer
WSEQ   0,0          ;in AC2, jump to YESEVEN. Otherwise, jump
JMP    NOTEVEN      ;to NOTEVEN.
JMP    YESEVEN
```

Sign Extend and Divide DIVX

ECLIPSE Instruction

DIVX



Function: AC0&AC1 / AC2 -> AC1(quotient)&AC0(remainder)
0 -> CRY

Parameters: AC0 = sign extension of AC1 -> remainder
AC1 = low dividend -> quotient
AC2 = divisor -> unchanged

NOTE: If quotient overflows: 1 -> CRY, AC0&AC1 = ?, instruction terminates

DIVX extends the sign of the number in AC1 into AC0 by placing a copy of bit 16 of AC1 into bits 16–31 of AC0. After extending the sign, **DIVX** performs a signed divide operation.

Arguments

None

Registers, Flags, and Stacks

AC0(16–31)	After execution, contains signed 16-bit remainder.
AC1(16–31)	Before execution, contains signed 16-bit dividend.
	After execution, contains signed 16-bit quotient.
AC2(16–31)	Before execution, contains signed 16-bit divisor.
	After execution, contents unchanged.
AC3	Unused
CARRY	0
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

DIV	Unsigned Divide
DIVS	Signed Divide
NDIV	Narrow Divide
WDIV	Wide Divide
WDIVS	Wide Signed Divide

Exceptions

If the quotient overflows, CARRY is set to 1, the contents of AC0 and AC1 are undefined, and the instruction terminates.

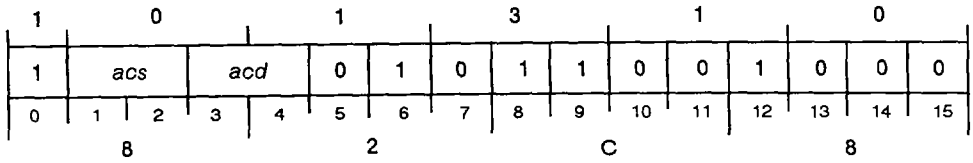
Example

```
DIVX                ;If the signed integer in AC1 is evenly
ZEX     0,0         ;divisible by the signed integer in AC2,
WSEQ    0,0         ;jump to YESEVEN. Otherwise, jump to NOTEVEN.
JMP     NOTEVEN
JMP     YESEVEN
```

Double Logical Shift — DLSH

ECLIPSE Instruction

DLSH *acs,acd*



Function: shift *acd*&(*acd*+1)(*acs*(bits 24–31[+ = left,- = right]))
-> *acd*&(*acd*+1)

Parameters: *acd* = high-order -> high-order result
acd+1 = low-order -> low-order result

DLSH shifts the 32-bit value contained in *acd* and *acd*+1 either left or right depending on the number contained in *acs*. Bits shifted out are lost and the vacated bit positions are filled with zeros.

Arguments

acs(24–31)	Before execution, contains signed 8-bit integer which determines direction of shift and number of bits to be shifted. If number is positive, shifting is to left. If number is negative, shifting is to right. If number is zero, no shifting is performed. Number of bits to be shifted is equal to magnitude of number. After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd(16–31)	Before execution, contains high-order 16 bits of 32-bit value. After execution, contains high-order 16 bits of result.
acd+1(16–31)	Before execution, contains low-order 16 bits of 32-bit value. AC3 + 1 is AC0. After execution, contains low-order 16 bits of result.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

LSH	Logical Shift
WLSH	Wide Logical Shift

Exceptions

If the magnitude of the number in *acs* is greater than 31_{10}, bits 16–31 of *acd* and *acd*+1 are set to 0.

Example

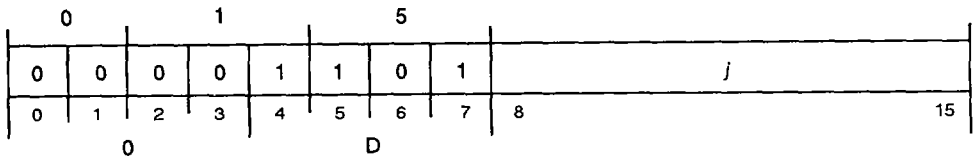
```
DLSH 3,1
....
```

;Shift the contents of AC3 and AC0 in the
;magnitude and direction indicated by the
;contents of AC1. Assume that AC3 contains
;66666_8, AC0 contains 155555_8, and AC1
;contains -5. Following execution, AC3 will
;contain 1555_8 and AC3 will contain 133333_8.

Move Alphabetics DMVA

Edit Subopcode

DMVA *j*



Function:
(SI) -> (DI)
(SI + 1) -> (DI + 1)
...
(SI + *j*-1) -> (DI + *j*-1)
SI + *j* -> SI
DI + *j* -> DI
1 -> T

Parameters: None

DMVA moves *j* characters from the source field to the destination field. It then increases both SI and DI by *j* and sets T to 1.

Arguments

j	8-bit value

Registers, Flags, and Stacks

DI	Before execution, contains byte pointer to first byte of destination field.
	After execution, DI + *j*
Overflow	Unaffected
P	P + 2
SI	Before execution, contains byte pointer to first byte of source field.
	After execution, SI + *j*.
T	Set to 1

Related Instructions

DMVC	Move Characters

Exceptions

If the source field data type is 5 (packed), or if any of the characters moved is not an alphabetic (A-Z, a-z, or space), **DMVA** initiates a commercial fault.

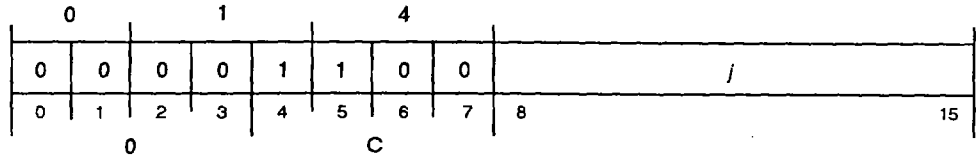
Example

```
DMVA  5   ;Move 5 alphabetic characters from source to
          ;destination.
```

Move Characters DMVC

Edit Subopcode

DMVC *j*



Function:
(SI) -> (DI)
(SI + 1) -> (DI + 1)
...
(SI + *j*-1) -> (DI + *j*-1)
SI + *j* -> SI
DI + *j* -> DI
1 -> T

Parameters: None

NOTE: If source data type = 5, then commercial fault initiated, no action performed.

DMVC moves *j* characters from the source field to the destination field. It then increases SI and DI by *j*, and sets T to 1. **DMVC** performs no validation of the characters moved.

Arguments

j	8-bit value

Registers, Flags, and Stacks

DI	Before execution, contains byte pointer to first byte of destination field.
	After execution, DI + *j*.
Overflow	Unaffected
P	P + 2
SI	Before execution, contains byte pointer to first byte of source field.
	After execution, SI + *j*.
T	Set to 1

Related Instructions

DMVA	Move Alphabetics

Exceptions

If the data descriptor indicates that the source is data type 5 (packed), **DMVC** initiates a commercial fault and performs no other action.

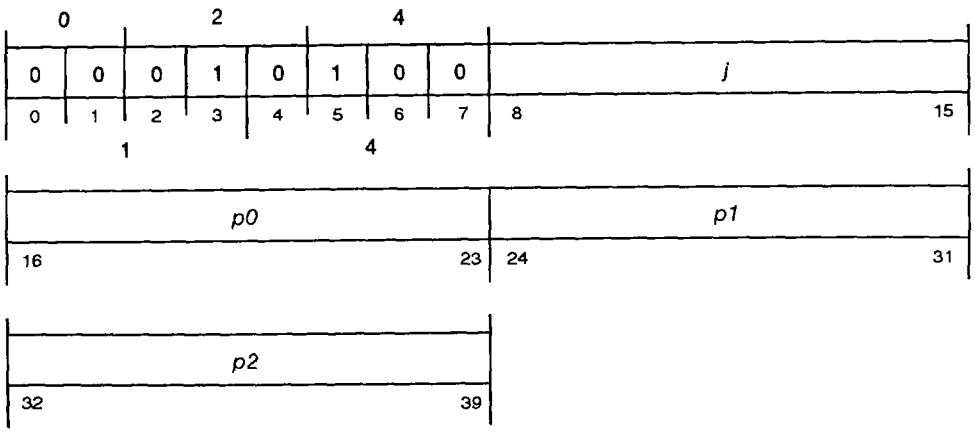
Example

```
DMVC  3   ;Moves 3 characters from source field to destination field.
```

Move Float DMVF

Edit Subopcode

DMVF *j,p0,p1,p2*



Function:
```
j(SI) --> (DI)
If T =1
        ASCII(SI) --> DI
        DI = DI + 1
 If T = 0
        and digit = 0 or space
                p0 --> DI
        DI = DI + 1
                and digit = non-0
                T = 1
         If S = 0, p1 --> DI
         If S = 1, p2 --> DI
        DI = DI + 1
        ASCII(SI) --> DI
        DI = DI + 1
```

Parameters: None

DMVF moves *j* digits from the source integer to the destination field. Each digit is handled as follows:

If T is 1, **DMVF** writes the unpacked (ASCII) version of the digit at the location specified by DI and increments DI.

If T is 0, and

the digit is a zero or space, **DMVF** writes *p0* at the location specified by DI, and increments DI.

the digit is nonzero, **DMVF** sets T to 1, and the characters placed in the destination field depend on S.

If S is 0, **DMVF** writes *p1* at the location specified by DI.

If S is 1, **DMVF** writes *p2* at the location specified by DI.

DMVF then increments DI, writes the unpacked (ASCII) version of the digit at the location specified by DI, and increments DI again.

Arguments

j	8-bit value
p0,p1,p2	8-bit characters

Registers, Flags, and Stacks

DI	Before execution, contains byte pointer to first byte in destination field. After execution, DI + *j*. If T = 0, and digit is nonzero, DI + *j* + 1.
Overflow	Unaffected
P	P + 5
S	Unchanged
SI	The **EDIT** and **WEDIT** instruction definitions explain how SI is affected by each digit transfer.
T	Unchanged

Related Instructions

DNDF	End Float

Exceptions

DMVF initiates a commercial fault if any of the digits processed is not valid for the specified data type.

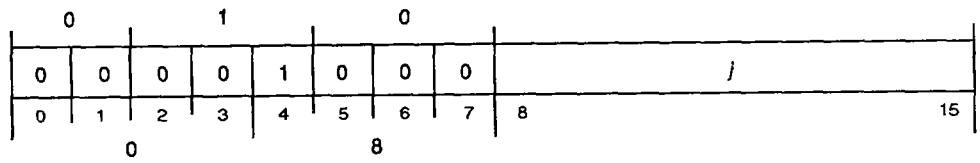
Example

```
;Move 8 digits from the source integer to the destination field. All
;leading zero digits will be replaced with a space character. When
;a nonzero digit is encountered, move the sign into the destination
;field ('+' for positive, '-' for negative), and then move the actual
;digit into the destination field. Then proceed to move ASCII digits
;to the destination field.
;
DMVF   10,40,53,55              ;All values are octal.
```

Move Numerics DMVN

Edit Subopcode

DMVN *j*



Function:
(SI) -> (DI)
(SI + 1) -> (DI + 1)
...
(SI + *j*-1) -> (DI + *j*-1)
DI + *j* -> DI
1 -> T

Parameters: None

DMVN moves *j* digits from the source integer to the destination field, beginning at the position specified by DI, incrementing DI after each digit is transferred. Each digit written into the destination field is the unpacked (ASCII) version of the digit. **DMVN** sets T to 1.

Arguments

j	8-bit value

Registers, Flags, and Stacks

DI	DI + *j*
Overflow	Unaffected
P	P + 2
SI	The **EDIT** and **WEDIT** instruction definitions explain how SI is affected by each digit transfer.
T	Set to 1

Related Instructions

DMVS	Move Number with Zero Suppression

Exceptions

DMVN initiates a commercial fault if the characters moved are not valid for the specified data type.

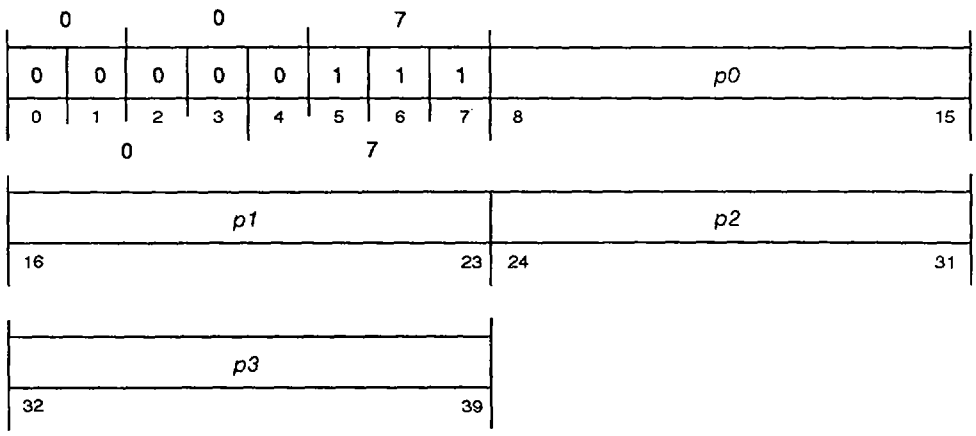
Example

```
DMVN  7   ;Move 7 ASCII digits from the source to the destination.
```

Move Digit with Overpunch DMVO

Edit Subopcode

DMVO *p0,p1,p2,p3*



Function: (SI) translated --> (DI)
DI + 1 --> DI
If (SI) ≠ 0, then T --> 1
If (SI) = 0 AND S = 0, translation = *p0*
If (SI) = 0 AND S = 1, translation = *p1*
If (SI) ≠ 0 AND S = 0, translation = (SI) + *p2*
If (SI) ≠ 0 AND S = 1, translation = (SI) + *p3*

Parameters: None

DMVO reads one digit from the source integer, and stores a translation of the digit in the destination field (at the location specified by DI). **DMVO** increments DI by one, and sets T to 1 if the source digit is nonzero.

The translation of the source digit is as follows:

If the digit is zero and

S is 0, the translation is equal to *p0*.

S is 1, the translation is equal to *p1*.

If the digit is nonzero and

S is 0, the translation is equal to the sum of the digit and *p2*.

S is 1, the translation is equal to the sum of the digit and *p3*.

Arguments

p0,p1,p2,p3	8-bit characters

Registers, Flags, and Stacks

DI	DI + 1
Overflow	Unaffected
P	P + 5

S	Unchanged
SI	The **EDIT** and **WEDIT** instruction definitions explain how SI is affected by the digit transfer.
T	Set to 1 if source digit nonzero; otherwise unchanged.

Related Instructions

DSSO, DSSZ,	Set S and T flags
DSTO, DSTZ	

Exceptions

DMVO initiates a commercial fault if the character is not valid for the specified data type.

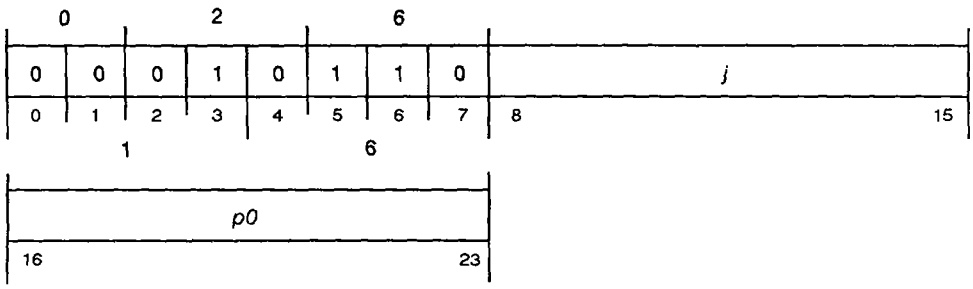
Example

```
;Move one overpunched digit from the source integer to the
;destination field. A positive zero will be represented by a "+"
;(53₈), and a negative zero will be represented by a "-"
;(55₈). Any other digit will be overpunched with the source sign.
;
DMVO   53,55,100,111              ;All values are octal.
```

Move Numeric with Zero Suppression DMVS

Edit Subopcode

DMVS *j,p0*



Function: SI + 1 --> SI
j(SI) --> (DI)
SI + j --> SI
DI + j --> DI
If T = 0, p0[char] --> 0s & spaces
? --> T

Parameters: None

DMVS moves *j* digits from the source field to the destination field. Each digit is handled as follows:

If T is 1, **DMVS** moves the digit from the source to the destination.

If T is 0, **DMVS** replaces all zeros and spaces with *p0*. When the first nonzero digit is encountered, **DMVS** sets T to 1.

DMVS increments DI by *j* and SI by the smaller value of either *j* or the remaining number of characters to move.

Arguments

j	8-bit value
p0	8-bit character

Registers, Flags, and Stacks

DI	Before execution, contains byte pointer to first byte in destination field. After execution, DI + *j*.
Overflow	Unaffected
P	P + 3
S	Unchanged
SI	The **EDIT** and **WEDIT** instruction definitions explain how SI is affected by each digit transfer.
T	Undefined

Related Instructions

DMVO	Move Digit with Overpunch

Exceptions

DMVS initiates a commercial fault if any of the digits processed is not a numeric (0-9) or a space.

DMVS destroys the data type specifier.

Example

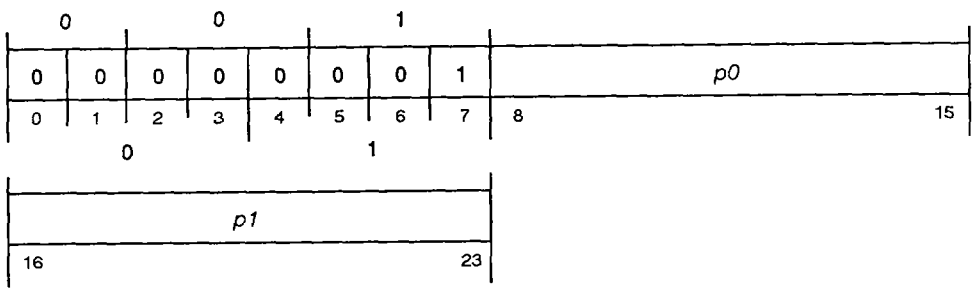
```
DMVS
```

End Float

DNDF

Edit Subopcode

DNDF *p0,p1*



Function:
If T = 1, DI -> DI
If T = 0
1 -> T
If S = 0, *p0*[char] -> (DI)
If S = 1, *p1*[char] -> (DI)
DI + 1 -> DI

Parameters: None

DNDF places a character in the destination field according to the state of the T and S flags.

If T is 1, **DNDF** places nothing in the destination field and leaves T and DI unchanged.

If T is 0 and

Sis 0, the instruction places *p0* in the destination field at the position specified by DI.

Sis 1, the instruction places *p1* in the destination field at the position specified by DI.

Arguments

p0,p1	8-bit characters

Registers, Flags, and Stacks

DI	DI (if T = 1) DI + 1 (if T = 0)
Overflow	Unaffected
P	P + 3
S	Unchanged
SI	Unused
T	Set to 1

Related Instructions

DMVF	Move Float

Exceptions

None

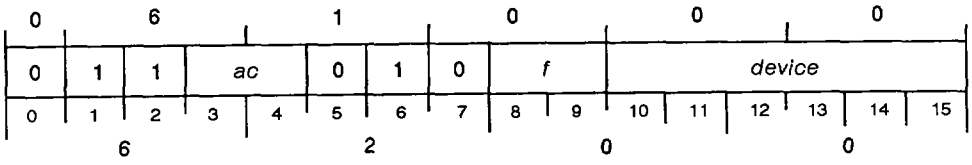
Example

```
;If T is still zero, move the sign of the source integer into the
;destination field. A positive source will be represented by a "+"
;(53₈), and a negative source will be represented by a "-"
;(55₈).
;
DNDF   53,55          ;Values are in octal.
```

Data Out A Buffer — DOA

ECLIPSE Instruction

DOA[*f*] *ac*,*device*



Function: *ac* -> *device* (A buffer)
[*f*] -> BUSY, DONE flags

Parameters: None

DOA transfers data from the specified accumulator to the A buffer of specified I/O device on the default I/O channel (IOC). After the data transfer, the BUSY and DONE flags are set according to the function specified by *f*.

Arguments

f — Specify from **S**, **C**, and **P** for desired I/O device flag control as follows:

f	BUSY	DONE
(option omitted)	No effect	No effect
S	Set to a 1	Set to a 0
C	Set to a 0	Set to a 0
P	Pulses a special I/O bus control line	

ac(16–31) — Before execution, contains data to be sent to A buffer of specified I/O device. Number of data bits moved depends upon size of buffer and mode of operation of device.

After execution, contents unchanged.

device — Specify either mnemonic or device code for desired I/O device.

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, DIB, DIC Transfer data from buffer of I/O device to an accumulator.

DOB, DOC Transfer data from an accumulator to the buffer of an I/O device.

Exceptions

None

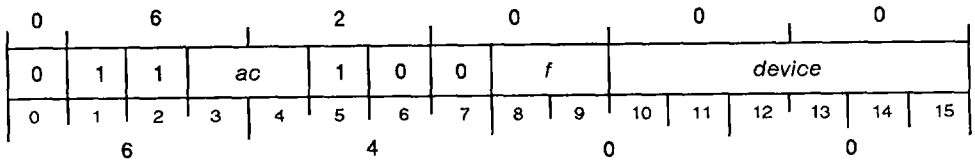
Example

```
DOAP   2,27          ;Move the contents of AC2 into the A buffer
                     ;of device 27 on the default IOC, and send
                     ;the Pulse command to the device.
```

Data out B Buffer **DOB**

ECLIPSE Instruction

DOB[*f*] *ac,device*



Function: *ac* -> *device* (B buffer)
[*f*] -> BUSY,DONE flags

Parameters: None

DOB transfers data from the specified accumulator to the B buffer of specified I/O device on the default I/O channel (IOC). After the data transfer, the BUSY and DONE flags are set according to the function specified by *f*.

Arguments

f Specify from **S**, **C**, and **P** for desired I/O device flag control as follows:

f	BUSY	DONE
(option omitted)	No effect	No effect
S	Set to a 1	Set to a 0
C	Set to a 0	Set to a 0
P	Pulses a special I/O bus control line	

ac(16–31) Before execution, contains data to be sent to B buffer of specified I/O device. Number of data bits moved depends upon size of buffer and mode of operation of device.

After execution, contents unchanged.

device Specify either mnemonic or device code for desired I/O device.

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, DIB, DIC Transfer data from buffer of I/O device to an accumulator.

DOA, DOC Transfer data from an accumulator to the buffer of an I/O device.

Exceptions

None

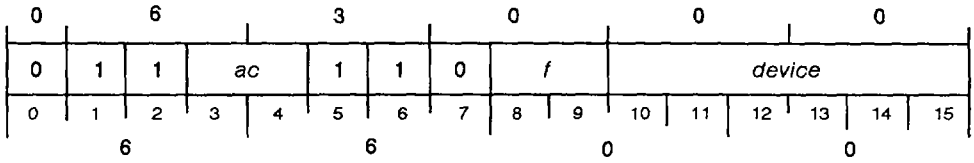
Example

```
DOBC   3,27        ;Move the contents of AC3 into the B buffer
                   ;of device 27 on the default IOC, and clear
                   ;the device's BUSY and DONE flags.
```

Data Out C Buffer DOC

ECLIPSE Instruction

DOC[*f*] *ac,device*



Function: *ac* -> *device* (C buffer)
[*f*] -> BUSY,DONE flags

Parameters: None

DOC transfers data from the specified accumulator to the C buffer of specified I/O device on the default I/O channel (IOC). After the data transfer, the BUSY and DONE flags are set according to the function specified by *f*.

Arguments

f Specify from **S**, **C**, and **P** for desired I/O device flag control as follows:

f	BUSY	DONE
(option omitted)	No effect	No effect
S	Set to a 1	Set to a 0
C	Set to a 0	Set to a 0
P	Pulses a special I/O bus control line	

ac(16–31) Before execution, contains data to be sent to C buffer of specified I/O device. Number of data bits moved depends upon size of buffer and mode of operation of device.

After execution, contents unchanged.

device Specify either mnemonic or device code for desired I/O device.

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, DIB, DIC Transfer data from buffer of I/O device to an accumulator.

DOA, DOB Transfer data from an accumulator to the buffer of an I/O device.

Exceptions

The assembler reserves the **DOC CPU (HALT)** instruction for stopping the processor.

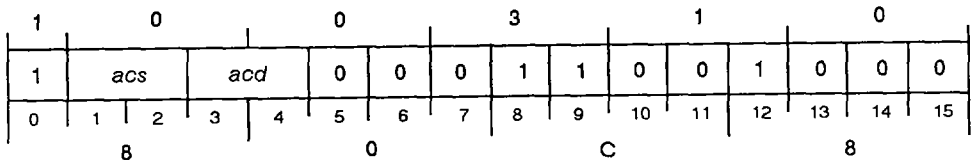
Example

```
DOC    2,27          ;Move the contents of AC2 into the C buffer
                     ;of device 27 on the default IOC, and do not
                     ;modify the device's BUSY and DONE flags.
```

Decimal Subtract DSB

ECLIPSE Instruction

DSB *acs,acd*



Function: $\overline{acd}$[bcd] - *acs*[bcd] - CRY -> *acd*

$\overline{\text{Decimal borrow}}$ -> CRY

Parameters: None

NOTE: If CRY = 0, result is −; if CRY = 1, result is +

DSB subtracts the decimal digit contained in *acs* from the decimal digit contained in *acd*, and then subtracts the complement of CARRY from this result. **DSB** places the decimal unit position of the final result in *acd* and the complement of the decimal borrow in CARRY.

For example, if the final result is negative, the instruction indicates a borrow and sets CARRY to 0. If the final result is positive, the instruction indicates no borrow and sets CARRY to 1.

Argument

acs(28–31) Before execution, contains an unsigned 4-bit binary-coded decimal (BCD) digit.

After execution, contents unchanged unless *acs* and *acd* are same accumulator.

acd(28–31) Before execution, contains an unsigned 4-bit binary-coded decimal (BCD) digit.

After execution, contains decimal unit position of result.

Registers, Flags, and Stacks

AC0–AC3 Can be individually specified for *acs* and *acd*; otherwise not used.

CARRY Contains decimal borrow.

If 0, result negative, indicates borrow.

If 1, result positive, indicates no borrow.

Overflow 0

PC PC + 1

PSR Unchanged

Stack	Unchanged

Related Instructions

DAD	Decimal Add

Exceptions

No validation of the input digits is performed. Therefore, if bits 28–31 of either *acs* or *acd* contain a number greater than 9_{10}, the results will be unpredictable.

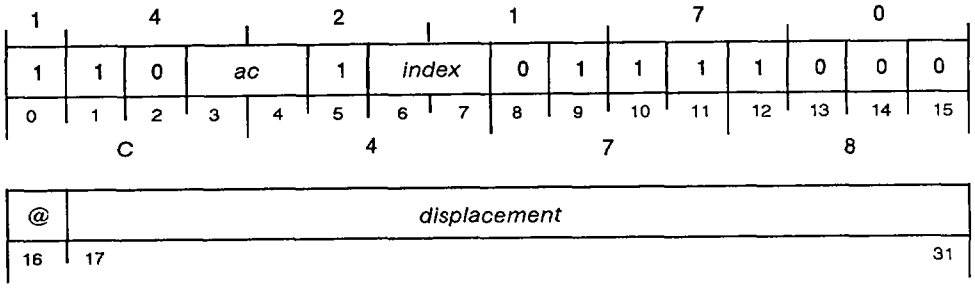
Example

```
DSB 3,2          ;Assume that AC2 contains 9, AC3 contains 7,
          ....   ;and CARRY contains 0. After this instruction
                 ;is executed, AC3 remains the same, AC2
                 ;contains 1, and CARRY is set to 1, indicating
                 ;no borrow.
```

Dispatch DSPA

ECLIPSE Instruction

DSPA *ac,*[@]*displacement*[,*index*]



Function: If L < = *ac* < = H then
If (E + #-L) ≠ −1 then (E + #-L) -> PC
Else (DSPA) + 2 -> PC
0 -> OVF

Parameters: *ac* = # -> unchanged
E = (dispatch table) -> unchanged
E-2 = L [2#] -> unchanged
E-1 = H [2#] -> unchanged
E + H-L = Last table entry -> unchanged
CRY = x -> unchanged

DSPA conditionally transfers program control to an address selected from a memory (dispatch) table (see Figure 11-1). Starting location of the table is specified by the effective address resolved from the instruction arguments. Displacement within the table is defined by a signed 16-bit integer stored in the specified accumulator.

Before being used to access the table, the displacement value is compared against two limit values, an upper and a lower, that define the address range of the table. The limit values are stored at the two locations immediately preceding the table.

If the displacement number is not outside the limit values, the new control address is fetched from the table; checked for validity; and, if valid, placed in the PC.

The fetch address is derived as follows:

Fetch address = effective address − lower limit + displacement number

Note that all values within the table, including the limits, are treated as signed 16-bit integers. Unused locations should be set to −1.

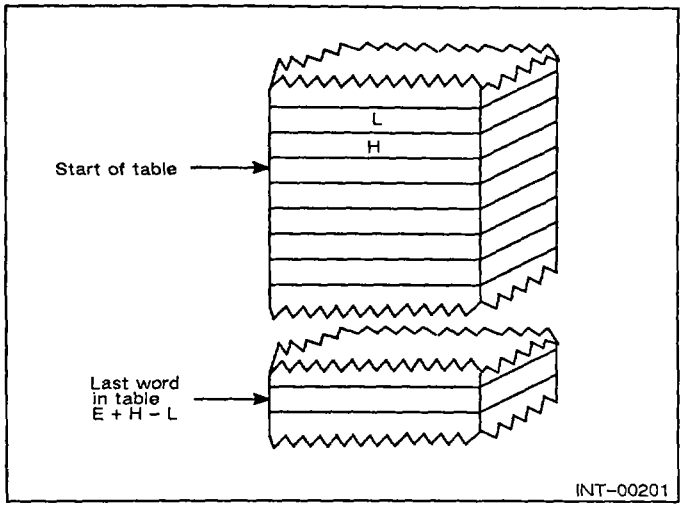


Figure 11-1 *DSPA dispatch table structure*

Arguments

ac(16-31) Before execution, contains signed 16-bit integer; used as the displacement number for the dispatch table.

After execution, contents unchanged.

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.	
CARRY	Unchanged	
Overflow	0	
PC	Fetched word PC + 2	(if valid effective address) (otherwise)
PSR	Unchanged	
Stack	Unchanged	

Related Instructions

LDSP Dispatch (Long Displacement)

Exceptions

In the valid address check, if the fetched number is a -1 (177777_8), the instruction terminates and processing continues with the next sequential 16-bit word.

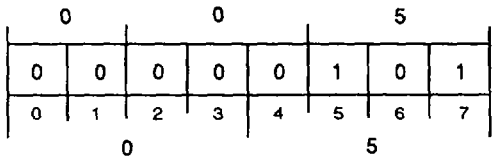
Example

```
    DSPA   2,TABLE      ;
    WBR    OUT_LMT      ;Dispatch value was out of the limits
                        ;or dispatch table entry was -1.
    .
    .
    .
    .WORD  13.          ;Lower limit.
    .WORD  16.          ;Upper limit.
TABLE:  VAL_13          ;Routine to handle 13. in AC2.
        VAL_14          ;Routine to handle 14. in AC2.
        -1              ;AC2 = 15. is not handled.
        VAL_16          ;Routine to handle 16. in AC2.
```

Set S to One DSSO

Edit Subopcode

DSSO



Function: 1 -> S

Parameters: None

DSSO sets the Sign flag (S) to 1.

Arguments

None

Registers, Flags, and Stacks

DI	Unused
Overflow	Unaffected
P	P + 1
S	Set to 1
SI	Unused

Related Instructions

DSSZ	Set S to Zero

Exceptions

None

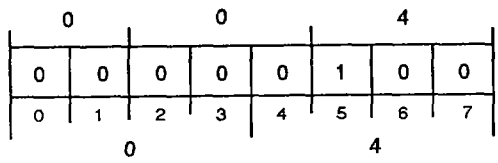
Example

```
DSSO        ;Sets Sign flag to one.
```

Set S to Zero — DSSZ

Edit Subopcode

DSSZ



Function: 0 -> S

Parameters: None

DSSZ sets the Sign flag (S) to 0.

Arguments

None

Registers, Flags, and Stacks

DI	Unused
Overflow	Unaffected
P	P + 1
S	Set to 0
SI	Unused

Related Instructions

DSSO	Set S to One

Exceptions

None

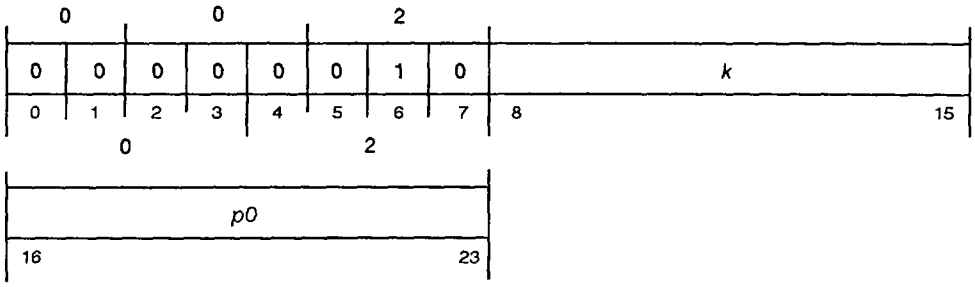
Example

```
DSSZ        ;Sets Sign flag to zero.
```

Store In Stack DSTK

Edit Subopcode

DSTK *k,p0*



Function: *p0*[char] -> (*k*)[2#]

Parameters: None

DSTK stores the byte *p0* into the stack location specified by *k*.

For **EDIT**, **DSTK** stores *p0* in bits 8–15 of a word in the narrow stack, setting bits 0–7 of the stack word to 0.

For **WEDIT**, **DSTK** stores *p0* in bits 24–31 of a double word in the wide stack, setting bits 0–23 of the stack double word to 0.

Arguments

k — Signed 8-bit integer

For **EDIT**:
If negative, stack word at: narrow stack pointer + 1 + *k*
If positive, stack word at: narrow frame pointer + 1 + *k*

For **WEDIT**:
If negative, stack double word at: WSP + 2 + 2***k*
If positive, stack double word at: WFP + 2 + 2***k*

p0 — 8-bit character

Registers, Flags, and Stacks

DI — Unused

Overflow — Unaffected

P — P + 3

SI — Unused

Stack — After execution,

For **EDIT**:
stack word(0–7) = 0
stack word(8–15) = *p0*

For **WEDIT**:
stack double word(0–23) = 0
stack double word(24–31) = *p0*

Related Instructions

DDTK	Decrement and Jump if Nonzero

Exceptions

None

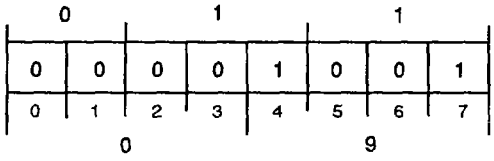
Example

```
WEDIT               ;Enter Edit subprogram.
...
DDTK  2,T           ;Stores "T" into double word in stack at
                    ;address wide frame pointer + 6.
```

Set T to One DSTO

Edit Subopcode

DSTO



Function: 1 -> T

Parameters: None

DSTO sets the significance Trigger (T) to 1.

Arguments

None

Registers, Flags, and Stacks

DI	Unused
Overflow	Unaffected
P	P + 1
SI	Unused
T	Set to 1
Stack	Unchanged

Related Instructions

DSTZ	Set T to Zero

Exceptions

None

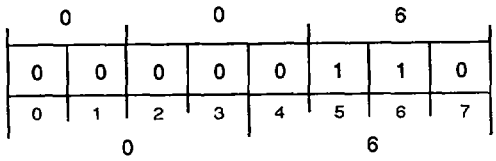
Example

```
DSTO      ;Sets significance Trigger to one.
```

Set T to Zero DSTZ

Edit Subopcode

DSTZ



Function: 0 -> T

Parameters: None

DSTZ sets the significance Trigger (T) to 0.

Arguments

None

Registers, Flags, and Stacks

DI	Unused
Overflow	Unaffected
P	P + 1
SI	Unused
T	Set to 0
Stack	Unchanged

Related Instructions

DSTO	Set T to One

Exceptions

None

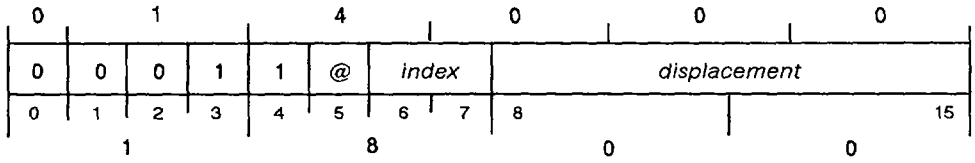
Example

```
DSTZ        ;Sets significance Trigger to zero.
```

Decrement and Skip if Zero DSZ

ECLIPSE Instruction

DSZ [@]*displacement*[,*index*]
(result ≠ 0 return)
(result = 0 return)



Function: (E) - 1 -> (E)
If resulting (E) = 0 then skip

Parameters: None

DSZ decrements by 1 an unsigned 16-bit integer in a specified memory location, writes the result back into the location, and skips the next sequential 16-bit word if the result is 0. This instruction is indivisible.

Arguments

[@]*displacement*[,*index*]

Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1 (result not 0) PC + 2 (result is 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

EDSZ, XNDSZ, XWDSZ, LNDSZ, LWDSZ

Decrement the contents of memory and skip if result equals zero.

Exceptions

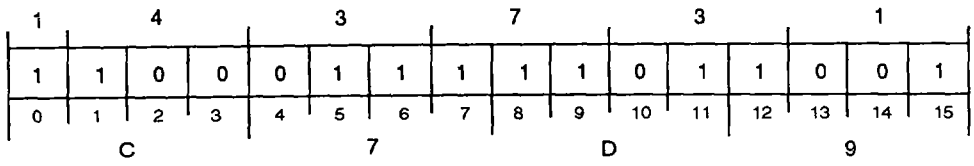
None

Example

```
    LDA    0,FIVE        ;Get a constant 5.
    STA    0,COUNTER     ;Initialize the loop counter.
LOOP:      . . .         ;Beginning of loop....
           DSZ COUNTER   ;Decrement counter and skip if zero.
           JMP LOOP      ;We're not done yet.
           . . .         ;We did the loop 5 times.....
FIVE:      .WORD 5       ;Constant 5.
COUNTER:   .WORD 0       ;Counter variable.
```

Decrement the Word Addressed by WSP and Skip if Zero — DSZTS

DSZTS
(decrement ≠ 0 return)
(decrement = 0 return)



Function: (wsp) -1 -> (wsp)
If resulting (wsp) = 0 then skip

Parameters: None

DSZTS decrements the unsigned 32-bit integer addressed by the wide stack pointer and skips the next 16-bit word if the decremented value is zero.

The operation performed by **DSZTS** is not indivisible.

Arguments

None

Registers, Flags, and Stacks

AC0-AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1 (addressed word ≠ 0) PC + 2 (addressed word = 0)
PSR	Unchanged
Stack	WSP Unchanged

Related Instructions

ISZTS Increment the Word Addressed by WSP and Skip if Zero

Exceptions

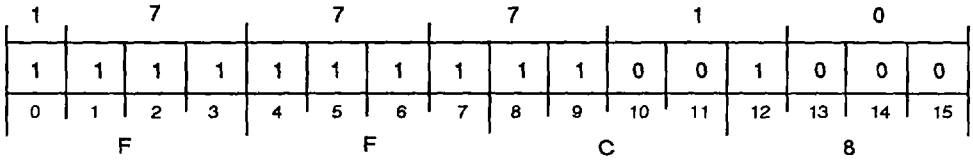
None

Example

```
    NLDAI  5,1          ;Get a 5 in AC1.
    WPSH   1,1          ;Push the 5 onto the stack.
LOOP:. . .              ;Beginning of loop....
    DSZTS               ;Decrement the top of the stack.
    WBR    LOOP         ;We're not done yet.
    WPOP   0,0          ;We did the loop 5 times. Pop the
                        ;stack back to its original state.
```

Load CPU Identification ECLID

ECLID



Function: CPU id -> AC0(model number [bits 0-15],
microcode revision [bits 16-23],
memory size [bits 24-31])

Parameters: None

ECLID loads CPU identification information into AC0.

Arguments

None

Registers, Flags, and Stacks

AC0 Before execution, contains CPU identification information as follows:

Bits	Contents
0-15	model number
16-23	microcode revision
24-31	memory size

AC1-AC3 Unused

CARRY Unchanged

Overflow 0

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

NCLID, LCPID Return CPU identification information.

Exceptions

None

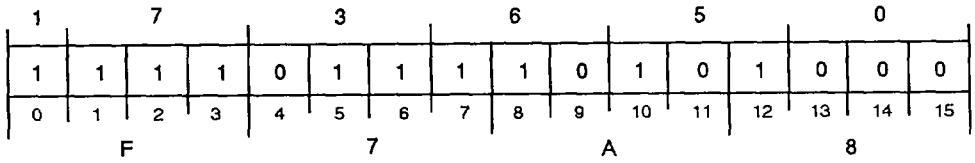
Example

```
    ECLID               ;Load the CPU identification into AC0.
    XWSTA     0,CPUID   ;Store the CPU id in memory.
    . . .
CPUID:      .DWORD 0 ;Variable for CPU id.
```

Edit EDIT

ECLIPSE Instruction

EDIT



Function: Enter edit subprogram

Parameters:
AC0 = byte pointer (1st subopcode) --> P
AC1 = data-type indicator --> ?
AC2 = byte pointer (destination) --> DI
AC3 = byte pointer (source) --> SI
T = 0 --> ?S = SI sign (0 = +, 1 = -) --> ?
SI = AC3 --> last byte pointer + 1
DI = AC2 --> last byte pointer + 1
P = AC0 --> last byte pointer + 1
CRY = x --> T

NOTE:
For subopcodes: *j* = # characters
If *j*(high-order bit) = 1, word at (SP + 1 + *j*)

During execution, a subprogram modifies DI, P, S, SI, and T; can test S and T flags.

EDIT provides entry and control for an edit subprogram. The subprogram converts a decimal number from either packed or unpacked form to a string of bytes. This subprogram can perform many different operations on the number and its destination field, including leading zero suppression, leading or trailing signs, floating fill characters, punctuation control, and insertion of text into the destination field. Subprogram instructions also perform operations on alphanumeric data.

EDIT uses the contents of the four accumulators to provide initial parameters for the subprogram, and maintains two flags and three indicators or pointers, that can be tested and modified by the subprogram. The flags are the significance Trigger (T) and the Sign flags (S); the three pointers are the Source Indicator (SI), the Destination Indicator (DI), and the opcode Pointer (P).

The significance Trigger flag is set to 1 when the first nonzero digit is processed; the Sign flag is set to reflect the sign of the source integer being processed. Some subprogram instructions may explicitly set or clear the S and T flags.

The three pointers are 16-bit byte pointers to the current byte in each respective area. These fields may overlap in any way. The instructions, however, process characters one at a time, so unusual side effects may be produced by certain types of overlap.

The subprogram is made up of eight-bit opcodes (subopcodes) followed by one or more eight-bit operands. P, a byte pointer, serves as the program counter for the Edit subprogram. The subprogram proceeds sequentially until a branching operation occurs -- much the same way programs are processed. Unless instructed to do otherwise, the Edit instruction updates P after each operation to point to the next sequential subopcode. The instruction continues to process eight-bit opcodes until directed to stop by the **DEND** subopcode.

The effective addresses generated by **EDIT** are confined to the first 64 Kbytes of the current segment.

In the description of some of the Edit subopcodes, the symbol *j* denotes how many characters a certain operation should process. When the high order bit of *j* is 1, *j* has a different meaning: *j* is interpreted as a signed eight-bit integer, and the number of characters to process is equal to the value of the word at the address *stack pointer + 1 + j*.

The Edit operations that process numeric data (**DMVF, DMVN, DMVO,** and **DMVS**) use the following algorithm to access each source digit:

1. If SI has ever moved outside the source area, a zero will be used for the source digit, and SI will not be affected. Note that zeros will be supplied for all future source digits, even if SI is moved back inside the source area.
2. If the source integer is data type 3, and SI currently points to the sign of the integer, SI will be incremented to skip over the sign.
3. The digit to which SI currently points is checked for validity, and the binary coded decimal (BCD) value of the digit is used. SI is incremented to point to the next digit in the source integer.
4. If the source integer is data type 2, and the last digit has been read, SI is incremented beyond the trailing sign byte.

Arguments

None

Registers, Flags, and Stacks

AC0(16-31) Initially contains byte pointer to first subopcode of the Edit subprogram.

After successful execution of subprogram, contains P, which points to byte following **DEND** subopcode.

AC1(16-31) Initially contains data-type indicator describing source integer being processed. The scale factor portion is not used. For further information, refer to "Decimal and Byte Operations" section of "Fixed-Point Computing" chapter.

After successful execution of subprogram, contents undefined.

AC2(16-31) Initially contains byte pointer to first byte of destination byte field (DI).

After successful execution of subprogram, contains byte pointer (DI) to next byte in destination field.

AC3(16-31) Initially contains byte pointer to first byte of source integer field (SI).

After successful execution of subprogram, contains byte pointer (SI) to next source byte.

CARRY After execution, contains T.

DI Initially, contains AC2(16-31).

After execution of subopcode, may be modified.

Overflow 0

P	Initially, contains AC0(16-31). Unless instructed otherwise, updated after each subopcode to point to next sequential subopcode.
PSR	If IRES contains 1, instruction assumes restart from interrupt. Do not set IRES under any other circumstances.
PC	PC + 1 (After successful completion of subprogram)
S	Initially, reflects sign of source integer being processed. 0 = positive; 1 = negative. May be explicitly set or cleared by some subopcodes.
SI	Initially, contains AC3(16-31). After execution of subopcode, may be modified.
Stack	Initially, narrow stack should have at least 16 words available for use by **EDIT**.
T	Initially, set to 0. Set to 1 when first nonzero digit processed. May be explicitly set or cleared by some subopcodes.

Related Instructions

LEFB	Load Effective Byte address
Edit subopcodes	Use these instructions to manipulate and process data as a subprogram.
WEDIT	Wide Edit

Exceptions

If the sign of the source integer is invalid, a commercial fault is inititated, and the **EDIT** terminates.

If the data type indicator in AC1 specifies that the source integer is data type 6 or 7, a commercial fault is initiated, and the instruction terminates.

See also the exceptions for the individual subopcodes.

If **EDIT** is interrupted, restart information is placed on the narrow stack, PSR(IRES) is set to 1, and a value of 177777_8 is placed in AC0.

EDIT considers the subprogram as data and does not check for execute protection.

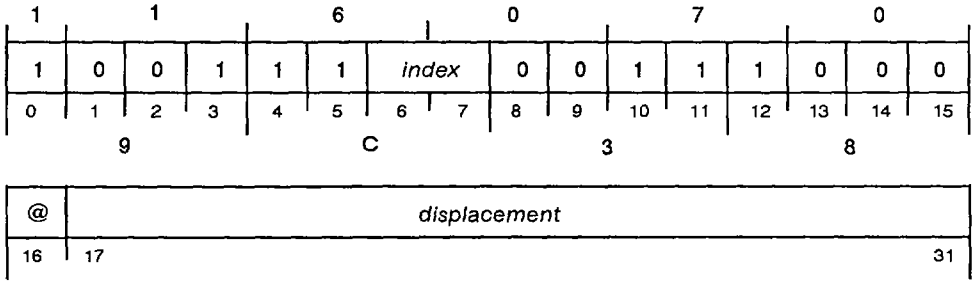
Example

Refer to the programming example shown for WEDIT (the instructions are similar in usage).

Extended Decrement and Skip if Zero EDSZ

ECLIPSE Instruction

EDSZ [@]*displacement*[,*index*]
(result ≠ 0 return)
(result = 0 return)



Function: (E) - 1 -> (E)
If resulting (E) = 0 then skip

Parameters: None

EDSZ decrements the unsigned 16-bit integer in the specified memory location by 1, writes the result back into the location, and skips the next sequential 16-bit word if the result is zero. This instruction is indivisible.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1 (result ≠ 0) PC + 2 (result = 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

DSZ, XNDSZ, XWDSZ, LNDSZ, LWDSZ Decrement memory location by one and skip if result is zero.

Exceptions

None

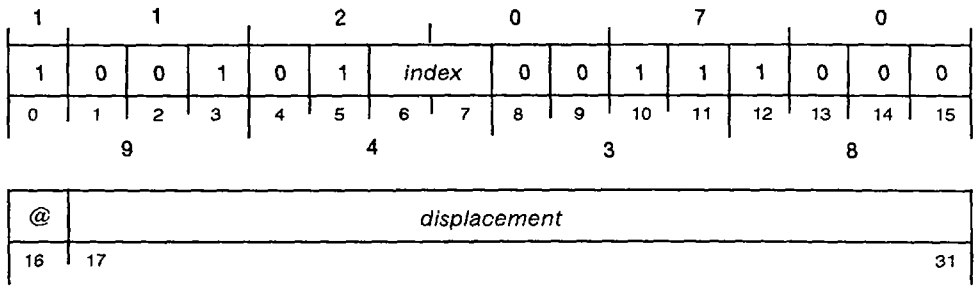
Example

```
    ELDA   0,FIVE        ;Get a constant 5.
    ESTA   0,COUNTER     ;Initialize the loop counter.
LOOP:. . .               ;Beginning of loop.
    .
    .
    .
    EDSZ   COUNTER       ;Decrement counter and skip if zero.
    JMP    LOOP          ;We're not done yet.
    .
    .
    .
                         ;We did the loop 5 times.
    .
    .
    .
FIVE:     .WORD 5        ;Constant 5.
COUNTER:  .WORD 0        ;Counter variable.
```

Extended Increment and Skip if Zero EISZ

ECLIPSE Instruction

EISZ [@]*displacement*[,*index*]
(result ≠ 0 return)
(result = 0 return)



Function: (E) + 1 -> (E)
If resulting (E) = 0 then skip

Parameters: None

EISZ increments the unsigned 16-bit integer in the specified memory location by 1, writes the result back into the location, and skips the next sequential 16-bit word if the result is zero. This instruction is indivisible.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1 (result ≠ 0) PC + 2 (result = 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

ISZ, XNISZ, XWISZ, LNISZ, LWISZ	Increment memory location by one and skip if result is zero.

Exceptions

None

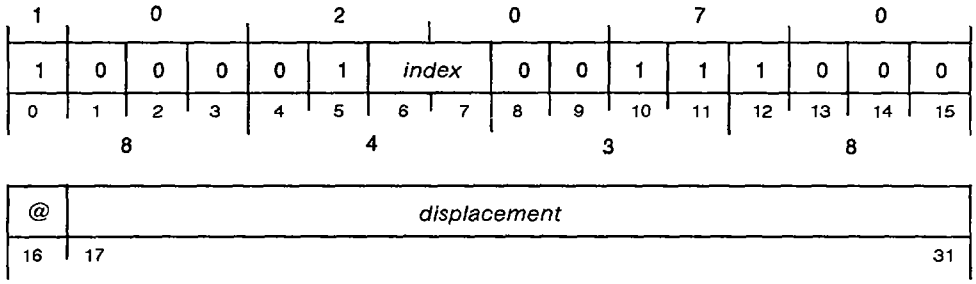
Example

```
    ELDA   0,NFIVE      ;Get a constant -5.
    ESTA   0,COUNTER    ;Initialize the loop counter.
LOOP:. . .              ;Beginning of loop.
    .
    .
    .
    EISZ   COUNTER      ;Increment counter and skip if zero.
    JMP    LOOP         ;We're not done yet.
    . . .               ;We did the loop 5 times.
    .
    .
    .
NFIVE:    .WORD -5      ;Constant -5.
COUNTER:  .WORD 0       ;Counter variable.
```

Extended Jump EJMP

ECLIPSE Instruction

EJMP [@]*displacement*[,*index*]



Function: E -> PC

Parameters: None

EJMP loads a specified address into the program counter. Program execution continues with the instruction addressed by the updated value of the program counter.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	Effective address resolved from arguments.
PSR	Unchanged
Stack	Unchanged

Related Instructions

JMP, XJMP, LJMP	Load the program counter with an effective address.

Exceptions

None

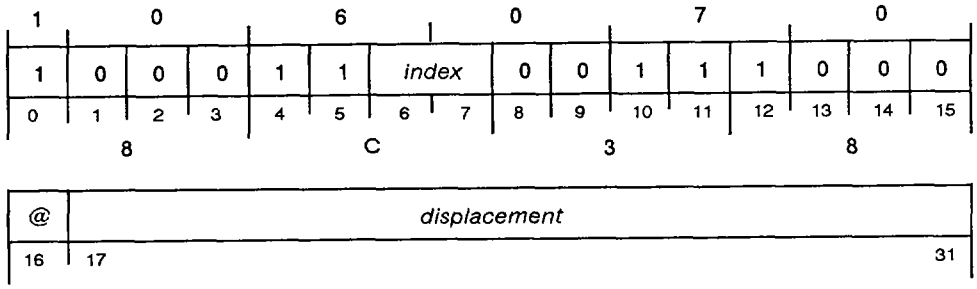
Example

```
    EJMP   FAR_AWAY     ;Jump to a location that requires an
                        ;extended displacement relative to the
                        ;current location.
FAR_AWAY:
          . . .
```

Extended Jump to Subroutine EJSR

ECLIPSE Instruction

EJSR [@]*displacement*[,*index*]



Function: E -> PC
PC + 2 -> AC3

Parameters: None

EJSR increments the PC by 2 (address for next sequential instruction) and stores the value in AC3; it also loads the effective address derived from the arguments into the program counter. Program execution continues with the instruction addressed by the updated value of the program counter.

The effective address is calculated before the incremented value of the program counter is stored.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

AC0–AC2	Unused
AC3	After execution, contains PC(before execution) + 2.
CARRY	Unchanged
Overflow	0
PC	Effective address resolved from arguments.
PSR	Unchanged
Stack	Unchanged

Related Instructions

JSR, XJSR, LJSR	Perform a jump to a subroutine.

Exceptions

None

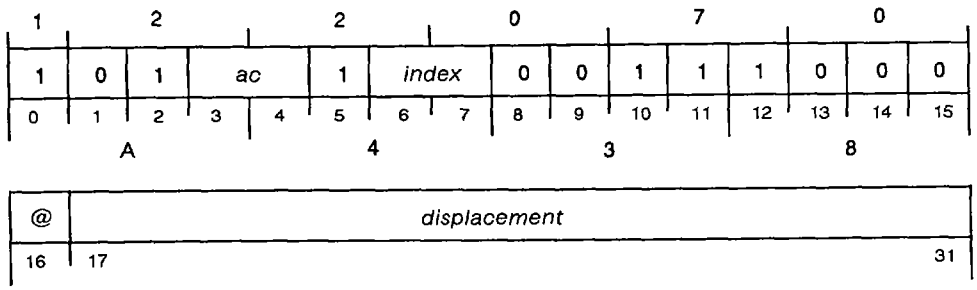
Example

```
    EJSR   SUBROUT      ;Jump to subroutine. Return PC is put in AC3.
    .
    .
    .
SUBROUT:
    .                   ;Do the subroutine, but don't modify
    .                   ;contents of AC3 because it has the
    .                   ;return address.
    JMP    0,3          ;Go back to the caller. ACs are not
                        ;necessarily restored.
```

Extended Load Accumulator ELDA

ECLIPSE Instruction

ELDA *ac*,[@]*displacement*[,*index*]



Function: (E) -> *ac*

Parameters: None

ELDA loads a word from a memory location into a specified accumulator.

Arguments

ac(16–31) After execution contains word from addressed memory location.

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

LDA, XNLDA, XWLDA, LNLDA, LWLDA Load an accumulator with the contents of memory.

Exceptions

None

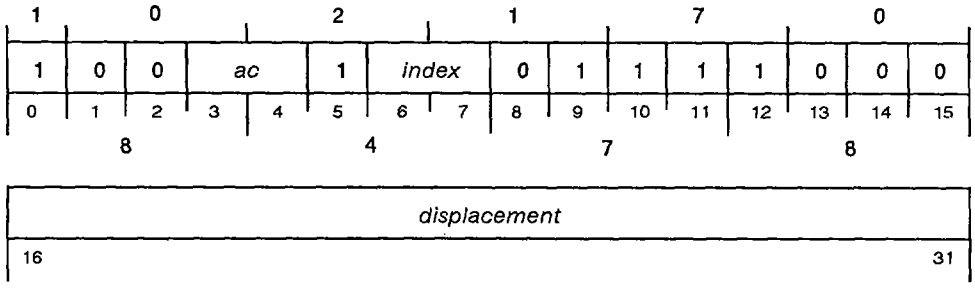
Example

```
    ELDA   1,COUNT      ;Get the counter value into AC1.
    . . .
COUNT:     .WORD 0      ;Counter value.
```

Extended Load Byte ELDB

ECLIPSE Instruction

ELDB *ac,displacement*[,*index*]



Function: (E)byte -> *ac*[bits 24-31, bits 16-23 set to 0]

Parameters: None

ELDB loads a byte from a memory location into a specified accumulator.

ELDB forms a byte pointer from *displacement*[,*index*] in the following way:

Shifts the 16-bit *displacement* field right one bit, producing a 15-bit offset value and a 1-bit byte indicator.

Uses *index* to determine a word address.

Adds the offset value to the word address to give a memory address. The byte indicator designates which byte of the addressed word will be loaded into *ac*.

Arguments

ac(24-31) After execution, contains byte with bits 16-23 set to 0.

displacement[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

LDB, XLDB, LLDB Load an accumulator with a byte from a memory location.

Exceptions

None

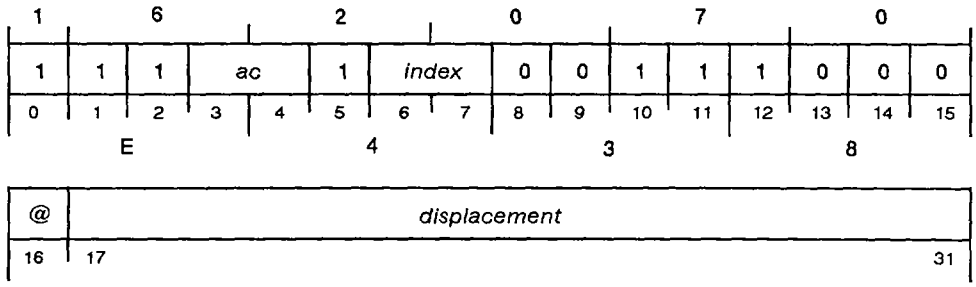
Example

```
    ELDB   2,(BYTE_PAIR*2)+1        ;Load AC2 with the low order byte.
    .                               ;from the word..
    .
    .
BYTE_PAIR:    .WORD 0               ;Location containing a pair of bytes.
```

Extended Load Effective Address ELEF

ECLIPSE Instruction

ELEF *ac*,[@]*displacement*[,*index*]



Function: E -> *ac*

Parameters: None

ELEF places an effective address into specified accumulator.

Arguments

ac After execution, contains effective address, resolved from arguments with bit 0 is set to 0.

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

LEF, XLEF, LLEF	Load an effective address into an accumulator.

Exceptions

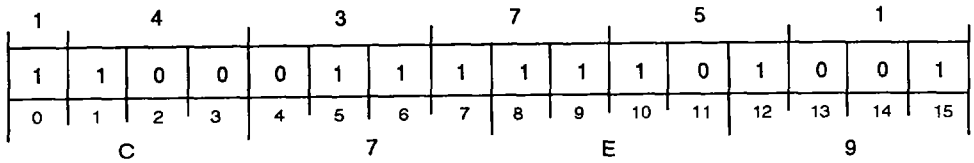
None

Example

```
    ELEF   2,WORD_ARRAY ;Get starting address of array of words.
    ADD    1,2          ;Add the word index from AC1.
    LDA    0,0,2        ;Get the word into AC0.
    . . .
WORD_ARRAY:
           .BLK 16.     ;Array of 16 words.
```

Enqueue Towards the Head ENQH

ENQH
(empty queue return)
(non-empty queue return)



Function: data element --> Q Head

Parameters: AC0 = E(Q descriptor) --> unchanged
AC1 = E(Q element) --> unchanged
AC2 = E(element to add) --> unchanged

NOTE: If AC1 = -1, element is added to queue head.

ENQH enqueues a data element either preceding another data element in the queue or at the head of the queue. The instruction checks the page or pages that contain the current element for valid read and write privileges. If the privileges are valid, **ENQH** checks the queue descriptor. If the descriptor indicates that the queue contains data elements, the instruction first reads all of the links required to complete the enqueuing operation.

The instruction checks all reads and writes of links in data elements and queue descriptors against the current segment. Segment numbers of the link addresses must be greater than or equal to the current segment.

The enqueue operation is not interruptible. The entire operation completes before any interrupts are enabled. **ENQH** is indivisible with respect to **ENQT**, **DEQUE**, or another **ENQH** instruction.

ENQH requires up to nine pages to be resident. (The worst case occurs when an element is inserted between two other elements, and when all elements and the queue header have one of their affected links on a page boundary. Then eight pages are required, in addition to page zero of the current segment.) The instruction first reads all links required to complete the enqueing operation. If a page fault occurs, **ENQH** starts again reading all the links until no page fault occurs. When all required pages are resident, the instruction attempts to enqueue the data element.

Arguments

None

Registers, Flags, and Stacks

AC0 Before execution, contains effective address of queue descriptor.

If new data element becomes head of queue, **ENQH** updates queue descriptor.

After execution, contents unchanged.

AC1	Before execution, contains effective address of data element in front of which new element is added. If AC1 contains −1, new element is added to head of queue. Contents ignored when enqueuing to an empty queue. Instruction enqueues an element by updating the various links of the involved elements: The element to be enqueued -- the forward link contains the address of the preceeding element in the queue; the backward link contains the address of the following element in the queue. The following element in the queue -- the backward link contains the address of the new element. The preceding element in the queue -- the forward link contains the address of the new element. After execution, contents unchanged.
AC2	Before execution, contains effective address of data element to be added to queue. After execution, contents unchanged.
AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1 (queue empty before execution) PC + 2 (queue not empty before execution)
PSR	Unchanged
Stack	Unchanged

Related Instructions

DEQUE	Dequeue a Queue Data Element
ENQT	Enqueue Towards the Tail
LLEF, XLEF	Use these instructions to place effective addresses into the accumulators.
Queue search	Use these instructions to locate an element in a queue.

Exceptions

If the page or pages that contain the current data element have invalid read and write access privileges, the appropriate protection fault occurs, and the queue remains unchanged.

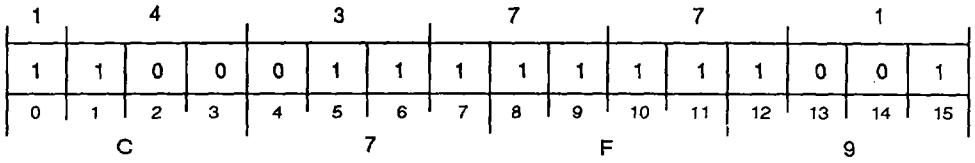
If the ring numbers of the link addresses are less than the current ring, the appropriate protection fault occurs.

Example

```
XLEF 0,QUEDES       ;Load into AC0 effective address of queue desc
WLDAI -1,1          ;Load AC1 with -1
XLEF 2,NEWDAT       ;Load into AC2 effective address of new
                    ;data element
ENQH                ;Enqueue new data element (NEWDAT) at head of
                    ;queue
WBR                 ;
.                   ;
.
.
```

Enqueue Towards the Tail ENQT

ENQT
(empty queue return)
(non-empty queue return)



Function: data element --> Q Tail

Parameters: AC0 = E(Q descriptor) --> unchanged
AC1 = E(Q element) --> unchanged
AC2 = E(element to add) --> unchanged

NOTE: If AC1 = -1, element is added to queue tail

ENQT enqueues a data element either following another data element in the queue or at the tail of the queue. The instruction checks the page or pages that contain the current element for valid read and write privileges. If the privileges are valid, **ENQT** checks the queue descriptor. If the descriptor indicates that the queue contains data elements, the instruction first reads all of the links required to complete the enqueuing operation.

The instruction checks all reads and writes of links in data elements and queue descriptors against the current segment. Segment numbers of the link addresses must be greater than or equal to the current segment.

The enqueue operation is not interruptible. The entire operation completes before any interrupts are enabled. **ENQT** is indivisible with respect to **ENQH**, **DEQUE**, or another **ENQT** instruction.

ENQT requires up to nine pages to be resident. (The worst case occurs when an element is inserted between two other elements, and when all elements and the queue header have one of their affected links on a page boundary. Then eight pages are required, in addition to page zero of the current segment.) The instruction first reads all links required to complete the enqueing operation. If a page fault occurs, **ENQT** starts again reading all the links until no page fault occurs. When all required pages are resident, the instruction attempts to enqueue the data element.

Arguments

None

Registers, Flags, and Stacks

AC0 Before execution, contains effective address of queue descriptor.

If new data element becomes tail of queue, **ENQT** updates queue descriptor.

After execution, contents unchanged.

AC1 Before execution, contains effective address of data element in back of which new element is added.

If AC1 contains -1, new element is added to tail of queue.

Contents ignored when enqueuing to an empty queue.

Instruction enqueues an element by updating the various links of the involved elements:

The element to be enqueued -- the forward link contains the address of the preceeding element in the queue; the backward link contains the address of the following element in the queue.

The following element in the queue -- the backward link contains the address of the new element.

The preceding element in the queue -- the forward link contains the address of the new element.

After execution, contents unchanged.

AC2 — Before execution, contains effective address of data element to be added to queue.

After execution, contents unchanged.

AC3 — Unused

CARRY — Unchanged

Overflow — 0

PC — PC + 1 (queue empty before execution)
PC + 2 (queue not empty before execution)

PSR — Unchanged

Stack — Unchanged

Related Instructions

DEQUE — Dequeue a Queue Data Element

ENQH — Enqueue Towards the Head

LLEF, XLEF — Use these instructions to place effective addresses into the accumulators.

Queue search — Use these instructions to locate an element in a queue.

Exceptions

If the page or pages that contain the current data element have invalid read and write access privileges, the appropriate protection fault occurs, and the queue remains unchanged.

If the ring numbers of the link addresses are less than the current ring, the appropriate protection fault occurs.

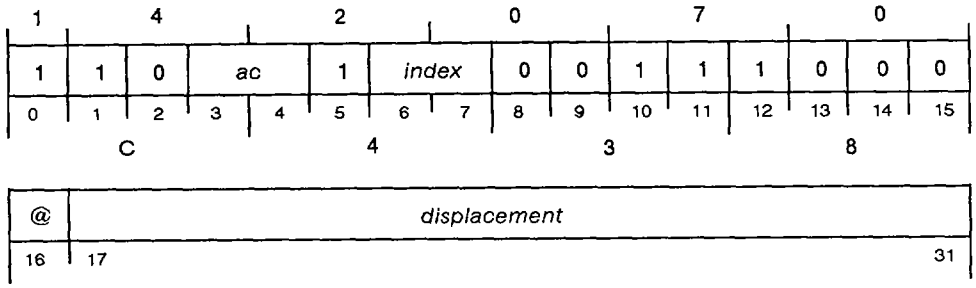
Example

```
    ;Move an element from one queue to the end of another
    ;
    ;It is the responsibility of the calling routine to set any element
    ;"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:    WSSVR       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                    ;Dequeue 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 dequeued.
          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.
```

Extended Store Accumulator ESTA

ECLIPSE Instruction

ESTA *ac*,[@]*displacement*[,*index*]



Function: *ac* -> (E)

Parameters: None

ESTA stores the contents of specified accumulator into the specified memory location.

Arguments

ac(16-31) Contains 16-bit value.

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

STA, XNSTA, XWSTA, LNSTA, LWSTA Store the contents of an accumulator to memory.

Exceptions

None

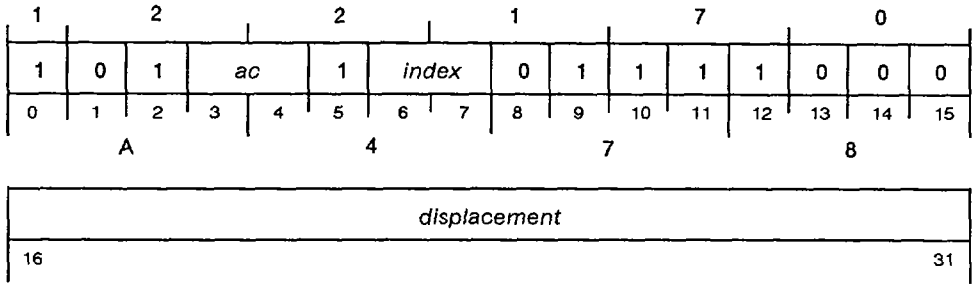
Example

```
    ESTA  1,COUNT      ;Store the counter value from AC1 to memory.
    . . .
COUNT:    .WORD 0      ;Counter value.
```

Extended Store Byte ESTB

ECLIPSE Instruction

ESTB *ac,displacement*[,*index*]



Function: *ac*[right byte] -> (E)byte

Parameters: None

ESTB stores a byte from an accumulator into a memory location.

ESTB forms a byte pointer from *displacement*[,*index*] in the following way:

Shifts the 16-bit *displacement* field right one bit, producing a 15-bit offset value and a 1-bit byte indicator.

Uses *index* to determine a word address.

Adds the offset value to the word address to give a memory address. The byte indicator designates which byte of the addressed word will contain the byte in *ac*.

Arguments

ac(24-31) Before execution, contains byte.

After execution, contents unchanged.

displacement[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

STB, XSTB, LSTB	Store a byte in an accumulator to a memory location.

Exceptions

None

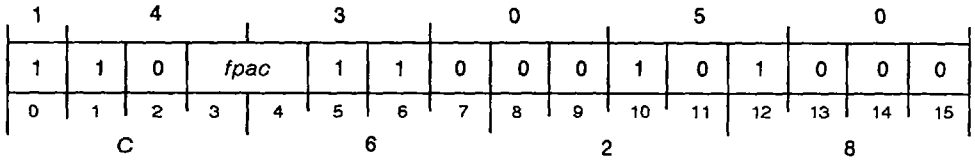
Example

```
    ESTB   2,(BYTE_PAIR*2)+1        ;Store the byte in bits 24-31 of AC2
    .                               ;into the low-order byte of the word
    .                               ;in memory.
    .
BYTE_PAIR:
    .WORD 0                         ;Location containing a pair of bytes.
```

Absolute Value FAB

ECLIPSE Instruction

FAB *fpac*



Function: absolute(*fpac*) -> *fpac*
0 -> FPSR(N)
update -> FPSR(Z)

Parameters: None

FAB sets the sign bit of *fpac* to 0. Then it updates the Z and N flags in the floating-point status register to reflect the new contents of *fpac*.

Arguments

fpac Before execution, contains a floating-point number.

After execution, sign bit set to 0.

Registers, Flags, and Stacks

CARRY Unchanged

FPAC0–FPAC3 Can be individually specified for *fpac*; otherwise unused.

FPSR N flag set to 0; Z flag updated.

PC PC + 1

Stack Unchanged

Related Instructions

FNEG Negate

Exceptions

None

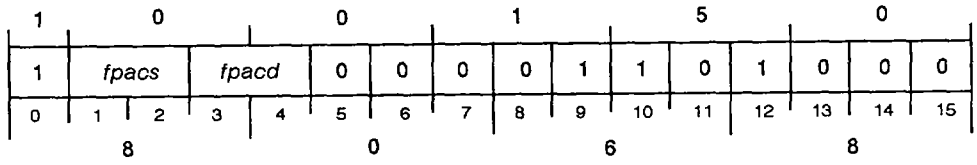
Example

```
LFLDD 0,FLOATX     ;Replace the floating-point number at
FAB   0            ;memory location FLOATX with its
LFSTD 0,FLOATX     ;absolute value.
```

Add Double (FPAC to FPAC) FAD

ECLIPSE Instruction

FAD *fpacs,fpacd*



Function: *fpacs + fpacd -> fpacd*

Parameters: None

FAD adds the floating-point number in *fpacs* to the floating-point number in *fpacd*.

Arguments

fpacs Before execution, contains 64-bit floating-point number.

After execution, contents unchanged unless *fpacs* and *fpacd* are same accumulator.

fpacd Before execution, contains 64-bit floating-point number.

After execution, contains normalized result.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be individually specified as *fpacs* and *fpacd*; otherwise unused.

FPSR Updated Z and N flags.

PC PC + 1

Stack Unchanged

Related Instructions

FAS Add Single (FPAC to FPAC)

Exceptions

If the addition produces an exponent overflow, the processor sets the FPSR(OVF) flag to one and terminates the instruction.

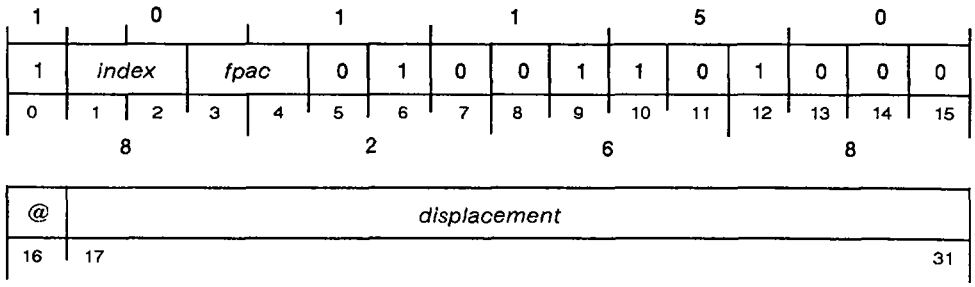
Example

```
FAD 2,0    ;Adds the double-precision number in FPAC2 to
....       ;the double-precision number in FPAC0,
....       ;returning the result to FPAC0.
```

Add Double (Memory to FPAC) FAMD

ECLIPSE Instruction

FAMD *fpac*,[@]*displacement*[,*index*]



Function: (E) + *fpac* -> *fpac*

Parameters: None

FAMD adds a 64-bit floating-point number from a memory location to the 64-bit floating-point number in a floating-point accumulator, placing the normalized result in the *fpac*.

Arguments

fpac Before execution, contains 64-bit floating-point number.

After execution, contains normalized result.

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

FPAC0–FPAC3	Can be individually specified as *fpac*; otherwise unused.
CARRY	Unchanged
PC	PC + 2
FPSR	Z and N flags updated.
Stack	Unchanged

Related Instructions

XFAMD, LFAMD, FAMS, XFAMS, LFAMS Add the contents of a memory location to a floating-point accumulator.

Exceptions

If the addition produces an exponent overflow, the processor sets the FPSR(OVF) flag to one, and terminates the instruction.

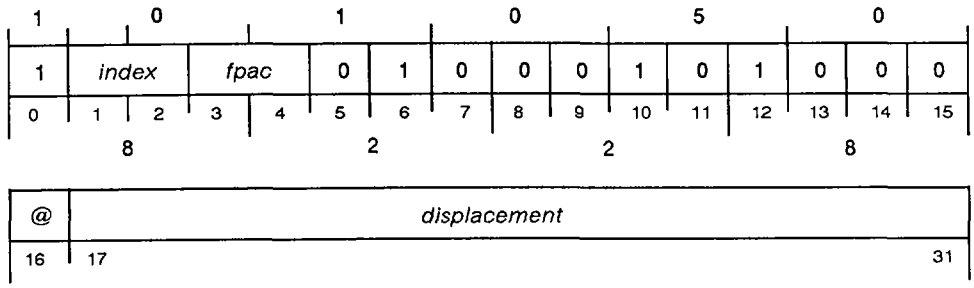
Example

```
FLDD   0,FLOATX     ;Add the two double precision floating-point
FAMD   0,FLOATY     ;numbers in memory locations FLOATX and
FSTD   0,FLOATZ     ;FLOATY, storing the result in memory location
                    ;FLOATZ.
```

Add Single (Memory to FPAC) FAMS

ECLIPSE Instruction

FAMS *fpac*,[@]*displacement*[,*index*]



Function: (E) + *fpac* -> *fpac*

Parameters: None

FAMS adds a 32-bit floating-point number from a memory location to the 32-bit floating-point number in a floating-point accumulator, placing the normalized result into the *fpac*.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contains normalized 32-bit result with bits 32-63 set to 0.

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
CARRY	Unchanged
PC	PC + 2
FPSR	Z and N flags updated.
Stack	Unchanged

Related Instructions

XFAMS, LFAMS, FAMD, XFAMD, LFAMD Add the contents of memory to a floating-point accumulator.

Exceptions

If the addition produces an exponent overflow, the processor sets the FPSR(OVF) flag to one, and terminates the instruction.

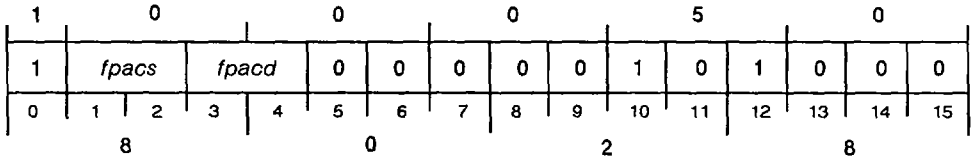
Example

```
FLDS   1,FLOATX     ;Add the two single precision floating-point
FAMS   1,FLOATY     ;numbers in memory locations FLOATX and
FSTS   1,FLOATZ     ;FLOATY, storing the result in memory location
                    ;FLOATZ.
```

Add Single (FPAC to FPAC) FAS

ECLIPSE Instruction

FAS *fpacs,fpacd*



Function: *fpacs* + *fpacd* -> *fpacd*

Parameters: None

FAS adds the 32-bit floating-point number in *fpacs* to the 32-bit floating-point number in *fpacd*, placing the normalized result into *fpacd*.

Arguments

fpacs(0-31) Before execution, contains 32-bit floating-point number.

After execution, contents unchanged unless *fpacs* and *fpacd* are same accumulator.

fpacd(0-31) Before execution, contains 32-bit floating point number.

After execution, contains normalized 32-bit result with bits 32-63 set to 0.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be individually specified as *fpacs* and *fpacd*; otherwise unused.

FPSR Updated Z and N flags

PC PC + 1

Stack Unchanged

Related Instructions

FAD Add Double (FPAC to FPAC)

Exceptions

If the addition produces an exponent overflow, the processor sets the FPSR(OVF) flag to one, and terminates the instruction.

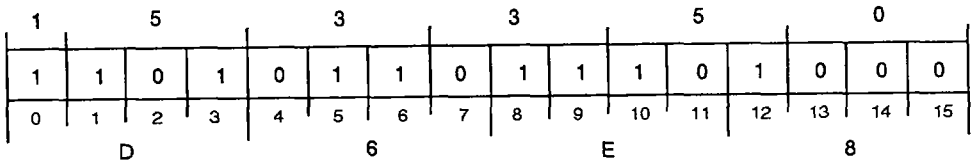
Example

```
FAS 2,0          ;Adds the single-precision number in FPAC2 to
....             ;the single-precision number in FPAC0,
....             ;returning the result to FPAC0.
```

Clear Errors FCLE

ECLIPSE Instruction

FCLE



Function: 0 -> FPSR(0-4, 28-31)

Parameters: None

NOTE: FPSR(FPPC) is undefined.

FCLE sets bits 0–4 and bits 28–31 of the floating-point status register to 0. Since **FCLE** sets the ANY bit of the FPSR to 0, the floating-point program counter is undefined.

Arguments

None

Registers, Flags, and Stacks

CARRY	Unchanged
FPAC0–FPAC3	Unused.
FPSR	Bits 0–4 and bits 28–31 set to 0
PC	PC + 1
Stack	Unchanged

Related Instructions

IORST	I/O Reset

Exceptions

None

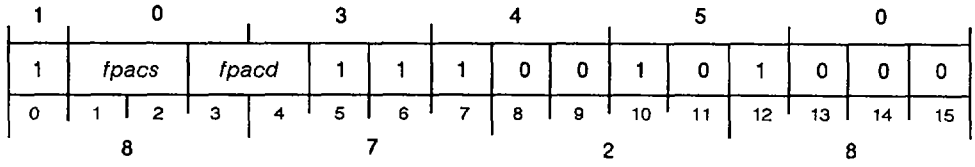
Example

```
FAS   1,2          ;Add the values in FPAC1 and FPAC2, and clear
FCLE               ;out the FPSR, in case there was overflow.
```

Compare Floating Point FCMP

ECLIPSE Instruction

FCMP *fpacs,fpacd*



Function: *fpacs* ?=? *fpacd*
result -> FPSR(N Z)
0 1 (*fpacs=fpacd*)
1 0 (*fpacs>fpacd*)
0 0 (*fpacs<fpacd*)

Parameters: None

NOTE: **FCMP** compares two 64-bit floating-point numbers.

FCMP algebraically compares two 64-bit floating-point numbers in two floating-point accumulators and sets the Z and N flags in the floating-point status register to reflect the results.

Arguments

fpacs Before execution, contains 64-bit floating point number.
After execution, contents unchanged.

fpacd Before execution, contains 64-bit floating point number.
After execution, contents unchanged.

Registers, Flags, and Stacks

FPAC0–FPAC3 Can be individually specified as *fpacs* and *fpacd*; otherwise unused.

FPSR Contains following updated Z and N flags:

N	Z	Result
0	1	*fpacs=fpacd*
1	0	*fpacs>fpacd*
0	0	*fpacs<fpacd*

PC PC + 1

Stack Unchanged

Related Instructions

FSEQ, FSGE, FSGT, FSLE, FSLT, FSNE Test the Z and/or N flags of the FPSR and conditionally skip.

Exceptions

None

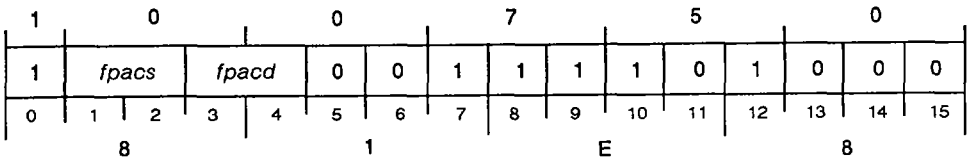
Example

```
FCMP 1,0  ;Compares the values in FPAC1 and FPAC0, returning the
....      ;result to the Z and N flags of the FPSR. Assume that FPAC1
          ;contains 13740270₈ and FPAC0 contains 13651301₈. After the
          ;comparison, FPSR(N) is set to 1, and FPSR(Z) is set to 0,
          ;indicating FPAC1 is greater than FPAC0.
```

Divide Double (FPAC by FPAC) FDD

ECLIPSE Instruction

FDD *fpacs,fpacd*



Function: *fpacd / fpacs -> fpacd*

Parameters: None

FDD divides the 64-bit floating-point number in *fpacd* by the 64-bit floating-point number in *fpacs*, placing the normalized 64-bit result in *fpacd*.

Arguments

fpacs Before execution, contains 64-bit floating-point number.

After execution, contents unchanged unless *fpacs* and *fpacd* are same accumulator.

fpacd Before execution, contains 64-bit floating-point number.

After execution, contains normalized 64-bit result.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be individually specified as *fpacs* and *fpacd*; otherwise unused.

FPSR Updated Z and N flags.

PC PC + 1

Stack Unchanged

Related Instructions

FDS Divide Single (FPAC by FPAC)

Exceptions

If the divisor (in *fpacs*) is zero, the processor sets the INV flag to 1, places error code 0 in the INP bits and the address of the instruction in the FPPC, and terminates the instruction.

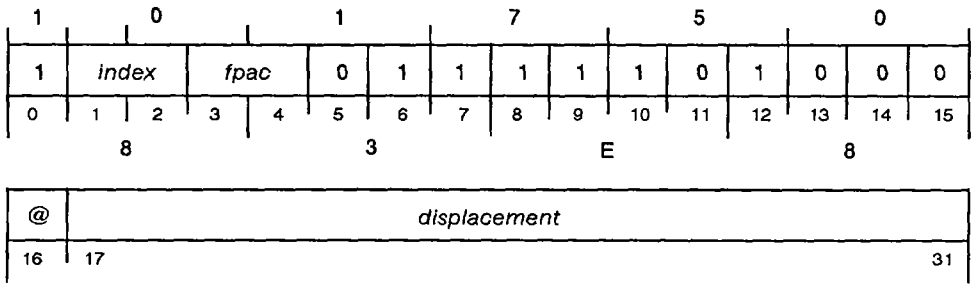
Example

```
FDD 2,0          ;Divides the contents of FPAC0 by the contents
....             ;of FPAC2, placing the result in FPAC0.
```

Divide Double (FPAC by Memory) FDMD

ECLIPSE Instruction

FDMD *fpac*,[@]*displacement*[,*index*]



Function: *fpac* / (E) -> *fpac*

Parameters: None

FDMD divides the 64-bit floating-point number in a floating-point accumulator by a 64-bit floating-point number from a memory location and places the normalized 64-bit result in the *fpac*.

Arguments

fpac Before execution, contains 64-bit floating-point number.

After execution, contains normalized 64-bit result.

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
CARRY	Unchanged
PC	PC + 2
FPSR	Updated Z and N flags.
Stack	Unchanged

Related Instructions

XFDMD, LFDMD, FDMS, XFDMS, LFDMS Divide an accumulator by the contents of memory.

Exceptions

If the divisor (in memory) is zero, the processor sets the FPSR(INV) flag to 1, places error code 0 in the FPSR(INP) bits and the address of the instruction in the FPSR(FPPC), and terminates the instruction.

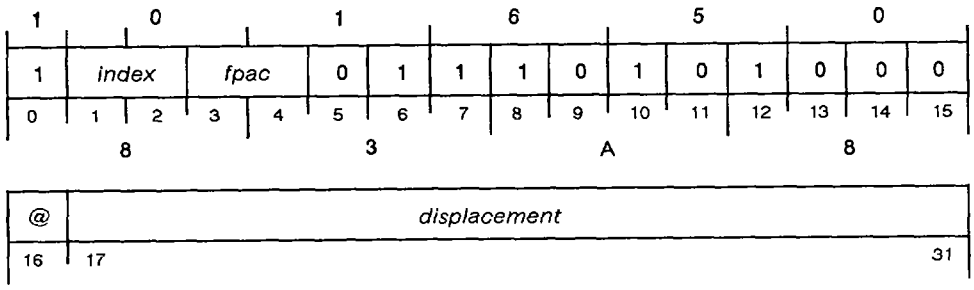
Example

```
FLDD   2,FLOATA     ;Divide the double precision number at
FDMD   2,FLOATB     ;location FLOATA by the double precision
FSTD   2,FLOATC     ;number at location FLOATB, and store the
                    ;result at location FLOATC.
```

Divide Single (FPAC by Memory) FDMS

ECLIPSE Instruction

FDMS *fpac*,[@]*displacement*[,*index*]



Function: *fpac* / (E) -> *fpac*

Parameters: None

FDMS divides the 32-bit floating-point number in a floating-point accumulator by a 32-bit floating-point number from a memory location and places the normalized 32-bit result in the *fpac*.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contains normalized 32-bit result with bits 32-63 set to 0.

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
CARRY	Unchanged
PC	PC + 2
FPSR	Updated Z and N flags.
Stack	Unchanged

Related Instructions

XFDMS, LFDMS, FDMD, XFDMD, LFDMD Divide an accumulator by the contents of memory.

Exceptions

If the divisor (in memory) is zero, the processor sets the FPSR(INV) flag to 1, places error code 0 in the FPSR(INP) bits and the address of the instruction in the FPSR(FPPC), and terminates the instruction.

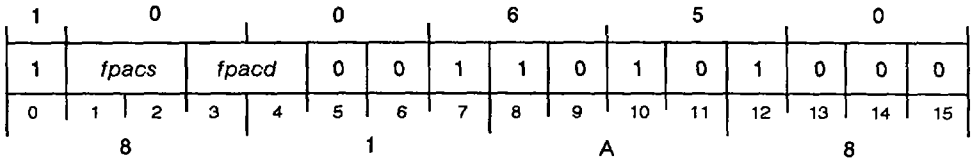
Example

```
FLDS   3,FLOAT1     ;Divide the single precision number at
FDMS   3,FLOAT2     ;location FLOAT1 by the single precision
FSTS   3,FLOAT3     ;number at location FLOAT2, and store the
                    ;result at location FLOAT3.
```

Divide Single (FPAC by FPAC) FDS

ECLIPSE Instruction

FDS *fpacs,fpacd*



Function: *fpacd / fpacs -> fpacd*

Parameters: None

FDS divides the 32-bit floating-point number in *fpacd* by the 32-bit floating-point number in *fpacs* and places the normalized 32-bit result in *fpacd*.

Arguments

fpacs Before execution, contains 32-bit floating-point number.

After execution, contents unchanged unless *fpacs* and *fpacd* are same accumulator.

fpacd Before execution, contains 32-bit floating-point number.

After execution, contains normalized 32-bit result with bits 32-63 set to 0.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be individually specified as *fpacs* and *fpacd*; otherwise unused.

FPSR Updated Z and N flags.

PC PC + 1

Stack Unchanged

Related Instructions

FDD Divide Double (FPAC by FPAC)

Exceptions

If the divisor (in *fpacs*) is zero, the processor sets the FPSR(INV) flag to 1, places error code 0 in the FPSR(INP) bits and the address of the instruction in the FPSR(FPPC), and terminates the instruction.

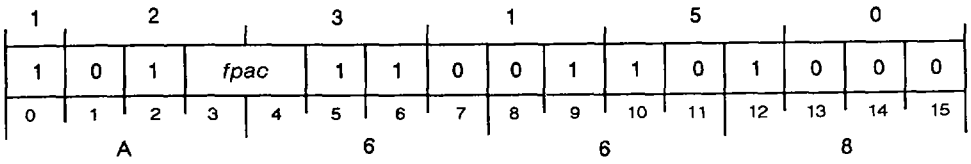
Example

```
FDS 2,0    ;Divides the contents of FPAC0 by the contents
....       ;of FPAC2, placing the result in FPAC0.
```

Load Exponent FEXP

ECLIPSE Instruction

FEXP *fpac*



Function: AC0[#] -> *fpac*(exp)

Parameters: *fpac* = fp# -> fp#(new exponent)

NOTE: If *fpac* = true 0; *fpac* = unchanged

FEXP loads an exponent from AC0 into an *fpac*.

Arguments

fpac Before execution, contains floating-point number.

After execution, bits 1–7 contain bits 17–23 from AC0 and bits 0 and 8–63 remain unchanged.

If *fpac* contains true zero, instruction does not load bits 1–7 of *fpac*.

Registers, Flags, and Stacks

AC0(17–23)	Before execution, contains new 7-bit exponent. Bits 0–16 and 24–31 ignored. After execution, contents unchanged.
AC1–AC3	Unused
FPAC0–FPAC3	Can be individually specified as *fpac*; otherwise unused.
FPSR	Updated Z and N flags.
Overflow	Unaffected
PC	PC + 1
Stack	Unchanged

Related Instructions

Load with immediate — Use these instructions to place a value into AC0.

Exceptions

None

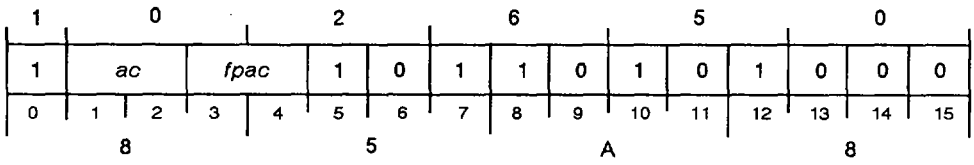
Example

```
XWLDA   0,NEWEXP        ;Use bits 17-23 of the double word at
FEXP    3               ;location NEWEXP to replace the
                        ;exponent in FPAC3.
```

Fix to AC (FPAC to AC) FFAS

ECLIPSE Instruction

FFAS *ac,fpac*



Function: *fpac*[integer] -> *ac*[2#]

Parameters: None

FFAS converts the integer portion of a 64-bit floating-point number in *fpac* to a signed 16-bit integer and places the result in *ac*. The *fpac* integer portion must be within the range of -32,768 to +32,767 inclusive.

The processor truncates the absolute value of the converted integer to the least significant 15 bits and appends a zero to the most significant bit, thus providing a positive result. If the sign of the number in *fpac* is negative, the processor negates the result, and places the result in *ac*.

Arguments

ac(16-31)	After execution, contains converted result.
fpac	Before execution, contains 64-bit floating-point number.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
FPAC0-FPAC3	Can be specified as *fpac*; otherwise unused.
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
FPSR	MOF flag set to 1 if integer portion of floating-point number is outside acceptable range. N and Z flags unchanged.
Stack	Unchanged

Related Instructions

FFMD	Fix to Memory
FINT	Integerize

Exceptions

If the integer portion of the floating-point number is less than -32,768 or greater than +32,767, the contents of *ac* are undefined, and **FFAS** sets FPSR(MOF) to 1.

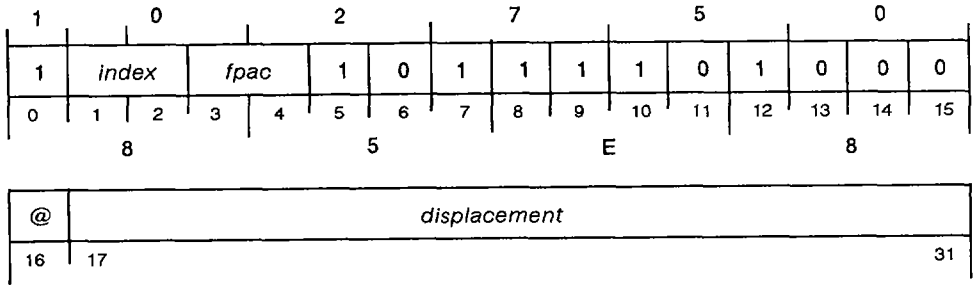
Example

```
FFAS  2,1          ;The processor converts the floating-point
                   ;number in FPAC1 and places the result in AC2.
```

Fix to Memory FFMD

ECLIPSE Instruction

FFMD *fpac*,[@]*displacement*[,*index*]



Function: *fpac*[integer] -> (E)[2# 32-bit]

Parameters: None

FFMD converts the integer portion of the 64-bit floating-point number contained in *fpac* to a signed 32-bit integer and places the result in a memory location.

The range of the integer portion of the floating point number determines the procedure used to produce the signed integer. The range is -2,147,483,648 to +2,147,483,647 inclusive:

If the integer portion is within this range, **FFMD** places the 32-bit, two's complement representation of this value into the memory location.

If the integer portion is outside this range, **FFMD** sets the FPSR(MOF) flag to 1. It then takes the absolute value of the integer portion in *fpac*. It takes the 31 least signficant bits of this absolute value and appends a 0 onto the leftmost bit to give a 32-bit integer. If the sign of *fpac* is negative, **FFMD** negates the 32-bit result. The instruction then places the 32-bit integer into the memory location.

Arguments

fpac Before execution, contains 64-bit floating-point number.

After execution, contents unchanged.

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
CARRY	Unchanged
PC	PC + 2
FPSR	MOF flag set to 1 if *fpac* integer portion is outside range; otherwise unchanged.
Stack	Unchanged

Related Instructions

FFAS	Fix to AC
FLAS	Float from AC
WFLAD	Wide Float from Fixed-Point Accumulator
FLMD	Float from Memory
WFFAD	Wide Fix from Floating-Point Accumulator

Exceptions

If the integer portion of the floating-point number contained in *fpac* is less than -2,147,483,648 or greater than +2,147,483,647, **FFMD** sets the FPSR(MOF) bit to 1.

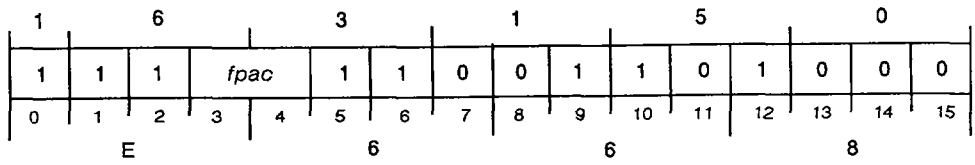
Example

```
FAD    1,2          ;Add FPAC1 to FPAC2, convert the result to
FFMD   2,RESULT     ;a 32-bit integer, and store this at memory
                    ;location RESULT.
```

Halve FHLV

ECLIPSE Instruction

FHLV *fpac*



Function: *fpac* / 2 -> *fpac*

Parameters: None

NOTE: **FHLV** does rounding
FHLV considers *fpac* to contain a 64-bit floating-point number.

FHLV divides the 64-bit floating-point number in *fpac* by 2. It does this by shifting the mantissa in *fpac* right one bit position and filling the vacated bit position with a zero. The bit shifted out is placed in the guard digit, the number is normalized, and then placed in *fpac*.

Arguments

fpac Before execution, contains 64-bit floating-point number.

After execution, contains normalized 64-bit result.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be individually specified as *fpac*; otherwise unused.

FPSR Updated Z and N flags. UNF flag set to 1 if underflow occurs.

PC PC + 1

Stack Unchanged

Related Instructions

FDD, FDS Divide FPAC by FPAC.

Exceptions

If an underflow occurs, the processor sets the FPSR(UNF) bit to 1.

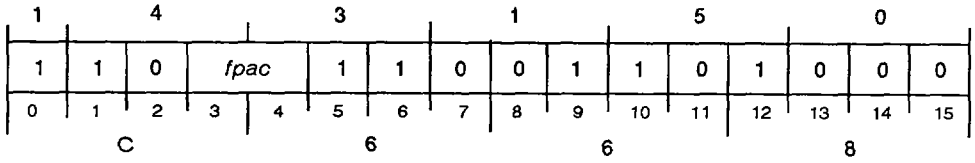
Example

```
FHLV  3   ;Divide the double precision floating-point
          ;number in FPAC3 by two.
```

Integerize FINT

ECLIPSE Instruction

FINT *fpac*



Function: *fpac*[integer] -> *fpac*

Parameters: None

NOTE: **FINT** truncates towards 0 with no rounding
FINT considers *fpac* to contain a 64-bit floating-point number

FINT sets the fractional part of a floating-point number in the specified *fpac* to zero and normalizes the result. The instruction truncates towards zero and does not round results.

Arguments

fpac Before execution, contains floating-point number.

After execution, contains normalized result.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be specified as *fpac*; otherwise unused.

FPSR Updated Z and N flags.

PC PC + 1

Stack Unchanged

Related Instructions

FFAS Fix to AC

FFMD Fix to Memory

Exceptions

If the absolute value of the number contained in the specified *fpac* is less than 1, the specified *fpac* is set to true zero.

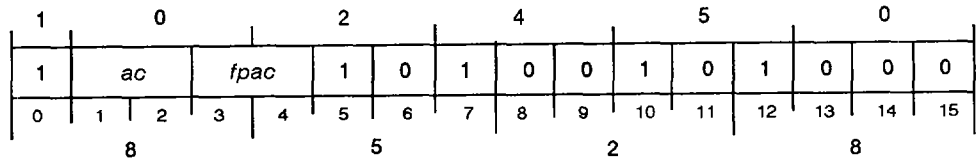
Example

```
FAD   1,2          ;Add FPAC1 to FPAC2, and leave the integer
FINT  2            ;portion of the sum in FPAC2.
```

Float from AC FLAS

ECLIPSE Instruction

FLAS *ac,fpac*



Function: *ac*[2#] -> *fpac*[fp#s]

Parameters: None

FLAS converts a signed 16-bit integer in *ac* to a single-precision floating-point number, placing the normalized result into *fpac*.

Arguments

ac(16-31)	Before execution, contains signed 16-bit integer in range of -32,768 to +32,767.
	After execution, contents unchanged.
fpac(0-31)	After execution, contains normalized result with bits 32-63 set to 0.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
FPAC0-FPAC3	Can be specified as *fpac*; otherwise unused.
FPSR	Updated Z and N flags.
Overflow	Unaffected
PC	PC + 1
Stack	Unchanged

Related Instructions

WFLAD	Wide Float from Fixed-Point Accumulator
FLMD	Float from Memory

Exceptions

None

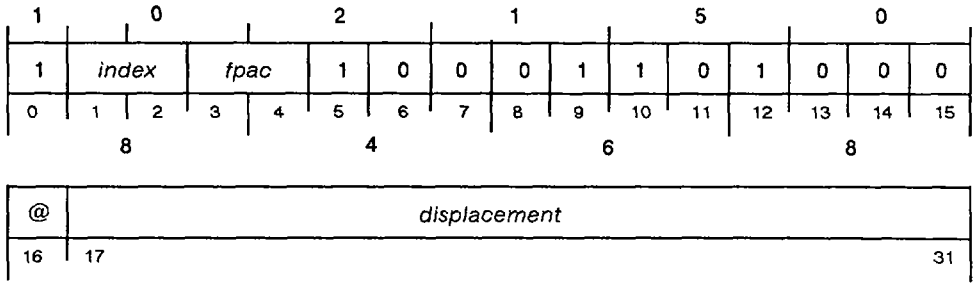
Example

```
XNLDA 0,INTEG      ;Convert the 16-bit integer at location
FLAS  0,2          ;INTEG into a floating point number,
                   ;leaving the result in FPAC2.
```

Load Floating-Point Double FLDD

ECLIPSE Instruction

FLDD *fpac*,[@]*displacement*[,*index*]



Function: (E) -> *fpac*

Parameters: None

FLDD loads a 64-bit floating-point number from the specified memory location into the specified *fpac*. Unnormalized data is moved without change.

Arguments

fpac After execution, contains 64-bit floating-point number.

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
CARRY	Unchanged
PC	PC + 1
FPSR	Updated Z and N flags; undefined for unnormalized data.
Stack	Unchanged

Related Instructions

XFLDD, LFLDD, FLDS, XFLDS, LFLDS Load a floating-point number in memory into a floating-point accumulator.

Exceptions

If the data loaded is unnormalized, FPSR(Z,N) flags are undefined.

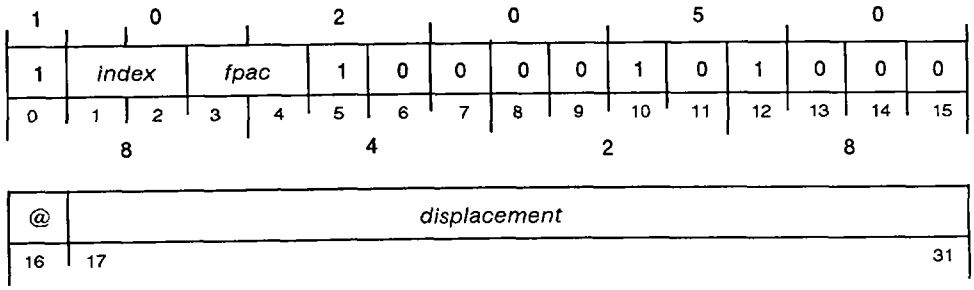
Example

```
FLDD  1,FLPT1      ;Load the double precision floating-point
                   ;number at memory location FLPT1 into FPAC1.
```

Load Floating-Point Single FLDS

ECLIPSE Instruction

FLDS *fpac*,[@]*displacement*[,*index*]



Function: (E) -> *fpac*

Parameters: None

FLDS loads a 32-bit floating-point number from the specified memory location into the specified *fpac*. Unnormalized data is moved without change.

Arguments

fpac(0-31) After execution, contains 32-bit floating-point number with bits 32-63 set to 0.

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
CARRY	Unchanged
PC	PC + 1
FPSR	Updated Z and N flags; undefined for unnormalized data.
Stack	Unchanged

Related Instructions

XFLDS, LFLDS, FLDD, XFLDD, LFLDD Load a floating-point number from memory into a floating-point accumulator.

Exceptions

If the data loaded is unnormalized, FPSR(Z,N) are undefined.

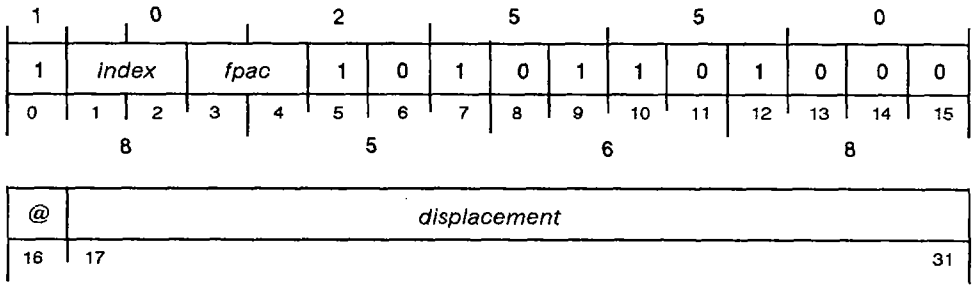
Example

```
FLDS   2,DATA        ;Load the single precision floating-point
                     ;number at memory location DATA into FPAC2.
```

Float from Memory FLMD

ECLIPSE Instruction

FLMD *fpac*,[@]*displacement*[,*index*]



Function: (E)[2# 32-bit] -> *fpac*[fp#d]

Parameters: None

FLMD converts a signed 32-bit integer in memory to a 64-bit floating-point number, and places the result in the specified *fpac*.

Arguments

fpac After execution, contains converted 64-bit floating-point number.

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

FPAC0-FPAC3
Can be individually specified as *fpac*; otherwise unused.

PC PC + 2

FPSR Updated Z and N flags.

Stack Unchanged

Related Instructions

FLAS Float from AC

FFMD Fix to Memory

Exceptions

None

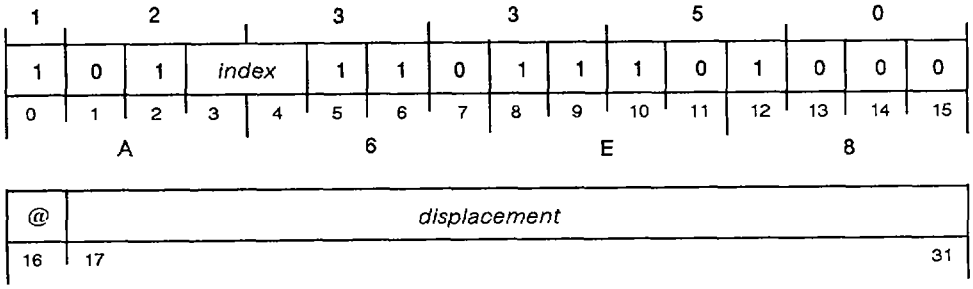
Example

```
FLMD  2,INT1      ;Convert the 32-bit integer at memory
                  ;location INT1 into a floating-point number,
                  ;leaving the result in FPAC2.
```

Load Floating-Point Status FLST

ECLIPSE Instruction

FLST [@]*displacement*[,*index*]



Function: (E) -> FPSR

Parameters: E = fp#s -> unchanged

FLST loads a 32-bit value from memory into the floating-point status register as follows:

FPSR(0) is not set from memory. If any of memory(1-4) are 1, ANY is set to 1; otherwise ANY is 0.

FPSR(1-11) receive memory(1-11)

FPSR(12-15) are not set from memory. These bits contain the floating-point identification code and cannot be changed.

FPSR(16-32) are set to 0

FPSR(33-63):

If ANY = 0, FPSR(33-63) are undefined

If ANY = 1,

FPSR(33-35) receive the value of the current segment

FPSR(36-48) receive zeros

FPSR(49-63) receive memory(17-31)

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

FPAC0-FPAC3	Unused
PC	PC + 2
FPSR	Receives contents of two sequential memory locations.
Stack	Unchanged

Related Instructions

LFLST Load Floating-Point Status (Long Displacement)

Exceptions

Note that if the FPSR ANY and TE flags are both 1 after the FPPC is loaded, execution jumps to a floating-point fault handler routine.

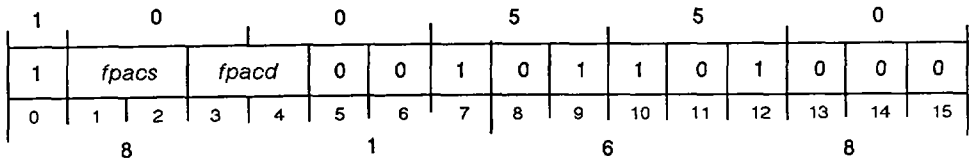
Example

```
FLST   STATUS        ;Load the narrow FPSR from the double word
                     ;at memory location STATUS.
```

Multiply Double (FPAC by FPAC) FMD

ECLIPSE Instruction

FMD *fpacs,fpacd*



Function: *fpacd * fpacs -> fpacd*

Parameters: None

FMD multiplies the 64-bit floating-point number in *fpacd* by the 64-bit floating-point number in *fpacs* and places the normalized 64-bit result in *fpacd*.

Arguments

fpacs Before execution, contains 64-bit floating-point number.

After execution, contents unchanged unless *fpacs* and *fpacd* are same accumulator.

fpacd Before execution, contains 64-bit floating point number.

After execution, contains normalized 64-bit result.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be individually specified as *fpacs* and *fpacd*; otherwise unused.

FPSR Updated Z and N flags.

PC PC + 1

Stack Unchanged

Related Instructions

FMS Multiply Single (FPAC by FPAC)

Exceptions

If multiplication produces an exponent overflow, the processor sets the FPSR(OVF) flag to one and terminates the instruction.

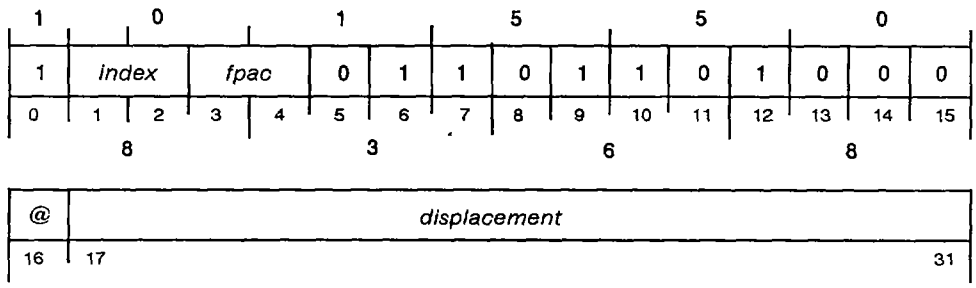
Example

```
FMD 1,3    ;Multiplies the contents of FPAC3 by the
....       ;contents of FPAC1 and returns the result to
           ;FPAC3.
```

Multiply Double (FPAC by Memory) FMMD

ECLIPSE Instruction

FMMD *fpac*,[@]*displacement*[,*index*]



Function: *fpac* * (E) -> *fpac*

Parameters: None

FMMD multiplies a 64-bit floating-point number from a memory location by the 64-bit floating-point number in the *fpac* and places the normalized 64-bit result in the *fpac*.

Arguments

fpac Before execution, contains 64-bit floating-point number.

After execution, contains normalized 64-bit result.

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
CARRY	Unchanged
PC	PC + 2
FPSR	Updated Z and N flags.
Stack	Unchanged

Related Instructions

XFMMD, LFMMD, FMMS, XFMMS, LFMMS Multiply an accumulator by the contents of memory.

Exceptions

If multiplication produces an exponent overflow, the processor sets the FPSR(OVF) flag to 1, and terminates the instruction.

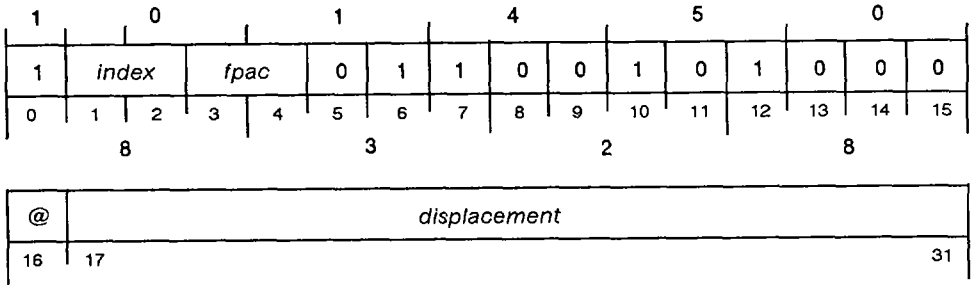
Example

```
FLDD   3,FLPT1      ;Multiply the two double precision numbers
FMMD   3,FLPT2      ;at memory locations FLPT1 and FLPT2, and
FSTD   3,FLPT3      ;store the result at memory location FLPT3.
```

Multiply Single (FPAC by Memory) FMMS

ECLIPSE Instruction

FMMS *fpac*,[@]*displacement*[,*index*]



Function: *fpac* * (E) -> *fpac*

Parameters: None

FMMS multiplies a 32-bit floating-point number from a memory location by the 32-bit floating-point number in the *fpac* and places the normalized 32-bit result in the *fpac*.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contains normalized 32-bit result.

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
CARRY	Unchanged
PC	PC + 2
FPSR	Updated Z and N flags.
Stack	Unchanged

Related Instructions

XFMMS, LFMMS, FMMD, XFMMD, LFMMD Multiply an accumulator by the contents of memory.

Exceptions

If multiplication produces an exponent overflow, the processor sets the FPSR(OVF) flag to 1 and terminates the instruction.

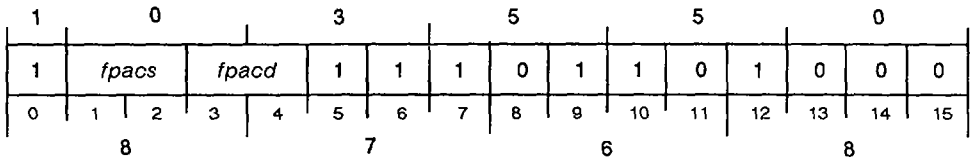
Example

```
FLDS   0,DATA1      ;Multiply the two single precision numbers
FMMS   0,DATA2      ;at memory locations DATA1 and DATA2, and
FSTS   0,RESULT     ;store the result at memory location RESULT.
```

Move Floating Point FMOV

ECLIPSE Instruction

FMOV *fpacs,fpacd*



Function: *fpacs -> fpacd*

Parameters: None

FMOV places the contents of *fpacs* in *fpacd* and updates the Z and N flags of the FPSR to reflect the new contents of *fpacd*. **FMOV** moves unnormalized data without changing it.

Arguments

fpacs	Before execution, contains 64-bit value.
	After execution, contents unchanged unless *fpacs* and *fpacd* are same accumulator.
fpacd	Before execution, contains 64-bit value.
	After execution, contains result.

Registers, Flags, and Stacks

FPAC0–FPAC3	Can be specified as *fpacs* and *fpacd*; otherwise unused.
FPSR	Updated Z and N flags
PC	PC + 1
Stack	Unchanged

Related Instructions

FNOM	Normalize

Exceptions

If unnormalized data is moved, the FPSR Z and N flags are undefined.

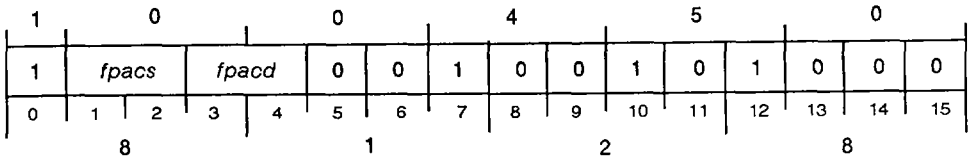
Example

```
FMOV  2,0          ;Places the contents of FPAC2 into FPAC0.
```

Multiply Single (FPAC by FPAC) FMS

ECLIPSE Instruction

FMS *fpacs,fpacd*



Function: *fpacd * fpacs -> fpacd*

Parameters: None

FMS multiplies the 32-bit floating-point number in *fpacs* by the 32-bit floating-point number in *fpacd* and places the normalized 32-bit result in *fpacd*.

Arguments

fpacs(0-31) Before execution, contains 32-bit floating-point number.

After execution, contents unchanged unless *fpacs* and *fpacd* are same accumulator.

fpacd(0-31) Before execution, contains 32-bit floating-point number.

After execution, contains normalized 32-bit result with bits 32-63 set to 0.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be individually specified as *fpacs* and *fpacd*; otherwise unused.

FPSR Updated Z and N flags.

PC PC + 1

Stack Unchanged

Related Instructions

FMD Multiply Double (FPAC by FPAC)

Exceptions

If multiplication produces an exponent overflow, the processor sets the FPSR(OVF) flag to 1 and terminates the instruction.

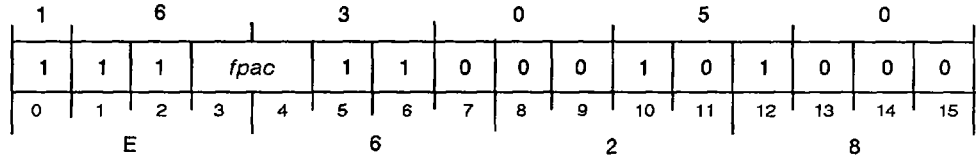
Example

```
FMS   1,3          ;Multiplies the contents of FPAC3 by the
....               ;contents of FPAC1 and returns the result to
                   ;FPAC3.
```

Negate FNEG

ECLIPSE Instruction

FNEG *fpac*



Function: *-fpac -> fpac*

Parameters: None

NOTE: True 0 is unchanged.

FNEG inverts the sign bit of the *fpac*. If *fpac* contains true zero, the instruction leaves the sign bit unchanged.

Arguments

fpac Before execution, contains floating-point number.

After execution, sign bit inverted. Bits 1-63 unchanged.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be specified as *fpac*; otherwise unused.

FPSR Updated Z and N flags.

PC PC + 1

Stack Unchanged

Related Instructions

FAB Absolute Value

Exceptions

None

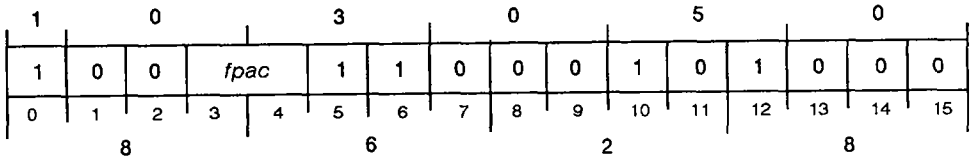
Example

```
FNEG  1   ;Negate the floating-point number in FPAC1.
```

Normalize FNOM

ECLIPSE Instruction

FNOM *fpac*



Function: norm(*fpac*) -> *fpac*

Parameters: None

FNOM normalizes the floating-point number in the *fpac*. Then it sets a true zero in *fpac* if all the bits of the mantissa are zero.

Arguments

fpac Before execution, contains floating-point number.

After execution, contains normalized result.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be specified as *fpac*; otherwise unused.

FPSR Updated Z and N flags.

PC PC + 1

Stack Unchanged

Related Instructions

FINT Integerize

Exceptions

If an exponent underflow occurs, **FNOM** sets the FPSR(UNF) flag to 1.

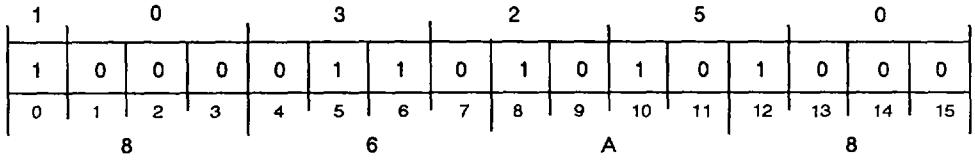
Example

```
LFLDD    2,FLOATD ;Load the double precision number at location
FNOM     2        ;FLOATD into FPAC2, and then make sure it
                  ;is normalized.
```

No Skip FNS

ECLIPSE Instruction

FNS



Function: PC + 1 -> PC

Parameters: None

FNS never skips the next sequential word.

Arguments

None

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

FSA	Skip Always

Exceptions

None

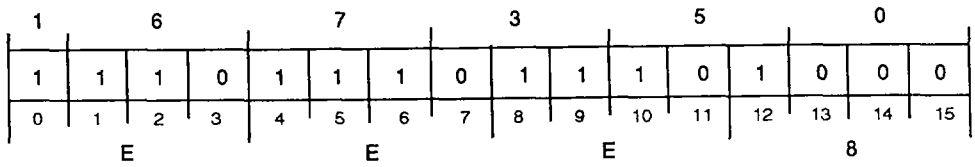
Example

```
FAD   1,2          ;Add FPAC1 to FPAC2, and leave the word
FNS                ;following the FAD opcode available for
                   ;debug purposes.
```

Pop Floating-Point State FPOP

ECLIPSE Instruction

FPOP



Function: 18 stack words -> registers
1st four words -> FPAC3
2nd four words -> FPAC2
3rd four words -> FPAC1
4th four words -> FPAC0
last 2 words -> FPSR

Parameters: None

NOTE: FPSR(bits 12-15) are not set by this instruction

FPOP pops the state of the floating-point unit, an 18-word block, off the narrow stack and loads the contents into the four floating-point accumulators and the floating-point status register. The format of the 18-word block is shown in Figure 11-2.

The first 16 words popped are loaded into the four floating-point accumulators; the last two words popped, a 32-bit operand, are loaded into the floating-point status register as follows:

FPSR(0) is not set from memory. If any of memory(1-4) are 1, ANY is set to 1; otherwise ANY is 0.

FPSR(1-11) receive memory(1-11)

FPSR(12-15) are not set from memory. These bits contain the floating-point identification code and cannot be changed.

FPSR(16-32) are set to 0.

FPSR(33-63) are set according to ANY:

If ANY = 0, FPSR(33-63) are undefined
If ANY = 1
FPSR(33-35) receive the value of the current segment
FPSR(36-48) receive zeros
FPSR(49-63) receive memory(17-31)

Unnormalized data is moved without change.

Arguments

None

Registers, Flags, and Stacks

FPAC0–FPAC3	Receive data from stack as shown in Figure 11–2.
CARRY	Unchanged
PC	PC + 1
FPSR	Receives data from stack two words.
Stack	Narrow stack pointer decremented by 18 words.

Related Instructions

WFPOP	Wide Pop Floating-Point State
FPSH, WFPSH	Push floating-point state onto stack.

Exceptions

Note that if the FPSR ANY and TE flags are both 1 after the FPPC is loaded, execution jumps to a floating-point fault handler routine.

Example

```
FPSH              ;Save the floating point state, because
EJSR  ROUTINE     ;this subroutine may change the values in the
FPOP              ;FPACs, and we want to keep them.The FPOP
                  ;restores the FPACs to their previous values.
```

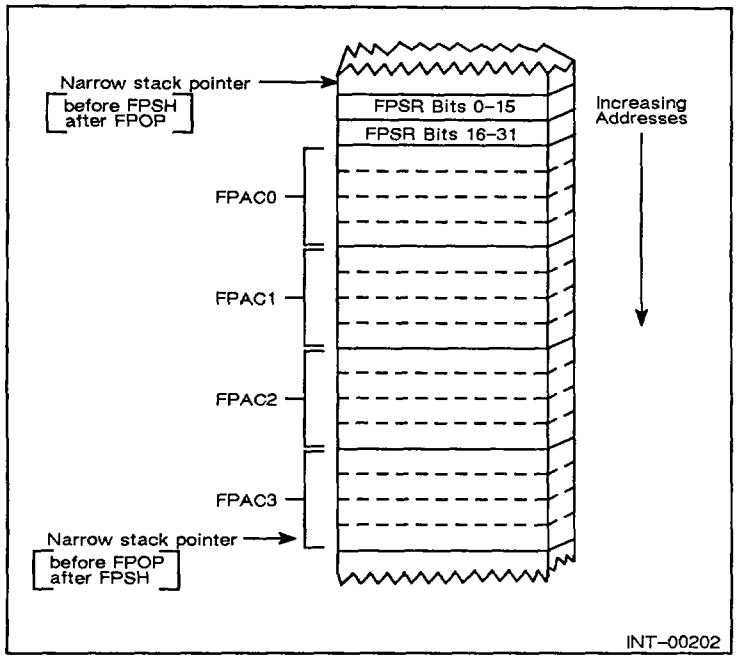
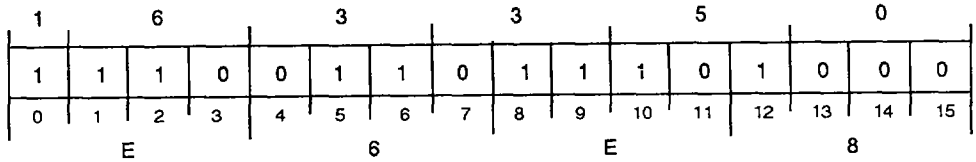


Figure 11–2 *Narrow stack, 18–word floating–point return block*

Push Floating-Point State FPSH

ECLIPSE Instruction

FPSH



Function: registers -> 18 stack words
FPSR -> 1st 2 words
FPAC0 -> 2nd 4 words
FPAC1 -> 3rd 4 words
FPAC2 -> 4th 4 words
FPAC3 -> 5th 4 words

Parameters: None

FPSH pushes the state of the floating-point unit, an 18-word return block, onto the narrow stack, leaving the contents of the floating-point accumulators and the floating-point status register unchanged. The format of the 18 words pushed is shown in Figure 11-2.

The first two words pushed onto the stack, a 32-bit operand, are taken from the floating-point status register as follows:

FPSR(0-15) comprise memory(0-15)
If ANY = 0, memory(16-31) will be undefined
If ANY = 1, FPSR(49-63) comprise memory(17-31) with memory(16) set to 0.

The next 16 words to be pushed are taken from the four fpacs.

Unnormalized data is moved without change.

Arguments

None

Registers, Flags, and Stacks

FPAC0-FPAC3	Provide data to the stack as shown in Figure 11-2.
CARRY	Unchanged
PC	PC + 1
FPSR	Unchanged
Stack	Narrow stack pointer incremented by 18 words.

Related Instructions

FPOP, WFPOP	Pop floating-point state from the stack.
WFPSH	Wide Push Floating-Point State

Exceptions

FPSH will not store a value with any combination of bit 5 (TE) and bits 1–4 concurrently set.

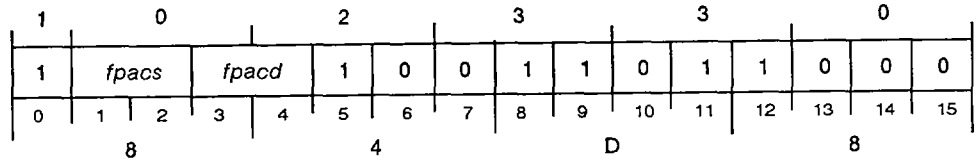
Example

```
FPSH                ;Save all FPACs, so WFSIND won't destroy
LFLDD     0,FDATA1  ;them. We want to take the sine of FDATA1.
WFSINDSINE          ;Take the sine -- FPAC values are destroyed.
LFSTD     0,FDATA1  ;Replace FDATA1 with its sine.
FPOP                ;Restore original floating point state.
```

Floating-Point Round Double to Single FRDS

ECLIPSE Instruction

FRDS *fpacs,fpacd*



Function: If FPSR(8) = 1,
Then *fpacs*[fp#d] rounded -> *fpacd*[fp#s]
Else *fpacs*[fp#d] truncated -> *fpacd*[fp#s]

Parameters: None

FRDS rounds a floating-point number according to the setting of the floating-point status register RND bit.

The algorithm is similar to unbiased rounding, except that it uses eight rounding digits instead of two guard digits.

The instruction forms the 32-bit result by normalizing the result mantissa and appending the *fpacs* sign and exponent (bits 0-7). Then it places the 32-bit result in *fpacd*.

Arguments

fpacs(8-31) Before execution, if FPSR(8) = 1, contains unrounded mantissa.

After execution, contents unchanged unless *fpacs* and *fpacd* are same accumulator.

fpacs(32-63) Before execution, if FPSR(8) = 1, contains rounding digits in 3 ranges:

0 to $7FFFFFFF_{16}$ inclusive. Result mantissa equals unrounded mantissa without change.

80000000_{16}. Result mantissa formed by adding least significant bit of unrounded mantissa to unrounded mantissa.

80000001_{16} to $FFFFFFFF_{16}$ inclusive. Result mantissa equals unrounded mantissa plus 1.

After execution, contents unchanged unless *fpacs* and *fpacd* are same accumulator.

fpacd(0-31) After execution:

If FPSR(8) = 0, contains *fpacs*.

If FPSR(8) = 1, contains rounded bits from *fpacs*(0-31).

Bits 32-63 always 0.

Registers, Flags, and Stacks

FPAC0–FPAC3	Can be specified as *fpacs* and *fpacd*; otherwise unused.
FPSR	Updated Z and N flags.
PC	PC + 1
Stack	Unchanged

Related Instructions

FINT	Integerize

Exceptions

None

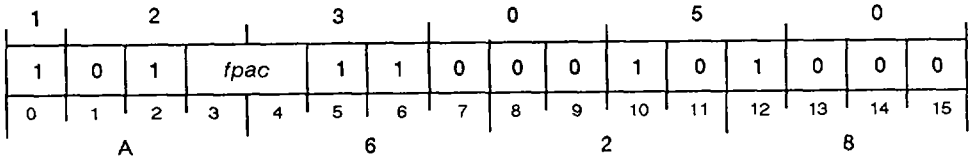
Example

```
FAD   1,2          ;Add FPAC1 to FPAC2, and round the double
FRDS  2,3          ;precision result to single precision,
                   ;leaving the result in FPAC3.
```

Read High Word — FRH

ECLIPSE Instruction

FRH *fpac*



Function: *fpac*[high 16 bits zero-extended] -> AC0

Parameters: None

FRH zero-extends the value in the high-order 16 bits of *fpac* to 32 bits and places it in AC0. The instruction moves unnormalized data without change.

Arguments

fpac	Before execution, contains floating-point number.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0	After execution, contains zero-extended value from high-order 16 bits of *fpac*.
AC1-AC3	Unused
FPAC0-FPAC3	Can be specified as *fpac*; otherwise unused.
FPSR	Unchanged
Overflow	Unaffected
PC	PC + 1
Stack	Unchanged

Related Instructions

FEXP	Load Exponent

Exceptions

None

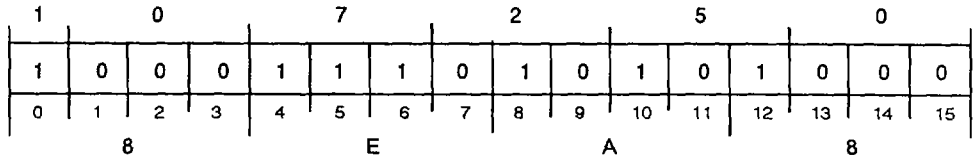
Example

```
FAS   0,1          ;Add FPAC0 to FPAC1, and store the result in
FRH   1            ;AC0.
```

Skip Always FSA

ECLIPSE Instruction

FSA



Function: PC + 2 -> PC

Parameters: None

FSA always skips the next sequential word.

Arguments

None

Registers, Flags, and Stacks

AC0-AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

FNS	No Skip

Exception

None

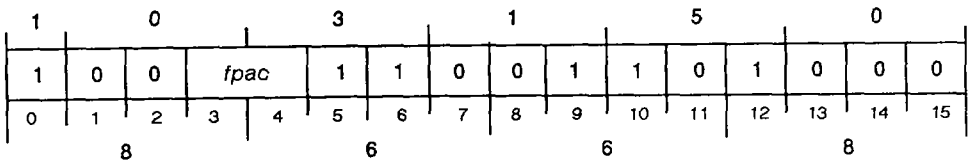
Example

```
FAD   3,2          ;Add FPAC3 to FPAC2, and leave the two words
FSA                ;following the FAD opcode available for debug
FSA                ;purposes. The second FSA is always skipped.
```

Scale FSCAL

ECLIPSE Instruction

FSCAL *fpac*



Function: AC0[#](17-23) - *fpac*(exp) = *fpac*(mantissa shift)
AC0(17-23) -> *fpac*(exp)

Parameters: None

NOTE: **FSCAL** sets *fpac* to True Zero if all mantissa bits shifted out.

FSCAL shifts the mantissa of the floating-point number in *fpac* either right or left, depending upon the contents of AC0.

It does this by subtracting the *fpac* exponent from the exponent in AC0. The difference between the exponents specifies D, a number of hex digits.

If D is zero or if *fpac* is true zero, the instruction updates the Z and N flags and stops.

If D is positive, the instruction shifts the mantissa of the number contained in *fpac* to the right by D digits.

If D is negative, the instruction shifts the mantissa of the number contained in *fpac* to the left by D digits. Then it sets the MOF flag in the floating-point status register.

After the right or left shift, the instruction loads the contents of bits 17–23 of AC0 into the exponent field of *fpac*. Bits shifted out of either end of the mantissa are lost.

If all the bits in the mantissa are shifted out, **FSCAL** will set the *fpac* to true zero. The instruction does not do rounding.

Arguments

fpac	Before execution, contains a 64-bit floating-point number.
	After execution, contains result.

Registers, Flags, and Stacks

AC0(17–23)	Before execution, contains unchanged exponent.
	After execution, contents unchanged.
AC1–AC3	Unused
FPAC0–FPAC3	Can be specified as *fpac*; otherwise unused.
FPSR	Updated Z, N, and MOV flags.
PC	PC + 1
Stack	Unchanged

Related Instructions

Load with immediate	Use these instructions to place a shifting value into AC0.

Exceptions

None

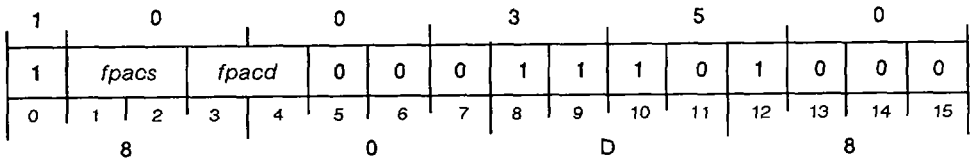
Example

```
FSCAL  3          ;Scale the value in FPAC3 according to
FSNM              ;bits 17-23 of AC0. This can be
JMP    TOOBIG     ;used to check the exponent of the value in
JMP    NOTBIG     ;FPAC3. If FPAC3's exponent is larger than
                  ;the value in bits 17-23 of AC0, jump to
                  ;TOOBIG. Otherwise, jump to NOTBIG.
```

Subtract Double (FPAC from FPAC) FSD

ECLIPSE Instruction

FSD *fpacs,fpacd*



Function: *fpacd - fpacs -> fpacd*

Parameters: None

FSD subtracts the 64-bit floating-point number in *fpacs* from the 64-bit floating-point number in *fpacd* and places the normalized 64-bit result in *fpacd*.

Arguments

fpacs Before execution, contains 64-bit floating point number.

After execution, contents unchanged unless *fpacs* and *fpacd* are same accumulator.

fpacd Before execution, contains 64-bit floating point number.

After execution, contains normalized 64-bit result.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be individually specified as *fpacs* and *fpacd*; otherwise unused.

FPSR Updated Z and N flags.

PC PC + 1

Stack Unchanged

Related Instructions

FSS Subtract Single (FPAC from FPAC)

Exceptions

If subtraction produces an exponent overflow or underflow, the processor sets the appropriate FPSR flag to 1 and terminates the instruction.

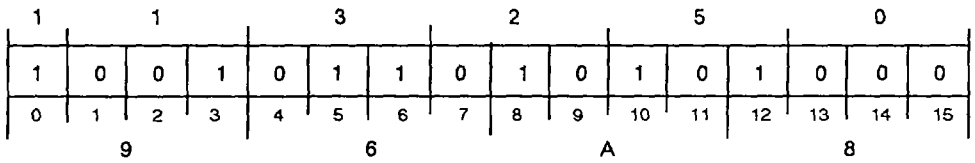
Example

```
FSD   2,0          ;Subtracts the contents of FPAC2 from FPAC0
....               ;and returns the result to FPAC0.
```

Skip on Zero FSEQ

ECLIPSE Instruction

FSEQ
(Z = 0 return)
(Z = 1 return)



Function: If FPSR(Z) = 1 then skip

Parameters: None

FSEQ skips the next sequential word if the Z flag of the floating-point status register is 1.

Arguments

None

Registers, Flags, and Stacks

CARRY	Unchanged
FPAC0–FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if Z = 0) PC + 2 (if Z = 1)
Stack	Unchanged

Related Instructions

Floating-point FPSR skip instructions.

Exceptions

None

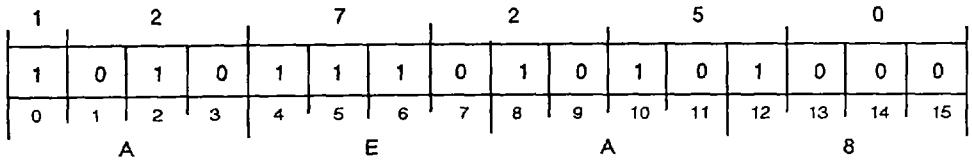
Example

```
FCMP   1,2          ;Compare the values in FPAC1 and FPAC2.
FSEQ                ;If they are equal, jump to EQUAL. Otherwise,
JMP    NEQUAL       ;jump to NEQUAL.
JMP    EQUAL
```

Skip on Greater than or Equal to Zero FSGE

ECLIPSE Instruction

FSGE
(N ≠ 0 return)
(N = 0 return)



Function: If FPSR (N) = 0 then skip

Parameters: None

FSGE skips the next sequential word if the N flag of the floating-point status register is 0.

Arguments

None

Registers, Flags, and Stacks

CARRY	Unchanged
FPAC0–FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if N ≠ 0) PC + 2 (if N = 0)
Stack	Unchanged

Related Instructions

Floating-point FPSR skip instructions.

Exceptions

None

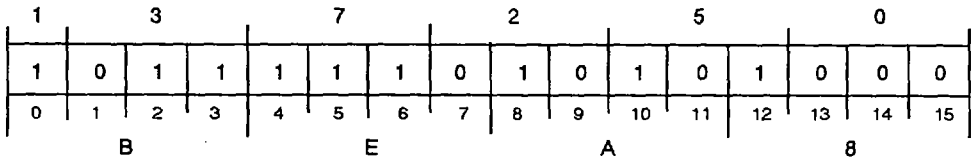
Example

```
FCMP   0,1          ;Compare the values in FPAC0 and FPAC1.
FSGE                ;If FPAC1 >= FPAC0, skip one word.
JMP    LOC2         ;Jump to LOC2 if FPAC1 < FPAC0.
JMP    LOC1         ;Jump to LOC1 if FPAC1 >= FPAC0.
```

Skip on Greater than Zero FSGT

ECLIPSE Instruction

FSGT
(Z or N ≠ 0 return)
(Z and N = 0 return)



Function: If FPSR(N&Z) = 0 then skip

Parameters: None

FSGT skips the next sequential word if both the Z and N flags of the floating-point status register are 0.

Arguments

None

Registers, Flags, and Stacks

CARRY	Unchanged
FPAC0–FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if Z or N ≠ 0) PC + 2 (if Z and N = 0)
Stack	Unchanged

Related Instructions

Floating-point FPSR skip instructions.

Exceptions

None

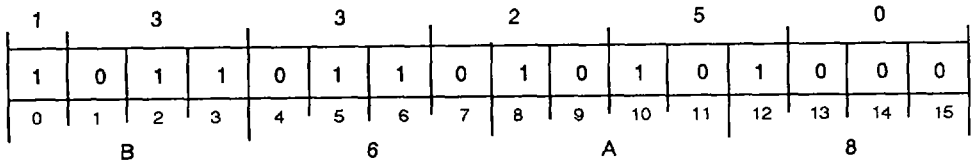
Example

```
FCMP   3,2          ;Compare the values in FPAC3 and FPAC2.
FSGT                ;If FPAC2 > FPAC3, skip one word.
JMP    LOC2         ;Jump to LOC2 if FPAC2 <= FPAC3.
JMP    LOC1         ;Jump to LOC1 if FPAC2 > FPAC3.
```

Skip on Less than or Equal to Zero FSLE

ECLIPSE Instruction

FSLE
(Z and N ≠ 1 return)
(Z or N = 1 return)



Function: If FPSR(N or Z) = 1 then skip

Parameters: None

FSLE skips the next sequential word if either the Z flag or the N flag of the floating-point status register is 1.

Arguments

None

Registers, Flags, and Stacks

CARRY	Unchanged
FPAC0–FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if Z and N≠1) PC + 2 (If Z or N = 1)
Stack	Unchanged

Related Instructions

Floating-point FPSR skip instructions.

Exceptions

None

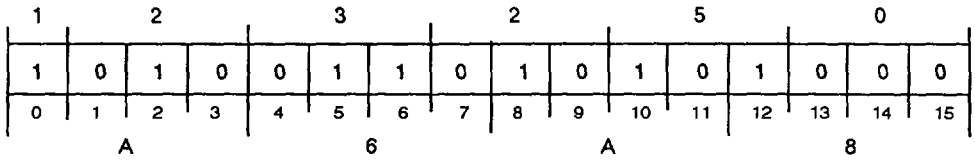
Example

```
FCMP   1,3          ;Compare the values in FPAC1 and FPAC3.
FSLE                ;If FPAC3 <= FPAC1, skip one word.
JMP    LOC2         ;Jump to LOC2 if FPAC3 > FPAC1.
JMP    LOC1         ;Jump to LOC1 if FPAC3 <= FPAC1.
```

Skip on Less than Zero FSLT

ECLIPSE Instruction

FSLT
(N ≠ 1 return)
(N = 1 return)



Function: If FPSR(N) = 1 then skip

Parameters: None

FSLT skips the next sequential word if the N flag of the floating-point status register is 1.

Arguments

None

Registers, Flags, and Stacks

CARRY	Unchanged
FPAC0–FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if N ≠ 1) PC + 2 (if N = 1)
Stack	Unchanged

Related Instructions

Floating-point FPSR skip instructions.

Exceptions

None

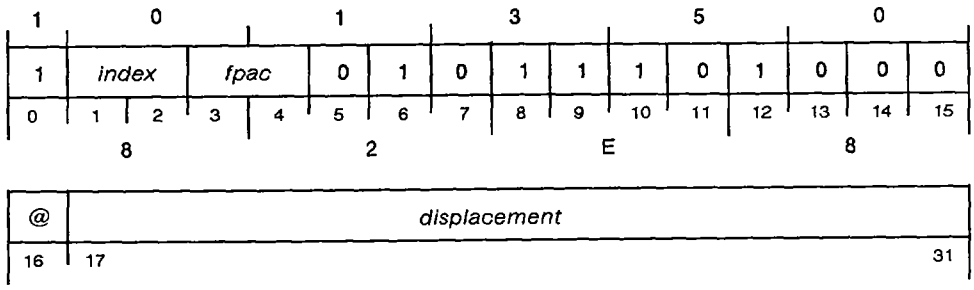
Example

```
FCMP   1,2          ;Compare the values in FPAC1 and FPAC2.
FSLT                ;If FPAC2 < FPAC1, skip one word.
JMP    LOC2         ;Jump to LOC2 if FPAC2 >= FPAC1.
JMP    LOC1         ;Jump to LOC1 if FPAC2 < FPAC1.
```

Subtract Double (Memory from FPAC) FSMD

ECLIPSE Instruction

FSMD *fpac*,[@]*displacement*[,*index*]



Function: *fpac* - (E) -> *fpac*

Parameters: None

FSMD subtracts a 64-bit floating-point number in memory from a 64-bit floating-point number in a floating-point accumulator, and places the normalized 64-bit result in the *fpac*.

Arguments

fpac Before execution, contains 64-bit floating-point number.

After execution, contains normalized 64-bit result.

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
CARRY	Unchanged
PC	PC + 2
FPSR	Updated Z and N flags.
Stack	Unchanged

Related Instructions

XFSMD, LFSMD, FSMS, XFSMS, LFSMS Subtract the contents of memory from an accumulator.

Exceptions

If subtraction produces an exponent overflow or underflow, the processor sets the appropriate FPSR flag to 1 and terminates the instruction.

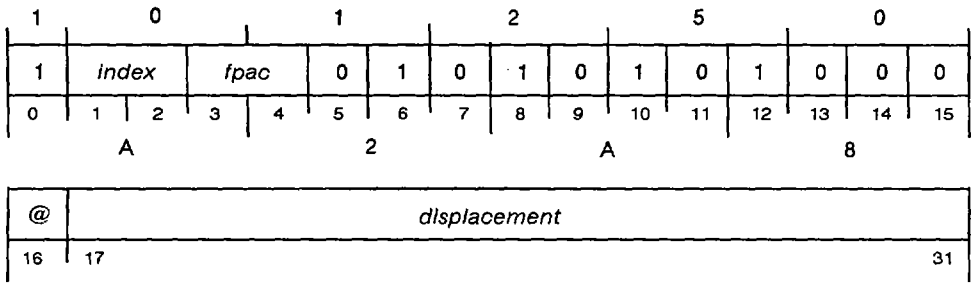
Example

```
FLDD  0,DATA1     ;Subtract the double precision floating point
FSMD  0,DATA2     ;number at memory location DATA2 from the
FSTD  0,DATA3     ;double precision floating point number at
                  ;memory location DATA1, storing the result at
                  ;memory location DATA3.
```

Subtract Single (Memory from FPAC) FSMS

ECLIPSE Instruction

FSMS *fpac*,[@]*displacement*[,*index*]



Function: *fpac* - (E) -> *fpac*

Parameters: None

FSMS subtracts a 32-bit floating-point number in memory from a 32-bit floating-point number in a floating-point accumulator, and places the normalized 32-bit result in the *fpac*.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contains normalized 32-bit result with bits 32-63 set to 0.

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
CARRY	Unchanged
PC	PC + 2
FPSR	Updated Z and N flags.
Stack	Unchanged

Related Instructions

XFSMS, LFSMS, FSMD, XFSMD, LFSMD Subtract the contents of memory from an accumulator.

Exceptions

If subtraction produces an exponent overflow or underflow, the processor sets the appropriate FPSR flag to 1 and terminates the instruction.

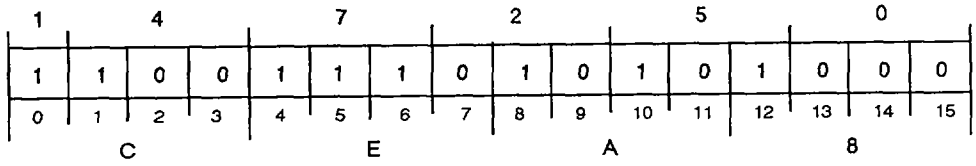
Example

```
FLDS   1,FLPT1      ;Subtract the single precision floating-point
FSMS   1,FLPT2      ;number at memory location FLPT2 from the
FSTS   1,RESULT     ;single precision floating-point number at
                    ;memory location FLPT1, storing the result at
                    ;memory location RESULT.
```

Skip on No Invalid Input Argument — FSND

ECLIPSE Instruction

FSND
(INV ≠ 0 return)
(INV = 0 return)



Function: If FPSR(INV) = 0 then skip

Parameters: None

FSND skips the next sequential word if the INV flag of the floating-point status register is 0. (**FSND** was previously called "Skip on No Zero Divide" and remains valid for this function. Refer to "Faults and Status" in the chapter on floating-point computation.)

Arguments

None

Registers, Flags, and Stacks

CARRY	Unchanged
FPAC0–FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if INV ≠ 0) PC + 2 (if INV = 0)
Stack	Unchanged

Related Instructions

Floating-point FPSR skip instructions.

Exceptions

None

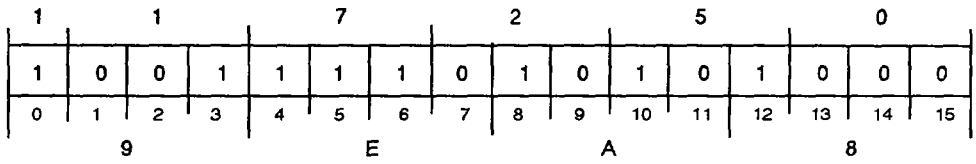
Example

```
FDD    2,3          ;Divide FPAC3 by FPAC2. If there is a divide
FSND                ;error (noted by the INV bit being set), jump
JMP    ERROR        ;to ERROR. Otherwise, jump to NOERROR.
JMP    NOERROR
```

Skip on Nonzero FSNE

ECLIPSE Instruction

FSNE
(Z ≠ 0 return)
(Z = 0 return)



Function: If FPSR(Z) = 0 then skip

Parameters: None

FSNE skips the next sequential word if the Z flag of the floating-point status register is 0.

Arguments

None

Registers, Flags, and Stacks

CARRY	Unchanged
FPAC0-FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if Z ≠ 0) PC + 2 (if Z = 0)
Stack	Unchanged

Related Instructions

Floating-point FPSR skip instructions.

Exceptions

None

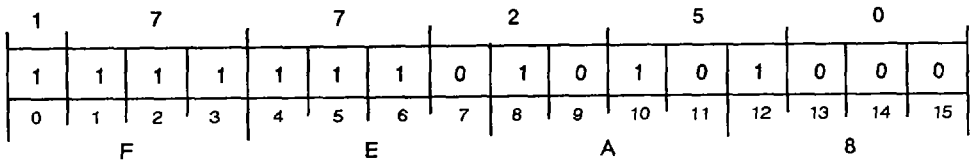
Example

```
FCMP   1,2          ;Compare the values in FPAC1 and FPAC2.
FSNE                ;If the values are not equal, jump to NEQUAL.
JMP    EQUAL        ;If the values are equal, jump to EQUAL.
JMP    NEQUAL
```

Skip on No Error FSNER

ECLIPSE Instruction

FSNER
(FPSR bits 1–4 ≠ 0 return)
(FPSR bits 1–4 = 0 return)



Function: If FPSR(1–4) = 0 then skip

Parameters: None

FSNER skips the next sequential word if bits 1–4 of the floating-point status register are all 0.

Arguments

None

Registers, Flags, and Stacks

CARRY	Unchanged
FPAC0–FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if FPSR bits 1–4 ≠ 0) PC + 2 (if FPSR bits 1–4 = 0)
Stack	Unchanged

Related Instructions

Floating-point FPSR skip instructions.

Exceptions

None

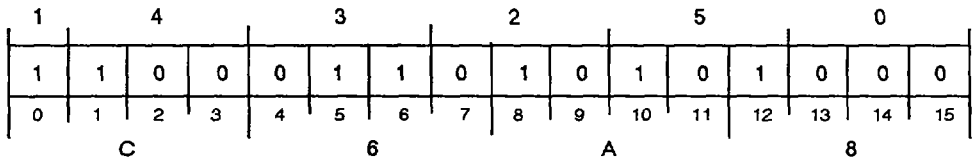
Example

```
EJSR   ROUTINE      ;Call a floating point routine.
FSNER               ;If there are no floating-point errors
JMP    ERROR        ;when the routine returns, jump to NOERROR.
JMP    NOERROR      ;Otherwise, jump to ERROR.
```

Skip on No Mantissa Overflow FSNM

ECLIPSE Instruction

FSNM
(MOF ≠ 0 return)
(MOF = 0 return)



Function: If FPSR(MOF) = 0 then skip

Parameters: None

FSNM skips the next sequential word if the mantissa overflow flag (MOV) of the floating-point status register is 0.

Arguments

None

Registers, Flags, and Stacks

CARRY	Unchanged
FPAC0-FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if MOF≠ 0) PC + 2 (if MOF = 0)
Stack	Unchanged

Related Instructions

Floating-point FPSR skip instructions.

Exceptions

None

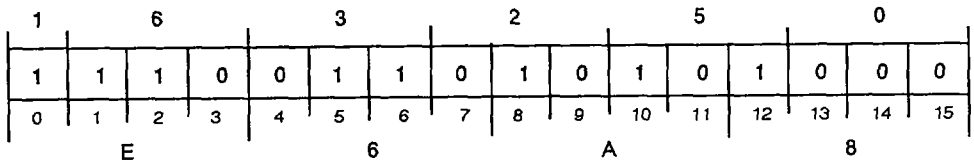
Example

```
FFAS   1,2          ;Convert the contents of FPAC2 to an
FSNM                ;integer, storing the result in AC1.
JMP    OVERFL       ;If mantissa overflow results, jump to
JMP    NOOVER       ;OVERFL. Otherwise, jump to NOOVER.
```

Skip on No Overflow FSNO

ECLIPSE Instruction

FSNO
(OVF ≠ 0 return)
(OVF = 0 return)



Function: If FPSR(OVF) = 0 then skip

Parameters: None

FSNO skips the next sequential word if the overflow (OVF) flag of the floating-point status register is 0.

Arguments

None

Registers, Flags, and Stacks

CARRY	Unchanged
FPAC0–FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if OVF ≠ 0) PC + 2 (if OVF = 0)
Stack	Unchanged

Related Instructions

Floating-point FPSR skip instructions.

Exceptions

None

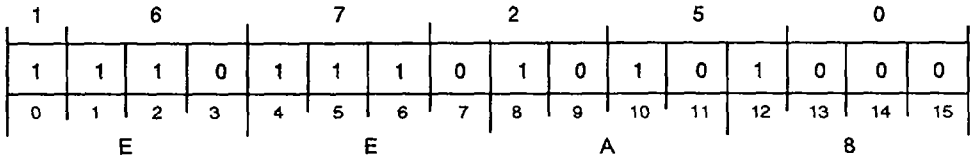
Example

```
FAS   2,3          ;Add FPAC3 to FPAC2. If overflow results,
FSNO               ;jump to OVERFL. Otherwise, jump to NOOVER.
JMP   OVERFL
JMP   NOOVER
```

Skip on No Overflow and No Invalid Input Argument — FSNOD

ECLIPSE Instruction

FSNOD
(INV or OVF ≠ 0 return)
(INV and OVF = 0 return)



Function: If FPSR(OVF&INV) = 0 then skip

Parameters: None

FSNOD skips the next sequential word if both the OVF flag and the INV flag of the floating-point status register are 0. (**FSNOD** was previously called "Skip on No Overflow and No Zero Divide.")

Arguments

None

Registers, Flags, and Stacks

CARRY	Unchanged
FPAC0–FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if INV or OVF ≠ 0) PC + 2 (if INV and OVF = 0)
Stack	Unchanged

Related Instructions

Floating-point FPSR skip instructions.

Exceptions

None

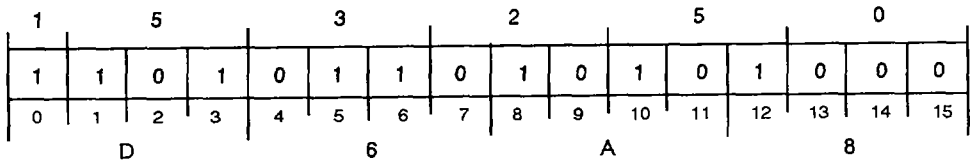
Example

```
FAD    1,2          ;Add FPAC1 to FPAC2, and divide the result
FDD    3,2          ;by FPAC3. If neither the OVF nor the INV
FSNOD               ;bits are set (meaning there was no error),
JMP    ERROR        ;jump to NOERROR. Otherwise, jump to ERROR.
JMP    NOERROR
```

Skip on No Underflow FSNU

ECLIPSE Instruction

FSNU
(UNF ≠ 0 return)
(UNF = 0 return)



Function: If FPSR(UNF) = 0 then skip

Parameters: None

FSNU skips the next sequential word if the underflow flag (UNF) of the floating-point status register is 0.

Arguments

None

Registers, Flags, and Stacks

CARRY	Unchanged
FPAC0–FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if UNF ≠ 0) PC + 2 (if UNF = 0)
Stack	Unchanged

Related Instructions

Floating-point FPSR skip instructions.

Exceptions

None

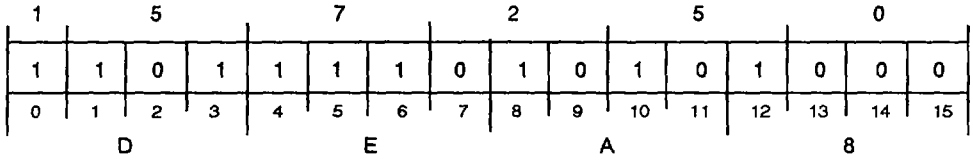
Example

```
FSD    0,3          ;Subtract FPAC0 from FPAC3. If an underflow
FSNU                ;results, jump to UNDERF. Otherwise, jump to
JMP    UNDERF       ;NOUNDER.
JMP    NOUNDER
```

Skip on No Underflow and No Invalid Input Argument — FSNUD

ECLIPSE Instruction

FSNUD
(UNF or INV ≠ 0 return)
(UNF and INV = 0 return)



Function: If FPSR(UNF&INV) = 0 then skip

Parameters: None

FSNUD skips the next sequential word if both the UNF flag and the INV flag of the floating-point status register are 0. (**FSNUD** was previously called "Skip on No Underflow and No Zero Divide.")

Arguments

None

Registers, Flags, and Stacks

CARRY	Unchanged
FPAC0–FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if UNF or INV ≠ 0) PC + 2 (if UNF and INV = 0)
Stack	Unchanged

Related Instructions

Floating-point FPSR skip instructions.

Exceptions

None

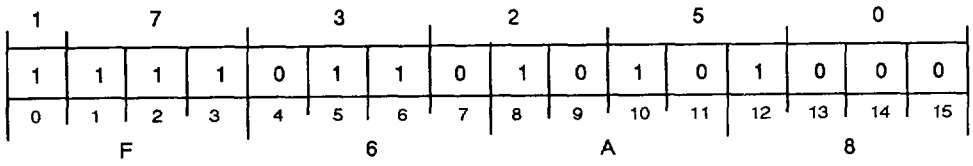
Example

```
FSD     1,2         ;Subtract FPAC1 from FPAC2, and divide the
FDD     3,2         ;result by FPAC3. If neither the UNF nor the
FSNUD               ;INV bits are set (meaning there was no
JMP     ERROR       ;error), jump to NOERROR. Otherwise, jump to
JMP     NOERROR     ;ERROR.
```

Skip on No Underflow and No Overflow FSNUO

ECLIPSE Instruction

FSNUO
(UNF or OFV ≠ 0 return)
(UNF and OVF = 0 return)



Function: If FPSR(UNF&OVF) = 0 then skip

Parameters: None

FSNUO skips the next sequential word if both the UNF flag and the OVF flag of the floating-point status register are 0.

Arguments

None

Registers, Flags, and Stacks

CARRY	Unchanged
FPAC0-FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if UNF or OFV ≠ 0) PC + 2 (if UNF and OVF = 0)
Stack	Unchanged

Related Instructions

Floating-point FPSR skip instructions.

Exceptions

None

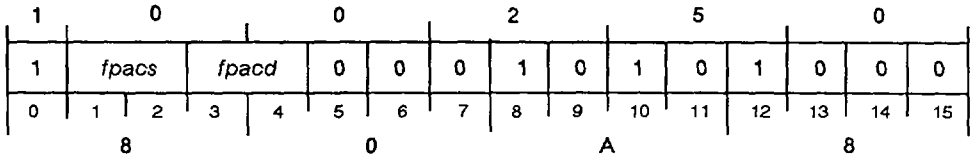
Example

```
FAD    2,3          ;Add FPAC2 to FPAC3. If there was neither
FSNUO               ;overflow nor underflow, jump to NOERROR.
JMP    ERROR        ;Otherwise, jump to overflow.
JMP    NOERROR
```

Subtract Single (FPAC from FPAC) FSS

ECLIPSE Instruction

FSS *fpacs,fpacd*



Function: *fpacd - fpacs -> fpacd*

Parameters: None

FSS subtracts the 32-bit floating-point number in *fpacs* from the 32-bit floating-point number in *fpacd* and places the normalized 32-bit result in *fpacd*.

Arguments

fpacs Before execution, contains 32-bit floating-point number.

After execution, contents unchanged unless *fpacs* and *fpacd* are same accumulator.

fpacd Before execution, contains 32-bit floating-point number.

After execution, contains normalized 32-bit result with bits 32-63 set to 0.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be individually specified as *fpacs* and *fpacd*; otherwise unused.

FPSR Updated Z and N flags.

PC PC + 1

Stack Unchanged

Related Instructions

FSD Subtract Double (FPAC from FPAC)

Exceptions

If subtraction produces an exponent overflow or underflow, the processor sets the appropriate FPSR flag to 1 and terminates the instruction.

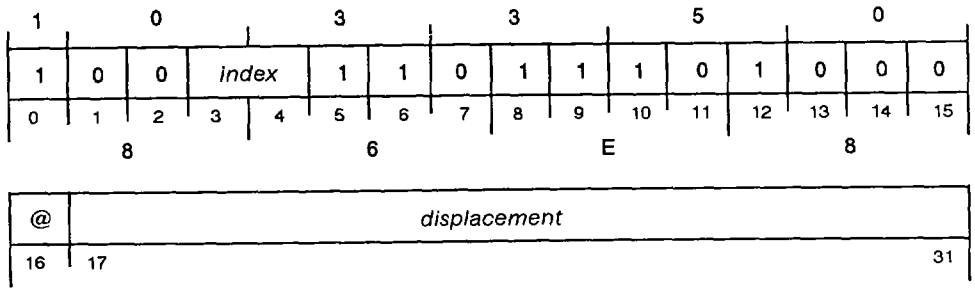
Example

```
FSS   2,0          ;Subtracts the contents of FPAC2 from FPAC0
....               ;and returns the result to FPAC0.
```

Store Floating-Point Status FSST

ECLIPSE Instruction

FSST [@]*displacement*[,*index*]



Function: FPSR -> (E)

Parameters: None

FSST stores the contents of the narrow floating-point status register into two sequential 16-bit memory locations, with the beginning address specified by the arguments.

The contents of the floating-point status register are stored as follows:

FPSR(0-15) comprise memory(0-15) to be stored

If ANY = 0, memory(16-31) will be undefined

If ANY = 1, FPSR(49-63) comprise memory(16-31) to be stored

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

FPAC0-FPAC3	Unused
PC	PC + 2
FPSR	Unchanged
Stack	Unchanged

Related Instructions

LFSST Store Floating-Point Status (Long Displacement)

Exceptions

FSST will not store a value with any combination of bit 5 (TE) and bits 1-4 concurrently set.

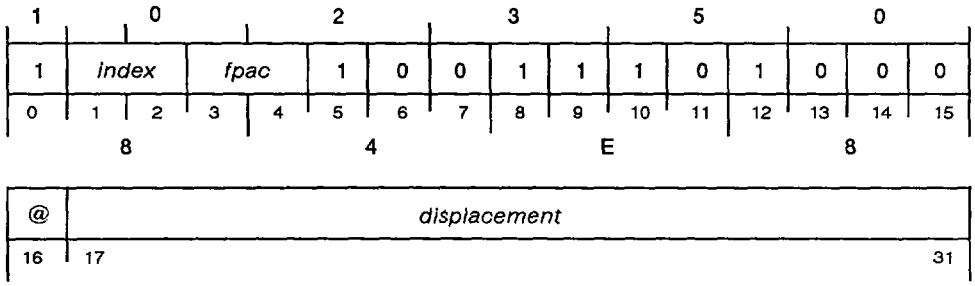
Example

```
FSST   STATUS      ;Store the narrow FPSR as a double word
                   ;at memory location STATUS.
```

Store Floating-Point Double FSTD

ECLIPSE Instruction

FSTD *fpac*,[@]*displacement*[,*index*]



Function: *fpac* -> (E)

Parameters: None

FSTD stores the contents of the specified floating-point accumulator into four sequential 16-bit memory locations, with the beginning address specified by the arguments. Unnormalized data is moved without change.

Arguments

fpac Before execution, contains 64-bit floating-point number.

After execution, contents unchanged.

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be individually specified as *fpac*; otherwise unused.

PC PC + 2

FPSR Unchanged

Stack Unchanged

Related Instructions

XFSTD, LFSTD, FSTS, XFSTS, LFSTS Store the contents of a floating-point accumulator into memory.

Exceptions

None

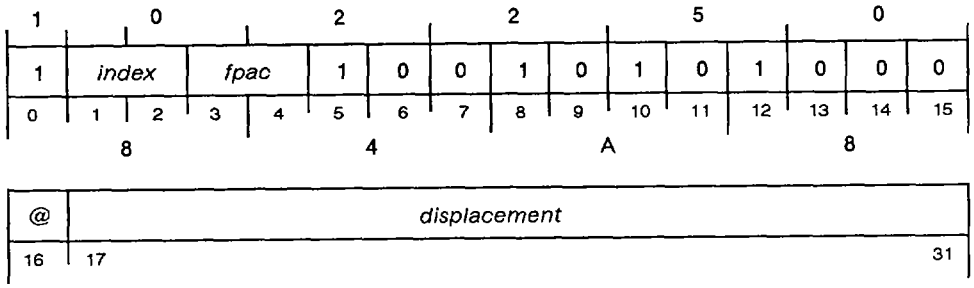
Example

```
FAD   2,3          ;Add FPAC2 to FPAC3, and store the double
FSTD  3,RESULT     ;precision result at memory location RESULT.
```

Store Floating-Point Single FSTS

ECLIPSE Instruction

FSTS *fpac*,[@]*displacement*[,*index*]



Function: *fpac* -> (E)

Parameters: None

FSTS stores the 32-bit floating-point number in the specified floating-point accumulator into two sequential 16-bit memory locations, with the beginning address specified by the arguments. Unnormalized data is moved without change.

Arguments

fpac(0-31) Before execution, contains a 32-bit floating-point number.

After execution, contents unchanged.

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
CARRY	Unchanged
PC	PC + 2
FPSR	Unchanged
Stack	Unchanged

Related Instructions

XFSTS, LFSTS, Store the contents of a floating-point accumulator into memory.
FSTD, XFSTD, LFSTD

Exceptions

None

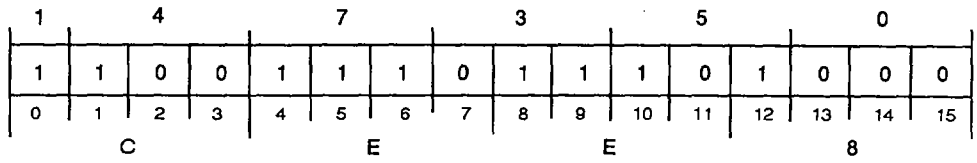
Example

```
FMS    1,2          ;Multiply FPAC1 by FPAC2, and store the single
FSTS   2,RESULT     ;precision result at memory location RESULT.
```

Trap Disable FTD

ECLIPSE Instruction

FTD



Function: 0 -> FPSR(TE)

Parameters: None

FTD sets the Trap Enable (TE) bit of the FPSR to 0.

Arguments

None

Registers, Flags, and Stacks

CARRY	Unchanged
FPAC0–FPAC3	Unused
FPSR	TE set to 0
PC	PC + 1
Stack	Unchanged

Related Instructions

FTE	Trap Enable
IORST	The I/O Reset instruction also sets TE to 0

Exceptions

None

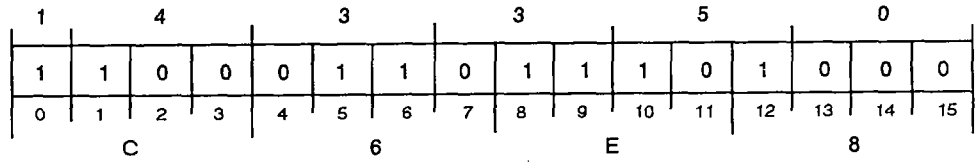
Example

```
FTD             ;Disable floating-point traps before dividing
FDD   3,2       ;FPAC2 by FPAC3, so any divide errors do not
                ;cause a floating-point fault.
```

Trap Enable FTE

ECLIPSE Instruction

FTE



Function: 1 -> FPSR(TE)

Parameters: None

FTE sets the Trap Enable (TE) bit of the FPSR to 1.

Arguments

None

Registers, Flags, and Stacks

CARRY	Unchanged
FPAC0–FPAC3	Unused
FPSR	TE set to 1
PC	PC + 1
Stack	Unchanged

Related Instructions

FTD	Trap Disable

Exceptions

If FPSR(ANY) is 1 before execution, **FTE** signals a floating-point trap. If FPSR(ANY) is 0 before execution, execution continues normally at the end of **FTE**.

When **FTE** is used to cause a floating-point trap, the floating-point program counter (FPPC) portion of the FPSR will contain the address of the first instruction to cause a fault. Even if another instruction causes a second fault before **FTE** executes, the FPPC will still contain the address of the first instruction that caused a fault.

When a floating-point fault occurs and TE is 1, the processor sets TE to 0 before transferring control to the floating-point error handler. TE should be set to 1 before resuming normal processing.

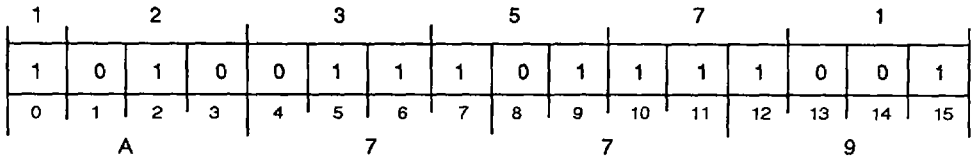
Example

```
FTE             ;Enable floating-point traps before
FMD   0,1       ;multiplying FPAC1 by FPAC0, so any multiply
                ;errors will cause a floating-point fault.
```

Fixed-Point Trap Disable FXTD

ECLIPSE Instruction

FXTD



Function: 0 -> OVK

Parameters: None

FXTD unconditionally sets the processor status register OVK and OVR flags to zero. This disables fixed-point overflow traps.

Arguments

None

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 1
PSR	OVK and OVR flags set to 0
Stack	Unchanged

Related Instructions

FXTE	Fixed-Point Trap Enable

Exceptions

None

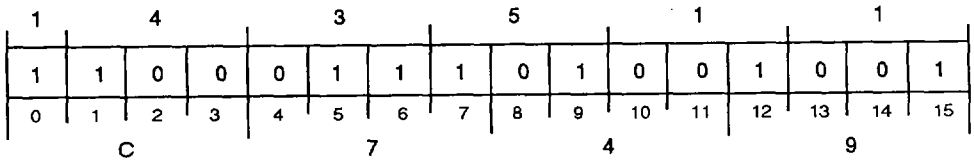
Example

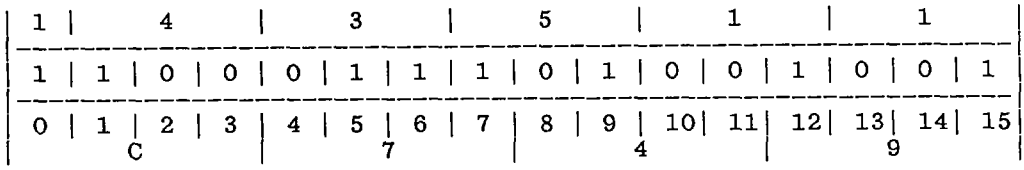
```
FXTD            ;Disable fixed-point traps before adding AC0
WADD  0,1       ;to AC1, so an overflow condition will not
                ;cause a fixed-point fault.
```

Fixed-Point Trap Enable FXTE

ECLIPSE Instruction

FXTE





Function: 1 -> OVK ; 0 -> OVR

Parameters: None

FXTE unconditionally sets the processor status register flags OVK to 1 and OVR to 0. This enables fixed-point overflow traps.

Arguments

None

Registers, Flags, and Stacks

AC0-AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 1
PSR	OVR set to 0; OVK set to 1
Stack	Unchanged

Related Instructions

FXTD	Fixed-Point Trap Disable

Exceptions

None

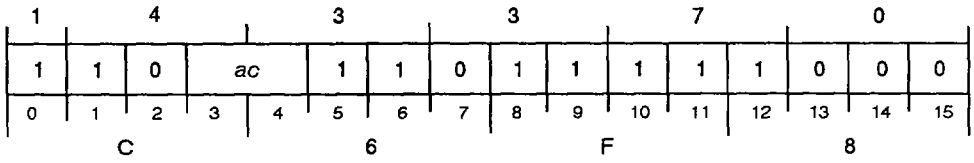
Example

```
FXTE            ;Enable fixed-point traps before multiplying
WMUL  1,2       ;AC2 by AC1, so an overflow condition will
                ;cause a fixed-point fault.
```

Halve HLV

ECLIPSE Instruction

HLV *ac*



Function: *ac* / 2 -> *ac*

Parameters: None

NOTE: HLV rounds toward 0

HLV divides the contents of an accumulator by two and rounds the result toward zero.

Arguments

ac(16–31) Before execution, contains signed 16–bit integer.

After execution, contains result.

Registers, Flags, and Stacks

AC0–AC3 Can be individually specified as *acs* and *acd*; otherwise unused.

CARRY Unchanged

Overflow 0

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

WHLV Wide Halve

Exceptions

None

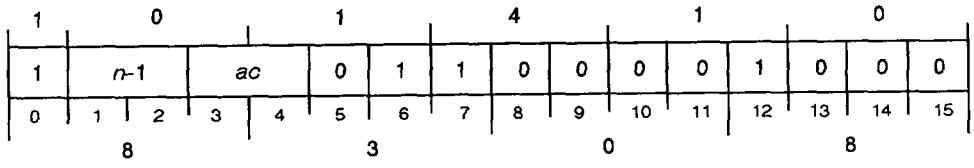
Example

```
HLV   3          ;Add one half of AC3 to AC1, leaving the
WADD  3,1        ;result in AC1.
```

Hex Shift Left HXL

ECLIPSE Instruction

HXL *n*,*ac*



Function: shift *ac* left (*n**4) -> *ac*

Parameters: None

HXL shifts the contents of the specified accumulator left a number of hex digits according to the immediate field *n*. Bits shifted out are lost, and the vacated bit positions are filled with zeros.

Arguments

n	Integer in range 1–4. If 4, then bits 16–31 of *ac* shifted out and set to 0. Assembler takes coded value of *n* and subtracts one from it before placing it in immediate field. Thus, programmer should code exact number of hex digits to be shifted.
ac(16–31)	Before execution, contains 16-bit value. After execution, contains result.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

HXR	Hex Shift Right
DHXL	Double Hex Shift Left
DHXR	Double Hex Shift Right

Exceptions

None

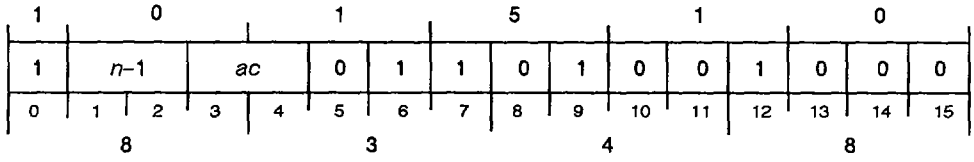
Example

```
ADC   1,1          ;AC1 is all ones.
HXL   1,1          ;Shift left 1 hex digit.
                   ;AC1[16-31] is now 177760₈.
```

Hex Shift Right HXR

ECLIPSE Instruction

HXR *n,ac*



Function: shift *ac* right (*n**4) -> *ac*

Parameters: None

HXR shifts the contents of the specified accumulator right a number of hex digits depending upon the immediate field, *n*. Bits shifted out are lost and the vacated bit positions are filled with zeros.

Arguments

n	Integer in range 1–4. If *n* is equal to 4, bits 16–31 of *ac* shifted out and set to 0. Assembler takes coded value of *n* and subtracts one from it before placing it in immediate field. Thus, programmer should code exact number of hex digits to be shifted.
ac(16–31)	Before execution, contains 16-bit value. After execution, contains result.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PSR	Unchanged
PC	PC + 1
Stack	Unchanged

Related Instructions

HXL	Hex Shift Left
DHXL	Double Hex Shift Left
DHXR	Double Hex Shift Right

Exceptions

None

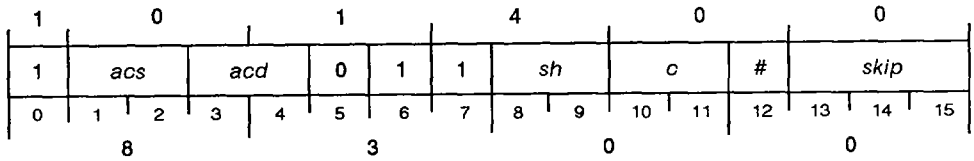
Example

```
ADC   1,1          ;AC1 is all ones.
HXR   2,1          ;Shift right 2 hex digits.
                   ;AC1[16-31] is now 000377₈.
```

Increment INC

ECLIPSE Instruction

INC[*c*][*sh*][*#*] *acs*,*acd*[,*skip*]
(*skip* false return)
(*skip* true return)



Function: *acs* + 1 -> *acd*

Parameters: None

NOTE: If $\overline{acs} > 177777(8)$, CRY -> CRY

INC initializes CARRY to specified value. Increments unsigned 16-bit integer in *acs* by one and places result in shifter. Performs specified shift operation, and loads result of shift into *acd* if no-load bit is 0. If skip condition is true, next sequential word is skipped.

Arguments

[*c*] Processor determines effect of CARRY flag (*c*) on initial value of CARRY before performing operation (opcode). Following table gives values of *c* and bits 10 and 11 and specifies operation.

Symbol [*c*]	Bits 10-11	Operation
omitted	0 0	Leave CARRY unchanged
Z	0 1	Initialize CARRY to 0
O	1 0	Initialize CARRY to 1
C	1 1	Complement CARRY

[*sh*] Processor shifts CARRY flag and 16 data bits after performing operation. Processor can shift bits left or right one bit position or can swap the two bytes. Following table gives values of *sh* and bits 8 and 9 and specifies shift operation.

Symbol [*sh*]	Bits 8-9	Shift Operation
omitted	0 0	Do not shift the result
L	0 1	Shift left
R	1 0	Shift right
S	1 1	Swap the two 8-bit bytes

[*#*] Except with no-load option (#), processor loads result of shift operation into destination accumulator. No-load option can test result of operation without destroying destination accumulator contents. Following table gives values of no-load option and bit 12 and specifies operation.

Symbol [#]	Bit 12	Operation
omitted	0	Load result into *acd*
#	1	Do not load result; restore initial CARRY flag

acs(16-31) Before execution, contains unsigned 16-bit integer.

After execution, contents unchanged unless *acs* and *acd* are same accumulator.

acd(16-31) Before execution, contains unsigned 16-bit integer.

After execution, contains result if no-load bit (#) is 0.

[*skip*] Processor skips next instruction if condition test true. Following table gives test conditions as specified by bits 13 to 15 and specifies operation.

Symbol [*skip*]	Bits 13-15	Operation
omitted	0 0 0	Never skip
SKP	0 0 1	Always skip
SZC	0 1 0	Skip if CARRY is 0
SNC	0 1 1	Skip if CARRY is not 0
SZR	1 0 0	Skip if result is 0
SNR	1 0 1	Skip if result is not 0
SEZ	1 1 0	Skip if either CARRY or result 0
SBN	1 1 1	Skip if both CARRY and result not 0

Skip omits next sequential 16-bit word. Make sure that *skip* does not transfer control to point within 32-bit or longer instruction.

Registers, Flags, and Stacks

AC0-AC3 Can be individually specified as *acs* and *acd*; otherwise unused.

CARRY If number is *acs* is 177777_8, initial CARRY complemented. Then, if left or right shift occurs, final resulting CARRY is bit shifted into CARRY.

Overflow 0

PC PC + 1 (false exit)
PC + 2 (true exit)

PSR Unchanged

Stacks Unchanged

Related Instructions

ADI, WADI, NADI, WNADI, XNADI, XWADI, LNADI, LWADI Add an immediate value to an accumulator or memory.

ISZ, EISZ, XNISZ, XWISZ, LNISZ, LWISZ Add one to memory and skip if result is zero.

WINC Wide Increment

Exceptions

If the incrementation produces a result that is greater than 32,768, the instruction complements CARRY. (See also CARRY)

Do not specify **INC** with no-load option (#) in combination with either never skip or always skip option. Instruction may not end in 1000_2 or 1001_2 (reserved for other instructions).

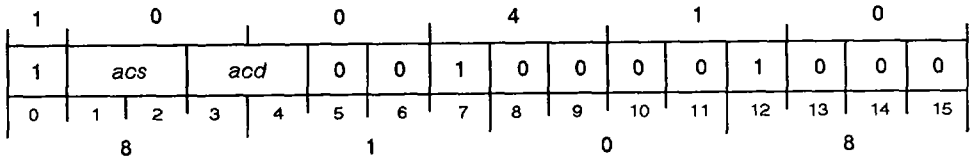
Example

```
INC 2,2          ;Increments the contents of AC2 by one and
....             ;returns the result to AC2.
```

Inclusive OR IOR

ECLIPSE Instruction

IOR *acs,acd*



Function: *acs* OR *acd* -> *acd*

Parameters: None

IOR forms the logical Inclusive OR of the contents of *acs* and *acd* and places the result in *acd*. Sets a bit position in the result to 1 if the corresponding bit position in one or both operands contains a 1; otherwise, the instruction sets the result bit to 0.

Arguments

acs(16-31)	Before execution, contains 16-bit value.
	After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd(16-31)	Before execution, contains 16-bit value.
	After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

WIOR	Wide Inclusive OR
XOR	Exclusive OR
WXOR	Wide Exclusive OR

Exceptions

None

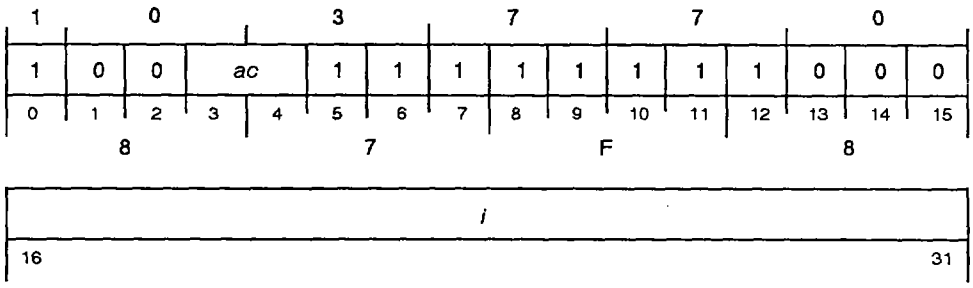
Example

```
IOR   2,0          ;Logically ORs the contents of AC2 and AC0 and
....               ;returns the result to AC0.
```

Inclusive OR Immediate IORI

ECLIPSE Instruction

IORI *i,ac*



Function: *i* OR *ac* -> *ac*

Parameters: None

IORI inclusively ORs the contents of the specified accumulator with the contents of the immediate field, placing the result in the specified accumulator.

Arguments

i	16-bit immediate value
ac(16-31)	Before execution, contains 16-bit value.
	After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

WIORI Wide Inclusive OR Immediate

Exceptions

None

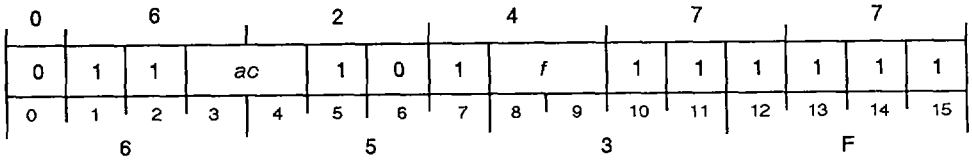
Example

```
IORI   77,0        ;Logically ORs the contents of AC0 with the
....               ;immediate value 77(8) and returns the result
                   ;to AC0.
```

I/O Reset IORST

ECLIPSE Instruction

IORST



Function: 0 -> priority mask of all IOCs
off -> Address Translator
0-> PSR
0 -> priority mask
0-> FPSR (0-9)
[*f*] -> ION flag
0-> BUSY, DONE flags

Parameters: None

NOTE: **IORST = DICC 0,CPU**

IORST resets the I/O system by sending a reset signal to all I/O devices on all I/O channels to clear their states. It sets the BUSY, DONE, and ION (Interrupt On) flags to 0, clears the 16-bit interrupt priority mask, setting it to all zeros, and disables data channel mapping.

The I/O Reset function may also be implemented with a **DIC**[*f*] *ac*,**CPU** instruction. The **DIC CPU** mnemonic sets the ION flag according to the function specified by *f* and sets the 16-bit priority mask to all zeros. When using the **DIC CPU** mnemonic, an accumulator must be specified to avoid an assembly error, even though the accumulator is ignored.

NOTE: *In multi-I/O channel environments,* **IORST** *sets the I/O channel mask register flag for 0 to 0, and sets bits 0, 3, 4, 7, 8, 9, and 14 of the I/O channel definition register of all IOCs (6000_8) to 0.*

In multiple-CPU systems, **IORST** *also resets IMODE to 0, clears all pending cross interrupts, and redirects all IOC traffic to the CPU that issued the* **IORST** *instruction.*

Arguments (DIC CPU only)

ac	Must be specified but is not used.
CPU	Device code 77_8 mnemonic.
f	Specify from **S**, **C**, and **P** for desired ION function.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; but unused.
BUSY, DONE flags	0
CARRY	Unchanged
FPSR	All 0
ION flag	0 or [*f*]
Overflow	Unaffected
PC	PC + 1

PSR	All 0
Stack	Unchanged

Related Instructions

DICC 0,CPU	Data General assemblers recognize **IORST** as equivalent to **DICC 0,CPU**.

Exceptions

None

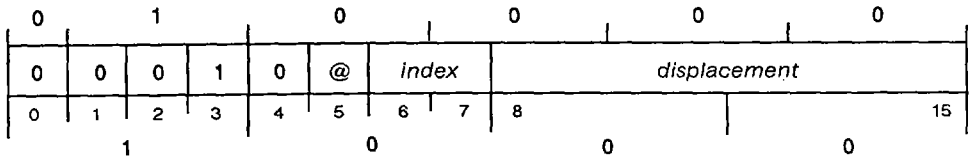
Example

```
IORST                   ;First instruction of operating system.
```

Increment and Skip if Zero — ISZ

ECLIPSE Instruction

ISZ [@]*displacement*[,*index*]
(result ≠ 0 return)
(result = 0 return)



Function: (E) + 1 -> (E)
If resulting (E) = 0 then skip

Parameters: None

ISZ increments an unsigned 16-bit integer in memory by 1, writes the result back into the location, and skips the next sequential instruction if the result is zero. This instruction is indivisible.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

AC0-AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1 (result ≠ 0) PC + 2 (result = 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

EISZ, XNISZ, XWISZ, LNISZ, LWISZ Increment the contents of memory and skip if result is zero.

Exceptions

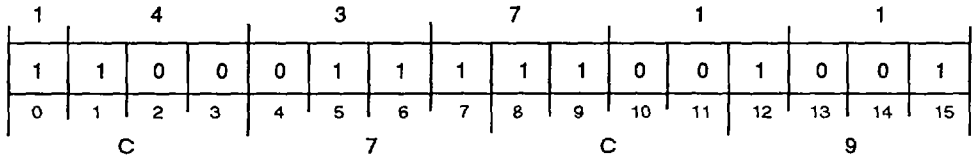
None

Example

```
    LDA    0,NFIVE      ;Get a constant -5.
    STA    0,COUNTER    ;Initialize the loop counter.
LOOP:      . . .        ;Beginning of loop.....
           ISZ COUNTER  ;Increment counter and skip if zero.
           JMP LOOP     ;We're not done yet.
           . . .        ;We did the loop 5 times.....
NFIVE:     .WORD -5     ;Constant -5.
COUNTER:   .WORD 0      ;Counter variable.
```

Increment the Word Addressed by WSP and Skip if 0 — ISZTS

ISZTS
(addressed word ≠ 0 return)
(addressed word = 0 return)



Function: (wsp) +1 -> (wsp)
If resulting (wsp) = 0 then skip

Parameters: None

ISZTS increments the unsigned 32-bit integer addressed by the wide stack pointer and skips the next 16-bit word if the incremented value is zero.

The operation performed by **ISZTS** is not indivisible.

Arguments

None

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1 (addressed word ≠ 0) PC + 2 (addressed word = 0)
PSR	Unchanged
Stack	WSP remains unchanged.

Related Instructions

DSZTS	Decrement the Word Addressed by WSP and Skip if Zero

Exceptions

None

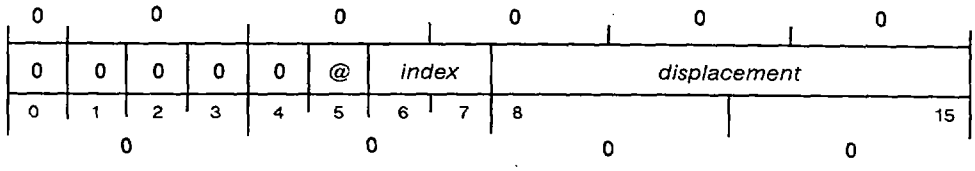
Example

```
    ;Subroutine to compare two strings
    ;
    ;Strings are assumed to be word aligned and to be followed by a
    ;terminating null (or two, if needed to fill a word).
    ;
    ;AC0 = Byte length of string (without terminator)
    ;AC1 = Word pointer to first string
    ;AC2 = Word pointer to second string
    ;
    ;Returns +1 if they match, 0 if they don't
CMPAR:  WPSH        3,3         ;Save return address
        LDAFP       3           ;Get frame pointer
        WINC        0,0         ;Get number of words with
                    ;terminator
        WINC        0,0         ;Number of characters plus 1
        WHLV        0           ;Number of characters plus 1 / 2
        XNSTA       0,WCNT,3    ;Save count
        XWSTA       1,WPTR.W,3  ;Save one of the pointers
CMPLP:  XNLDA       0,0,2       ;Pick up a word
        XNLDA       1,@WPTR.W,3 ;and its friend
        WSEQ        0,1         ;See if equal
            WPOPJ   ;No, return false (0)
        XWISZ       WPTR.W,3    ;Move to next word
        WINC        2,2         ;(S)
        XNDSZ       WCNT,3      ;See if done
        WBR CMPLP   ;
        ISZTS                   ;Bump return (they match)
        WPOPJ                   ;Return
```

Jump JMP

ECLIPSE Instruction

JMP [@]*displacement*[,*index*]



Function: E -> PC

Parameters: None

JMP loads an effective address derived from the arguments into the program counter. Program execution continues with the instruction addressed by the updated value of the program counter.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	Effective address resolved from arguments.
PSR	Unchanged
Stack	Unchanged

Related Instructions

WBR	Wide Branch
EJMP, XJMP, LJMP	Load an effective address into the program counter.

Exceptions

None

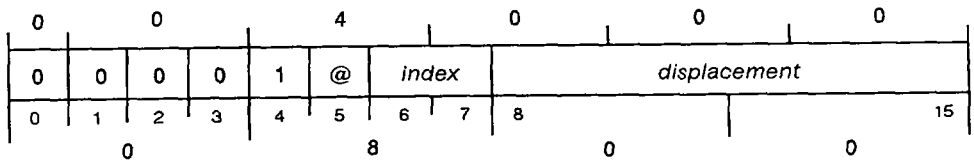
Example

```
    JMP   CLOSE       ;Jump to a location that is close by and
    .                 ;hence does not require an extended
    .                 ;displacement.
    .
CLOSE:
          . . .
```

Jump to Subroutine JSR

ECLIPSE Instruction

JSR [@]*displacement*[,*index*]



Function: E -> PC
PC+1 -> AC3

Parameters: None

JSR increments the program counter by 1 (to address the next sequential instruction), stores this value in AC3, and loads the effective address derived from the arguments into the program counter. Program execution continues with the instruction addressed by the updated value of the program counter. The effective address is calculated before the incremented value of the program counter is stored.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

AC0–AC2	Unused
AC3	PC(before execution) + 1
CARRY	Unchanged
Overflow	0
PC	Effective address resolved from arguments.
PSR	Unchanged
Stack	Unchanged

Related Instructions

EJSR, XJSR, LJSR	Jump to a subroutine.

Exceptions

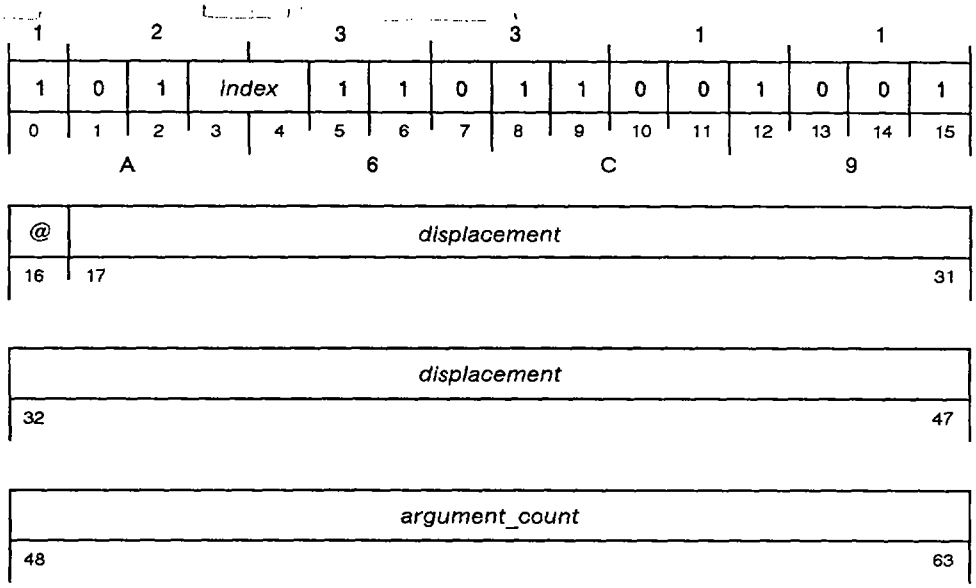
None

Example

```
    JSR    SUBROUT      ;Jump to subroutine. Return PC is put in AC3....
SUBROUT:   .             ;Do the subroutine, but don't modify contents of
           .             ;AC3 because it has the return address.
           JMP 0,3       ;Go back to the caller. ACs are not
                         ;necessarily restored.
```

Call Subroutine (Long Displacement) LCALL

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



Function: PC + 4 -> AC3
If E = valid, E -> PC
0 -> OVR

Parameters: None

NOTE: Valid E = inward ring cross and legal gate. If *argument_count* bit 0 = 0, PSR and *argument_count* pushed onto stack.

LCALL transfers program control to a subroutine in the current segment or through a gate array in a lower-numbered segment to a subroutine in that lower-numbered segment.

LCALL loads the program counter, plus four, into AC3. If the effective address (target address) is legal, the instruction checks the *argument count* field. **LCALL** then sets PSR(OVR) to zero and loads the program counter with the target address.

For more information on **LCALL**, refer to "Transferring Program Control to Another Segment" in the Program Flow Management chapter.

Arguments

[@]*displacement*[,*index*]
Effective address (target address) generated by instruction may specify the current ring, an inner ring, or an outer ring.

If target address specifies an *outward ring* crossing, protection fault occurs and error code 7 placed in AC1. PC in return block undefined.

If target address specifies an *inward ring* call, **LCALL** assumes target address has following format:
Bits 1-3 contain new segment number.
Bits 4-15 unused.
Bit 16 must contain 0 or results undefined.
Bits 17-31 contain gate number in inner ring.

If gate is illegal, a protection fault occurs, error code 6 placed in AC1, and no subroutine call made. PC in return block undefined.

If gate legal, or target address specifies the *current ring*, **LCALL** then checks *argument count* field.

[*argument_count*]

Contains 16-bit value specifying number of arguments pushed onto stack. **LCALL** creates a PSR/*argument count* double word depending on value of high bit (bit 48) of *argument count*.

If high bit is 0, **LCALL** pushes onto the wide stack a double word with the following format:

Bits 0-15 contain current PSR.
Bits 16-31 contain *argument count*.

If high bit is 1 (negative), **LCALL** assumes top double word of wide stack has following format:

Bits 0-15 undefined.
Bits 16-31 contain *argument count* with bit 16 = 0.

The instruction uses the wide stack double word and ignores *argument count* coded with **LCALL** instruction. The instruction then places current PSR into bits 0-15 of stack double word.

(If target address is in inner segment, **LCALL** copies the number of double words specified in *argument count* from the outer segment stack to the inner segment stack, and then pushes the PSR/*argument count* double word onto inner stack.)

Registers, Flags, and Stacks

AC0-AC2	Unused
AC3	After execution, contains PC + 4 (always references current segment)
CARRY	Unchanged
Overflow	Unaffected
PC	Target address
PSR	OVR set to 0
Stack	Wide stack in current segment contains arguments. If target address is current segment, wide stack also contains PSR/*argument count* double word. If target address is inner segment, inner segment wide stack contains copy of arguments and PSR/*argument count* double word.

Related Instructions

WRTN	Pops six double words from wide stack. Generally, should be the last instruction in the subroutine.
WSAVR, WSAVS	Push five double words onto the wide stack. Generally, should be the first instruction in the subroutine.

Exceptions

If the target address specifies an outward ring crossing, or an inward ring call with an illegal gate, a protection fault occurs. The processor pushes a fault–return block onto the wide stack in the current segment (PC contents are undefined), loads AC1 with an error code, and transfers program control to the protection violation fault handler.

If a wide stack overflow occurs while **LCALL** is pushing the PSR/*argument count* double word, a stack overflow occurs. The processor clears the PSR, pushes a fault return block (PC contents are undefined) onto the wide stack in the destination segment, loads AC1 with an error code, and transfers program control to the wide stack fault handler in the destination segment.

The error codes returned to AC1 are:

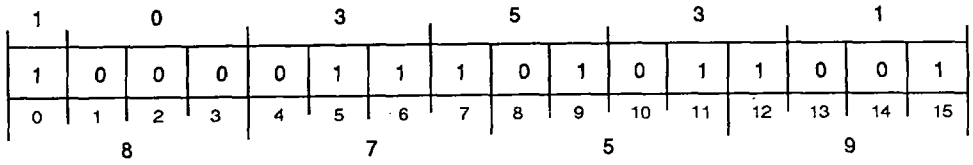
Error Code	Description	PC
2	Wide stack overflow	Wide stack fault handler
3	Invalid segment	Protection violation fault handler
6	Invalid gate	Protection violation fault handler
7	Illegal outward call	Protection violation fault handler

Example

```
        LCALL  4S3+2,0,6    ;LCALL transfers program control to segment 4
                            ;through the second element in the gate array.
                            ;(Second element contains the address of
                            ;INET.) LCALL passes 6 arguments to the
                            ;subroutine.
INET:   WSAVS 5
        .
        .
        .
        WRTN
```

Load CPU Identification LCPID

LCPID



Function: CPU id -> AC0(model number [bits 0-15], microcode revision [bits 16-23], memory size [bits 24-31])

Parameters: None

LCPID loads a double-word of CPU identification into AC0.

Arguments

None

Registers, Flags, and Stacks

AC0 Receives CPU identification information as follows:

Bits	Contents
0-15	model number
16-23	microcode revision
24-31	memory size

AC1–AC3 Unused

CARRY Unchanged

Overflow 0

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

ECLID, NCLID Return CPU identification information.

Exceptions

None

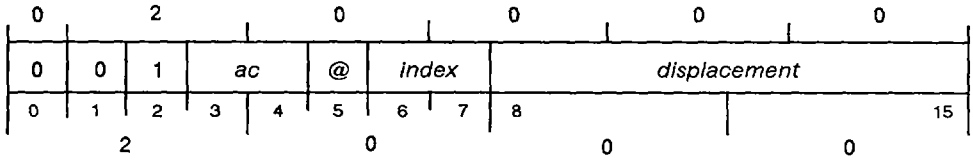
Example

```
LCPID              ;Load the CPU identification into AC0.
XWSTA 0,CPUID      ;Store the CPU id in memory.
.
.
.
CPUID: .DWORD 0    ;Variable for CPU id.
```

Load Accumulator LDA

ECLIPSE Instruction

LDA *ac*,[@]*displacement*[,*index*]



Function: (E) -> *ac*

Parameters: None

LDA loads a word from a memory location into an accumulator.

Arguments

ac(16–31) After execution, contains word from memory location.

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

ELDA, XNLDA, Load an accumulator with the contents of memory.
XWLDA, LNLDA, LWLDA

Exceptions

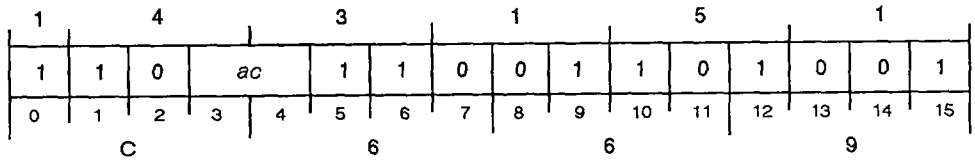
None

Example

```
    LDA   1,COUNT      ;Get the counter value into AC1.
     .
     .
     .
COUNT:    .WORD 0      ;Counter value.
```

Load Accumulator with WFP — LDAFP

LDAFP *ac*



Function: wfp -> *ac*

Parameters: None

LDAFP loads the specified accumulator with the contents of wide frame pointer.

Arguments

ac	After execution, contains wide frame pointer.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

LDASB, LDASL, LDASP	Load wide stack parameters into an accumulator.

Exceptions

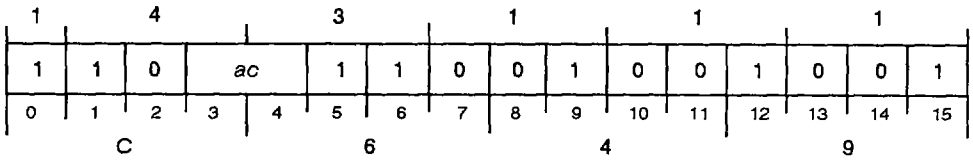
None

Example

```
LDAFP 0             ;Get the wide frame pointer.
XWSTA 0,SAVE_FP     ;Save its value in memory.
```

Load Accumulator with WSB LDASB

LDASB *ac*



Function: wsb -> *ac*

Parameters: None

LDASB loads the specified accumulator with the contents of the wide stack base.

Arguments

ac	After execution, contains wide stack base.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

LDAFP, LDASL, LDASP Load wide stack parameters into an accumulator.

Exceptions

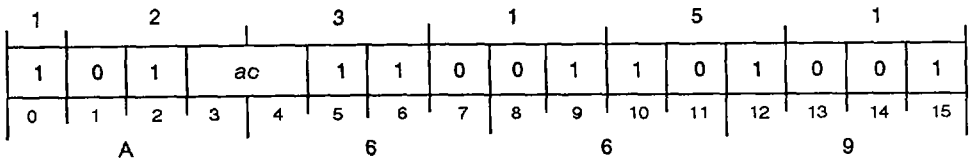
None

Example

```
LDASB 0             ;Get the wide stack base.
XWSTA 0,SAVE_SB     ;Save its value in memory.
```

Load Accumulator with WSL LDASL

LDASL *ac*



Function: wsl -> *ac*

Parameters: None

LDASL loads the specified accumulator with the contents of the wide stack limit.

Arguments

ac After execution, contains wide stack limit.

Registers, Flags, and Stacks

AC0-AC3 Can be specified as *ac*; otherwise unused.

CARRY Unchanged

Overflow 0

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

LDAFP, LDASB, LDASP Load wide stack parameters into an accumulator.

Exceptions

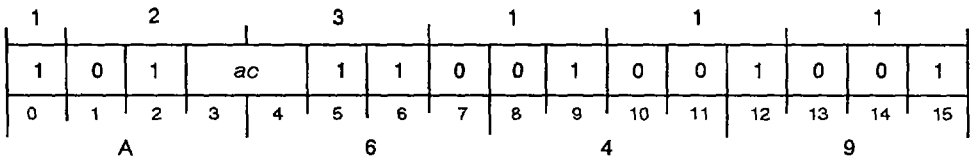
None

Example

```
LDASL 0             ;Get the wide stack limit.
XWSTA 0,SAVE_SL     ;Save its value in memory.
```

Load Accumulator with WSP LDASP

LDASP *ac*



Function: wsp -> *ac*

Parameters: None

LDASP loads the specified accumulator with the contents of the wide stack pointer.

Arguments

ac	After execution, contains wide stack pointer.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

LDAFP, LDASB, LDASL Load wide stack parameters into an accumulator.

Exceptions

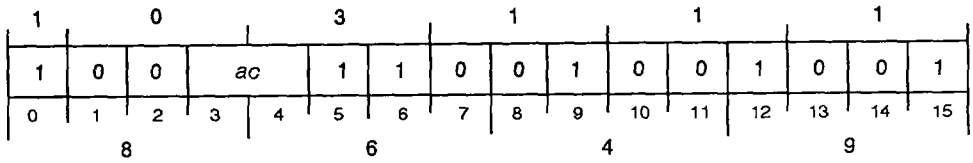
None

Example

```
LDASP 0             ;Get the wide stack pointer.
XWSTA 0,SAVE_SP     ;Save its value in memory.
```

Load Accumulator with Double Word LDATS

LDATS *ac*



Function: (wsp) -> *ac*

Parameters: None

LDATS uses the contents of the wide stack pointer as the address of a double word. It loads the contents of the addressed double word into the specified accumulator.

Arguments

ac	After execution, contains 32-bit result.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

STATS	Store Accumulator into Stack Pointer Contents

Exceptions

None

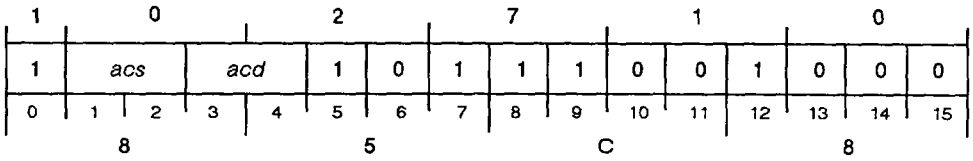
Example

```
WPSH  1,1          ;Push AC1 onto the stack.
.
.
.
LDATS 0            ;Get the pushed value of AC1 into AC0 without
                   ;popping the value off the stack.
```

Load Byte LDB

ECLIPSE Instruction

LDB *acs,acd*



Function: (E)byte -> *acd*[bits 24-31, bits 16-23 set to 0]

Parameters: *acs* = byte pointer -> unchanged

LDB copies a byte from a specified memory location into *acd*.

Arguments

acs(16-31) Before execution, contains byte address for fetch from memory. Effective address generated by instruction confined to first 64 Kbytes of current segment.

After execution, contents unchanged.

acd(24-31) After execution, contains byte from memory with bits 16-23 set to 0.

Registers, Flags, and Stacks

AC0-AC3 Can be individually specified as *acs* and *acd*; otherwise unused.

CARRY Unchanged

Overflow 0

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

ELDB, XLDB, LLDB, WLDB Load a byte from memory into an accumulator.

Exceptions

None

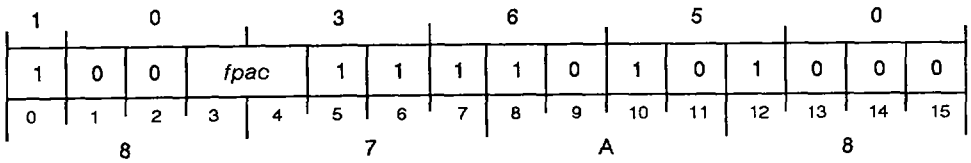
Example

```
    ELEF   2,(BYTE_PAIR*2)+1        ;Get byte address of low order byte.
    LDB    2,0                      ;Load AC0 with the low order byte
    .                               ;from the word.
    .
    .
BYTE_PAIR:
          .WORD 0                   ;Location containing a pair of bytes.
```

Load Integer LDI

ECLIPSE Instruction

LDI *fpac*



Function: @(AC3)[decimal #] -> *fpac*[norm fp#]
AC3 -> AC2
update -> FPSR(N,Z)

Parameters: AC1 = data-type indicator -> unchanged
AC2 = x -> AC3
AC3 = byte pointer -> last byte pointer + 1

NOTE: A -0 sets *fpac* to True Zero

LDI fetches a decimal integer (up to 16 digits) from memory, translates the integer to normalized floating-point format, and loads the result into the specified floating-point accumulator.

Arguments

fpac	After execution, contains translated floating-point integer. Unused lower-order byte locations set to 0.

Registers, Flags, and Stacks

AC0	Unused
AC1(16-31)	Before execution, specifies type and length of data to be translated. **LDI** does not use the scale factor in the data type indicator.
	After execution, contents unchanged.
AC2(16-31)	After execution, contains initial value of AC3.
AC3(16-31)	Before execution, contains starting byte address for location in memory. Effective address is confined to the first 64 Kbytes of the current segment.
	After execution, contains address of first byte following integer field.
CARRY	Unchanged
FPAC0-FPAC3	Can be individually specified for *fpac*; otherwise unused.
FPSR	Updated Z and N flags.
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

LDIX	Load Integer Extended
WLDI	Wide Load Integer
WLDIX	Wide Load Integer Extended

Exceptions

An invalid decimal number produces a commercial fault, leaving the contents of *fpac* undefined.

For data type 7, numbers are not normalized in the move. By convention, the first byte of a number stored according to data type 7 must contain the sign and exponent of the floating-point number. The exponent must be in "excess 64" representation. The instruction copies each byte (following the lead byte) directly to the mantissa of the specified *fpac*. It then sets to 0 each low-order byte in *fpac* that does not receive data from memory.

For data type 7, an integer with more than 8 digits produces a commercial fault.

An attempt to load a -0 sets *fpac* to true zero.

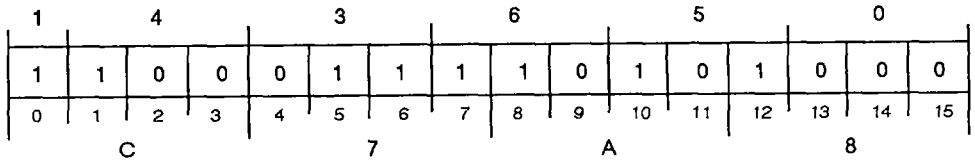
Example

```
XNLDA  1,DTYPE      ;AC1 contains the data type indicator.
XLEF   3,INTEGR     ;Word pointer to the integer field.
WADD   3,3          ;AC3 is a byte pointer to the integer field.
LDI    2            ;Convert the specified integer to floating
                    ;point, leaving the result in FPAC2.
```

Load Integer Extended LDIX

ECLIPSE Instruction

LDIX



Function: @(AC3)[decimal #] -> (FPAC0,1,2,3)[fp#]
AC3 -> AC2
? -> FPSR(N,Z)

Parameters: AC1 = data-type indicator -> unchanged
AC2 = x -> AC3
AC3 = byte pointer -> last byte pointer + 1

LDIX fetches a decimal integer from memory, expands the integer to 32 digits, divides the result into four 8-digit integers, converts the integers to floating-point numbers, and loads them into individual floating-point accumulators.

The sign of the 32-bit integer is stored in each fpac unless the integer in that fpac consists of all zeros, in which case the floating-point accumulator is set to true zero.

The integer fetched from memory must be of data type 0, 1, 2, 3, 4, or 5 and can contain up to 32 digits. The integer is expanded to 32 digits by adding zeros to the high-order bit locations.

Arguments

None

Registers, Flags, and Stacks

AC0	Unused
AC1(16-31)	Before execution, contains type and length of data to be translated. **LDIX** does not use the scale factor in the data type indicator. After execution, contents are unchanged.
AC2(16-31)	After execution, contains initial value of AC3.
AC3(16-31)	Before execution, contains starting byte address for location in memory. Effective address is confined to the first 64 Kbytes of the current segment. After execution, contains address of first byte following integer field.
CARRY	Unchanged
FPAC0	Receives first 8 (high-order) digits from extended integer.
FPAC1	Receives second 8 digits from extended integer.

FPAC2	Receives third 8 digits from extended integer.
FPAC3	Receives last 8 (low-order) digits from extended integer.
FPSR	Z and N flags unpredictable.
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

LDI	Load Integer
WLDI	Wide Load Integer
WLDIX	Wide Load Integer Extended

Exceptions

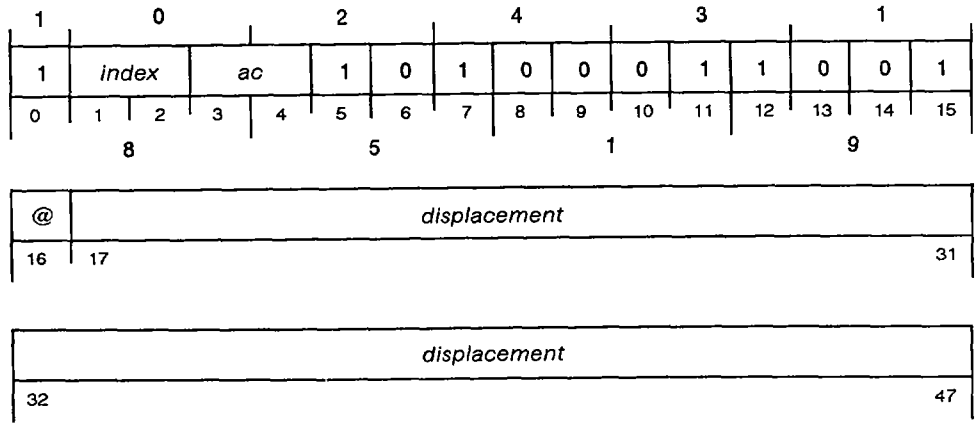
None

Example

```
XNLDA  1,DTYPE      ;AC1 contains the data type indicator.
XLEF   3,INTEGR     ;Word pointer to the integer field.
WADD   3,3          ;AC3 is a byte pointer to the integer field.
LDIX                ;Distribute the integer over the four FPACs.
```

Dispatch (Long Displacement) LDSP

LDSP *ac*,[@]*displacement*[,*index*]
(unsuccessful return)



Function:
If L <= *ac* <= H
Then If (E + 2*(#-L)) ≠ -1
Then (E + 2*(#-L)) + E+2*(#-L) -> PC
Else end
Else (LDSP) + 3 -> PC

Parameters:
ac = # -> unchanged
E = (dispatch table) -> unchanged
E-4 = L [2#] -> unchanged
E-2 = H [2#] -> unchanged
E+2*(H-L) = Last table entry -> unchanged
Dispatch table = 28-bit address offsets -> unchanged
CRY = x -> unchanged

LDSP transfers program control to a PC-relative address -- located in a dispatch table. The effective address becomes the first entry of the dispatch table. The processor uses the contents of *ac* to index to a 28-bit address in the table.

The processor compares the signed 32-bit integer in *ac* with the signed 32-bit integers in the two locations immediately preceding the table. The processor uses the second double word before the table as the lower limit (L) and the first double word before the table as the higher limit (H).

If the number in *ac* is less than the lower limit or greater than the higher limit, the processor transfers program control to the instruction following the **LDSP** instruction. Otherwise, the processor reads from the table the double word at E + 2(*ac*) - 2L.

The processor then tests this double word. If the double word equals -1 (37777777777_8), the processor transfers program control to the instruction following **LDSP**. Otherwise, the processor transfers program control by loading this double word plus the address of this double word into the program counter.

Arguments

ac Before execution, contains signed 32-bit integer for relative offset to table entry.

[@]*displacement*[,*index*]
Effective address of location of first entry in table generated by instruction confined to current segment.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	(E + 2(*ac*) – 2L) + E + 2(*ac*) – 2L (successful exit) Processor ignores bits 1–3 of table entries. PC + 3 (unsuccessful exit)
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load with immediate	Use these instructions to place a value into *ac*.

Exceptions

If the value to be placed in the program counter -- (E + 2(*ac*) – 2L) + E + 2(*ac*) – 2L -- produces an invalid address, a protection fault is generated.

Example

```
    .NREL
    XNLDA  3,SYMBL      ;Load AC3 with an ASCII character (+, -, *, /)
    LDSP   3,TABLE      ;Perform arithmetic operation based on SYMBL
SUBEM:   WNEG 2,2       ;Perform subtraction if dispatch fails
ADEMM:   WADD 2,1
         WBR OUTEM
MULEM:   WMULS
         WBR OUTEM
DIVEM:   WDIVS
OUTEM:   -
         ....
LOW:     .DWORD "*      ;52₈ -- relative lower limit
HIGH:    .DWORD "/      ;57₈ -- relative higher limit
TABLE:   .DWORD MULEM-.             ;*
         .DWORD ADEMM-.             ;+
         .DWORD -1      ;,
         .DWORD -1      ;-
         .DWORD -1      ;.
         .DWORD DIVEM-.             ;/
```

LDSP accesses the dispatch table (beginning at the TABLE address), uses SYMBL to index into the table, and transfers program control to the address in the table entry. Figure 11–3 is an example of the dispatch table.

TABLE - 4	"*	(52(8))	E - 4 (L)
TABLE - 2	"/	(57(8))	E - 2 (H)
TABLE	MULEM-.	(*)	E (table base)
	ADEMM-.	(+)	
	-1	(,)	
	-1	(-)	
	-1	(.)	
TABLE + 2(H - L)	DIVEM-.	(/)	(end of table)

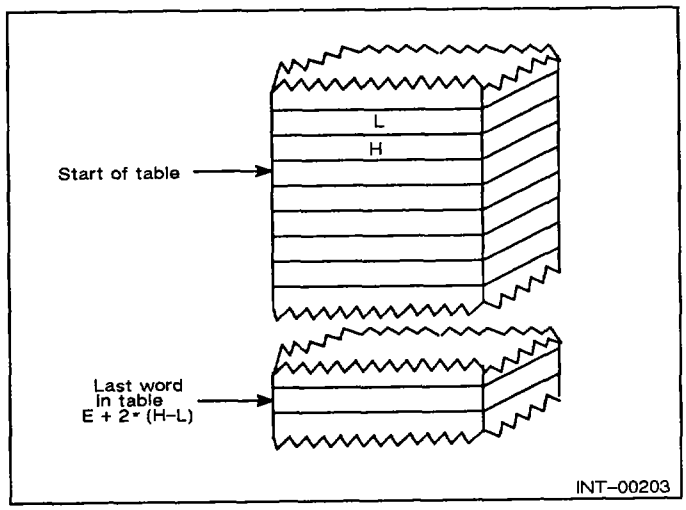
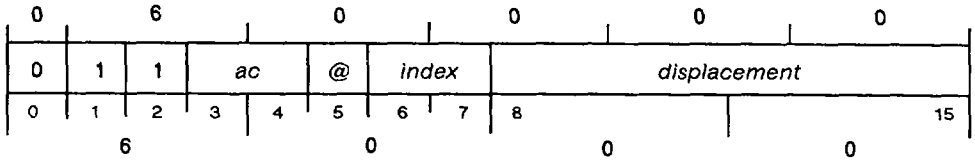


Figure 11-3 LDSP *dispatch table example*

Load Effective Address LEF

ECLIPSE Instruction

LEF *ac*,[@]*displacement*[,*index*]



Function: E -> *ac*

Parameters: None

LEF places an effective address in the specified accumulator. The instruction can only be executed when both the address translator and the LEF mode are enabled.

Arguments

ac After execution contains calculated effective address, resolved from arguments. Bit 0 set to 0; bits 1–3 define current segment; bits 4–16 all zeros; and bits 17–31 contain address.

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

ELEF, XLEF, LLEF Load an effective address into an accumulator.

Exceptions

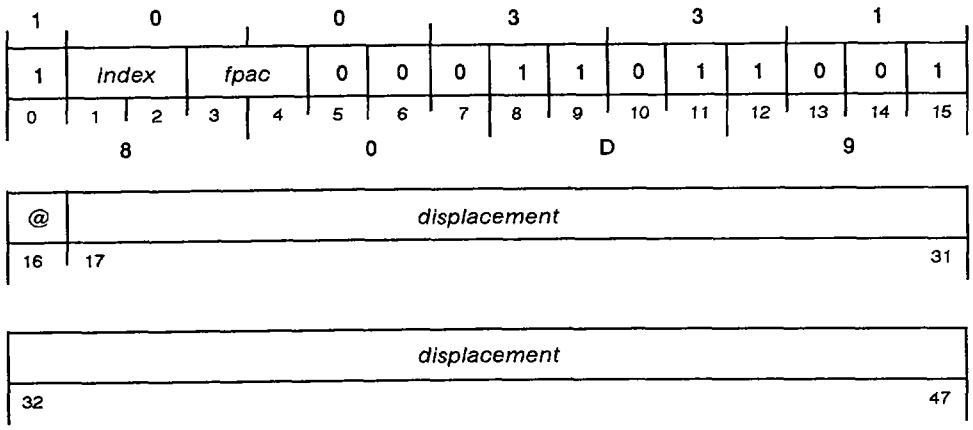
If the LEF mode is not enabled, the processor checks the I/O validity flag; if the I/O validity flag is set (I/O enabled) or the address translator is disabled, the processor executes the instruction as an I/O instruction. Otherwise, a protection violation occurs.

Example

```
LEF   2,@AWORD     ;Get starting address of array of words.
ADD   1,2          ;Add the word index from AC1.
LDA   0,0,2        ;Get the word into AC0.
```

Add Double (Memory to FPAC) (Long Displacement) LFAMD

LFAMD *fpac*,[@]*displacement*[,*index*]



Function: (E) + *fpac* -> *fpac*

Parameters: None

LFAMD adds a 64-bit floating-point number in memory to a 64-bit floating-point number in an *fpac* and places the normalized 64-bit result in the *fpac*.

Arguments

fpac Before execution, contains 64-bit floating-point number.

After execution, contains normalized 64-bit result.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
FPSR	Updated Z and N flags.
PC	PC + 3
Stack	Unchanged

Related Instructions

LFAMS	Add Single (Memory to FPAC) (Long Displacement)

Exceptions

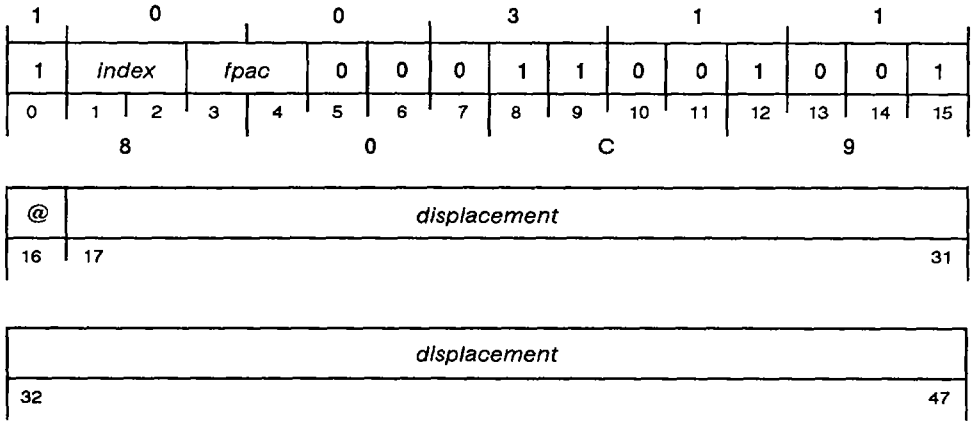
If addition produces an exponent overflow, the processor sets the FPSR(OVF) flag to 1 and terminates the instruction.

Example

```
LFLDD 2,DATA1      ;Add the two double precision floating-point
LFAMD 2,DATA2      ;numbers at memory locations DATA1 and DATA2,
LFSTD 2,RESULT     ;storing the result at memory location RESULT.
```

Add Single (Memory to FPAC) (Long Displacement) LFAMS

LFAMS *fpac*,[@]*displacement*[,*index*]



Function: (E) + *fpac* -> *fpac*

Parameters: None

LFAMS adds the 32-bit floating-point number in a memory location to a 32-bit floating-point number in an *fpac* and places the normalized 32-bit result in the *fpac*.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contains normalized 32-bit result with bits 32-63 set to 0.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
FPSR	Updated Z and N flags.
PC	PC + 3
Stack	Unchanged

Related Instructions

LFAMD	Add Double (Memory to FPAC) (Long Displacement)

Exceptions

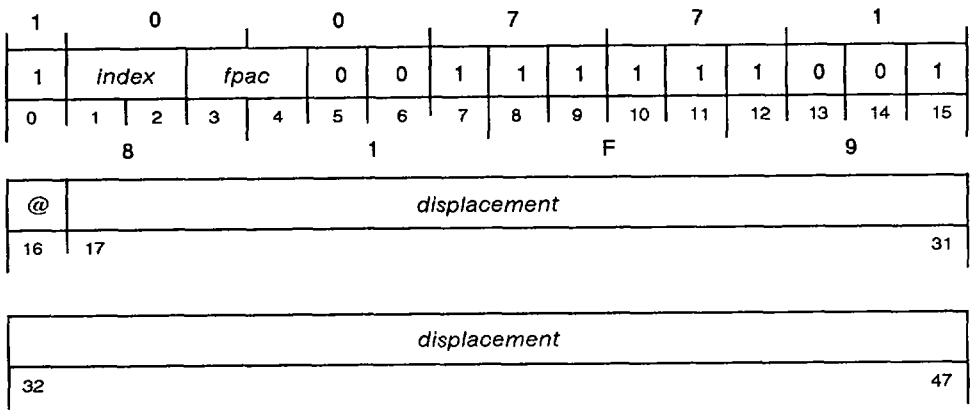
If addition produces an exponent overflow, the processor sets the FPSR(OVF) flag to 1 and terminates the instruction.

Example

```
LFLDS 2,FLPT1      ;Add the two single precision floating-point
LFAMS 2,FLPT2      ;numbers at memory locations FLPT1 and FLPT2,
LFSTS 2,RESULT     ;storing the result at memory location RESULT.
```

Divide Double (FPAC by Memory) (Long Displacement) LFDMD

LFDMD *fpac*,[@]*displacement*[,*index*]



Function: *fpac* / (E) -> *fpac*

Parameters: None

LFDMD divides the 64-bit floating-point number in the specified *fpac* by the 64-bit floating-point number in memory, and places the normalized 64-bit result in the *fpac*.

Arguments

fpac Before execution, contains 64-bit floating-point number.

After execution, contains normalized 64-bit result.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be specified as *fpac*; otherwise unused.
FPSR	Updated Z and N flags.
PC	PC + 3
Stack	Unchanged

Related Instructions

LFDMS Divide Single (FPAC by Memory) (Long Displacement)

Exceptions

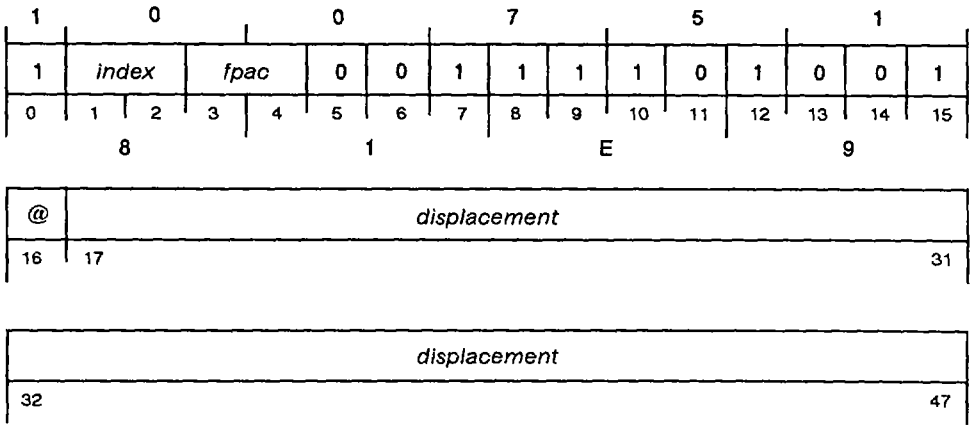
If the divisor (in memory) is zero, the processor sets the FPSR(INV) flag to 1, places error code 0 in FPSR(INP) bits and the address of the instruction in the FPSR(FPPC), and terminates the instruction.

Example

```
LFLDD 1,DIVIDND    ;Divide the double precision number at memory
LFDMD 1,DIVISOR    ;location DIVIDND by the double precision
LFSTD 1,QUOTNT     ;number at memory location DIVISOR, storing
                   ;the result at memory location QUOTNT.
```

Divide Single (FPAC by Memory) (Long Displacement) LFDMS

LFDMS *fpac*,[@]*displacement*[,*index*]



Function: *fpac* / (E) -> *fpac*

Parameters: None

LFDMS divides the 32-bit floating-point number in a floating-point accumulator by a 32-bit floating-point number in memory, and places the normalized result in the specified *fpac*.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contains normalized 32-bit result with bits 32-63 set to 0.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be specified as *fpac*; otherwise unused.
FPSR	Updated Z and N flags.
PC	PC + 3
Stack	Unchanged

Related Instructions

FDMD, FDMS, Divide a floating-point accumulator by the contents of memory.
XFDMD, XFDMS, LFDMD

Exceptions

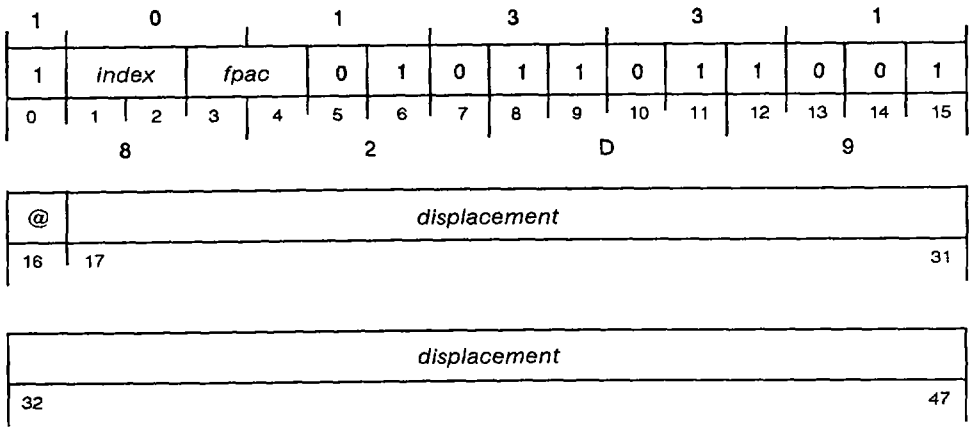
If the divisor (in memory) is zero, the processor sets the FPSR(INV) flag to 1, places error code 0 in FPSR(INP) bits and the address of the instruction in FPSR(FPPC), and terminates the instruction.

Example

```
LFLDS 0,FLOAT1     ;Divide the single precision number at memory
LFDMS 0,FLOAT2     ;location FLOAT1 by the single precision
LFSTS 0,FLOAT3     ;number at memory location FLOAT2, storing
                   ;the result at memory location FLOAT3.
```

Load Floating-Point Double (Long Displacement) LFLDD

LFLDD *fpac*,[@]*displacement*[,*index*]



Function: (E) -> *fpac*

Parameters: None

LFLDD loads a floating-point accumulator with the 64-bit contents of memory. The instruction will move unnormalized data without change.

Arguments

fpac After execution, contains 64-bit result.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be specified as *fpac*; otherwise unused.

FPSR Updated Z and N flags; undefined for unnormalized data.

PC PC + 3

Stack Unchanged

Related Instructions

FLDD, FLDS, XFLDD, XFLDS, LFLDS Load a floating-point accumulator with the contents of memory.

Exceptions

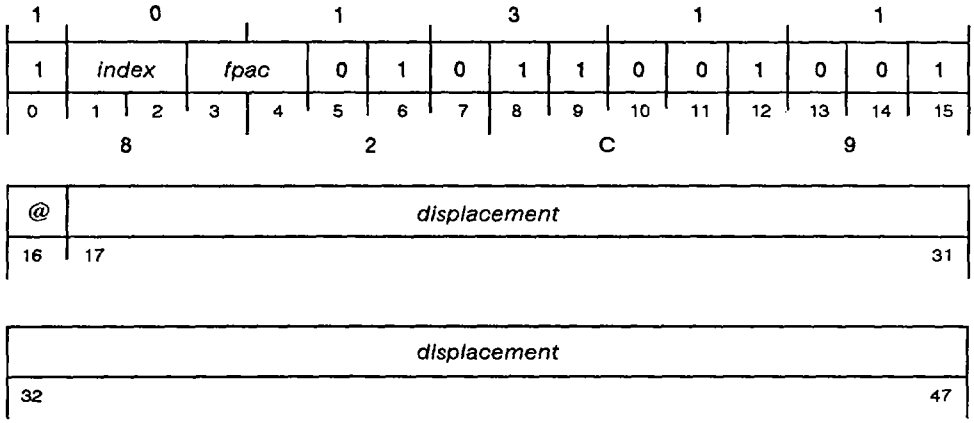
None

Example

```
LFLDD 3,FLPT3      ;Load the double precision floating-point
                   ;number at memory location FLPT3 into FPAC3.
```

Load Floating-Point Single (Long Displacement) LFLDS

LFLDS *fpac*,[@]*displacement*[,*index*]



Function: (E) -> *fpac*

Parameters: None

LFLDS loads a floating-point accumulator with the 32-bit contents of memory starting at the effective address. The instruction will move unnormalized data without change.

Arguments

fpac(0-31) After execution, contains 32-bit result with bits 32-63 set to 0.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be specified as *fpac*; otherwise unused.

FPSR Updated Z and N flags; undefined for unnormalized data.

PC PC + 3

Stack Unchanged

Related Instructions

FLDD, FLDS, XFLDD, XFLDS, LFLDD Load a floating-point accumulator with the contents of memory.

Exceptions

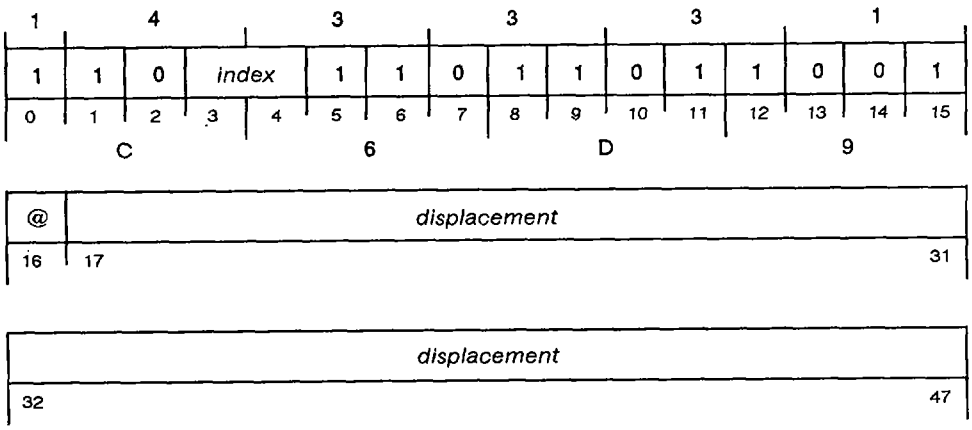
None

Example

```
LFLDS 2,FLPT2      ;Load the single precision floating-point
                   ;number at memory location FLPT2 into FPAC2.
```

Load Floating-Point Status (Long Displacement) LFLST

LFLST [@]*displacement*[,*index*]



Function: (E) -> FPSR

Parameters: E = fp#d -> unchanged

LFLST loads the contents of four sequential memory words, starting at the effective address, into the floating-point status register.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3 Unused.

FPSR After execution, contains two double words of memory as follows:

FPSR(0) not set from memory. If any of memory(1-4) is 1, ANY set to 1; otherwise, ANY is 0.

FPSR(1-11) from first memory(1-11).

FPSR(12-15) not set from memory. Contains unchangeable floating-point identification code.

FPSR(16-32) set to 0.

FPSR(33-63) set according to FPSR(ANY):

If ANY is 0, FPSR(33-63) undefined.

If ANY is 1, FPSR(28-31) contains first memory(28-31); FPSR(33-63) contains second memory(1-31).

PC PC + 3

Stack Unchanged

Related Instructions

FLST Load Floating-Point Status

FSST, LFSST Store floating-point status register to memory.

Exceptions

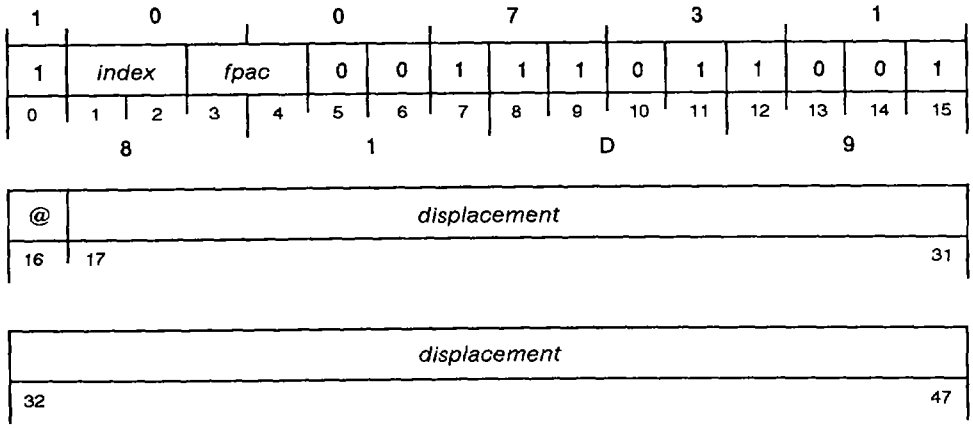
LFLST initiates a floating-point trap if FPSR(ANY) and FPSR(TE) are both 1 after FPSR(FPPC) is loaded.

Example

```
LFLST  STATUS        ;Move the four words beginning at memory
                     ;location STATUS into the wide FPSR.
```

Multiply Double (FPAC by Memory) (Long Displacement) LFMMD

LFMMD *fpac*,[@]*displacement*[,*index*]



Function: *fpac* * (E) -> *fpac*

Parameters: None

LFMMD multiplies a 64-bit floating-point number in an *fpac* by a 64-bit floating-point number in memory, and places the normalized 64-bit result in the *fpac*.

Arguments

fpac Before execution, contains 64-bit floating-point number.

After execution, contains normalized 64-bit result.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0–FPAC3 Can be individually specified as *fpac*; otherwise unused.

FPSR Updated Z and N flags.

PC PC + 3

Stack Unchanged

Related Instructions

LFMMS Multiply Single (FPAC by Memory) (Long Displacement)

Exceptions

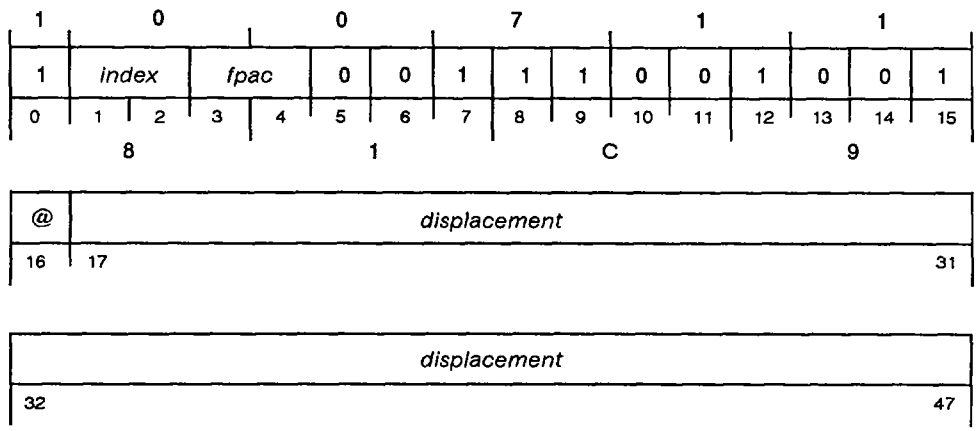
If multiplication produces an exponent overflow, the processor sets the FPSR(OVF) flag to 1 and terminates the instruction.

Example

```
LFLDD 0,DATA1      ;Multiply the two double precision floating
LFMMD 0,DATA2      ;numbers at memory locations DATA1 and DATA2,
LFSTD 0,DATA3      ;storing the result at memory location DATA3.
```

Multiply Single (FPAC by Memory) (Long Displacement) LFMMS

LFMMS *fpac*,[@]*displacement*[,*index*]



Function: *fpac* * (E) -> *fpac*

Parameters: None

LFMMS multiplies a 32-bit floating-point number in a floating-point accumulator by a 32-bit floating-point number in memory and places the normalized 32-bit result in the *fpac*.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contains normalized result with bits 32-63 set to 0.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in the 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
FPSR	Updated Z and N flags.
PC	PC + 3
Stack	Unchanged

Related Instructions

LFMMD Multiply Double (FPAC by Memory) (Long Displacement)

Exceptions

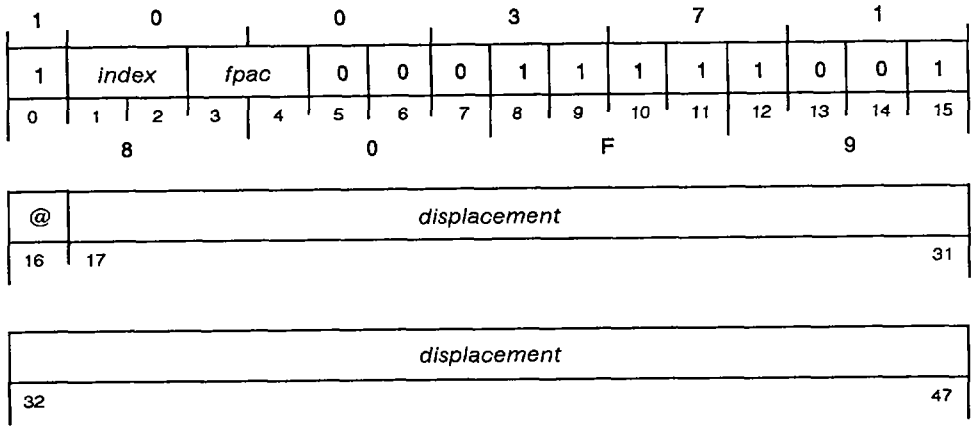
If multiplication produces an exponent overflow, the processor sets the FPSR(OVF) flag to 1 and terminates the instruction.

Example

```
LFLDS 3,MUL1        ;Multiply the two single precision floating
LFMMS 3,MUL2        ;numbers at memory locations MUL1 and MUL2,
LFSTS 3,PROD        ;storing the result at memory location PROD.
```

Subtract Double (Memory from FPAC) (Long Displacement) LFSMD

LFSMD *fpac*,[@]*displacement*[,*index*]



Function: *fpac* - (E) -> *fpac*

Parameters: None

LFSMD subtracts a 64-bit floating-point number in memory from a 64-bit floating-point number in a floating-point accumulator and places the normalized 64-bit result in the *fpac*.

Arguments

fpac Before execution, contains 64-bit floating-point number.

After execution, contains normalized 64-bit result.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be individually specified as *fpac*; otherwise unused.

FPSR Updated Z and N flags.

PC PC + 3

Stack Unchanged

Related Instructions

LFSMS Subtract Single (Memory from FPAC) (Long Displacement)

Exceptions

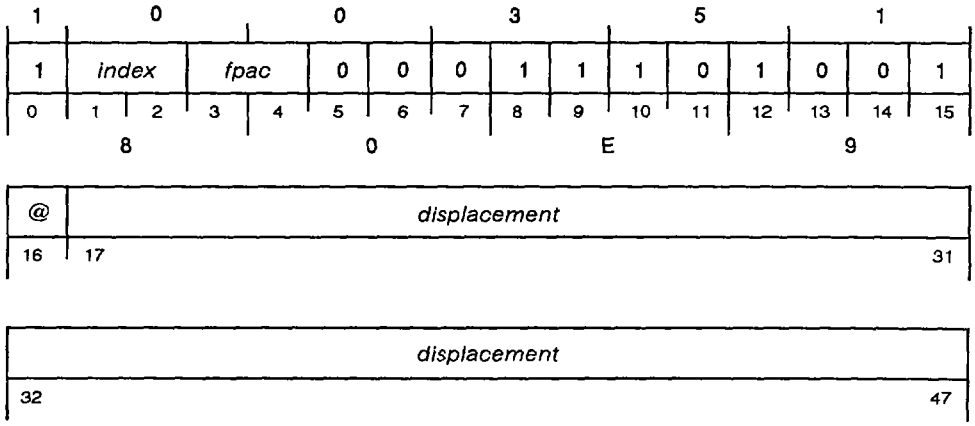
If subtraction produces an exponent overflow or underflow, the processor sets the appropriate FPSR flag to 1 and terminates the instruction.

Example

```
LFLDD 2,FLPT1      ;Subtract the double precision floating-point
LFSMD 2,FLPT2      ;number at memory location FLPT2 from the
LFSTD 2,FLPT3      ;double precision floating-point number at
                   ;memory location FLPT1, storing the result
                   ;at memory location FLPT3.
```

Subtract Single (Memory from FPAC) (Long Displacement) LFSMS

LFSMS *fpac*,[@]*displacement*[,*index*]



Function: *fpac* - (E) -> *fpac*

Parameters: None

LFSMS subtracts the 32-bit floating-point number in a memory location from the 32-bit floating-point number in a floating-point accumulator and places the normalized result in the *fpac*.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contains normalized 32-bit result with bits 32-63 set to 0.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in the 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
FPSR	Updated Z and N flags.
PC	PC + 3
Stack	Unchanged

Related Instructions

LFSMD	Subtract Double (Memory from FPAC) (Long Displacement)

Exceptions

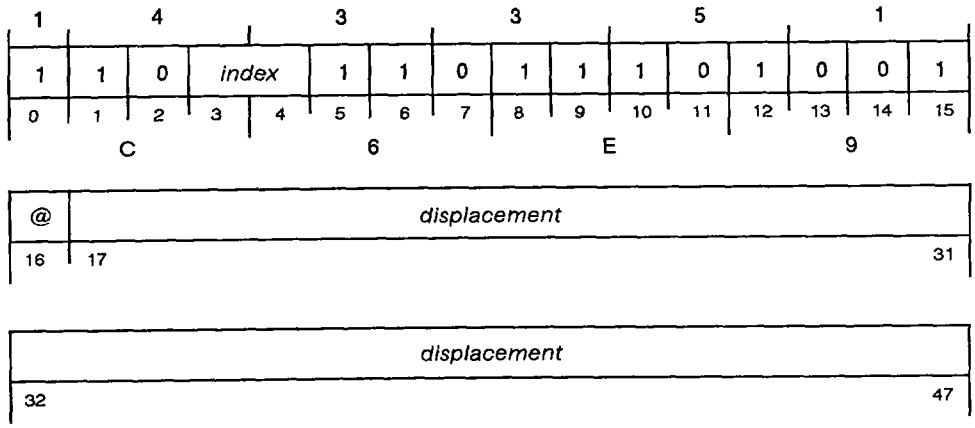
If subtraction produces an exponent overflow or underflow, the processor sets the appropriate FPSR flag to 1 and terminates the instruction.

Example

```
LFLDS 1,DATA1      ;Subtract the single precision floating-point
LFSMS 1,DATA2      ;number at memory location DATA2 from the
LFSTS 1,RESULT     ;single precision floating-point number at
                   ;memory location DATA1, storing the result
                   ;at memory location RESULT.
```

Store Floating-Point Status (Long Displacement) LFSST

LFSST [@]*displacement*[,*index*]



Function: FPSR -> (E)

Parameters: None

LFSST stores the contents of the FPSR into two sequential 32-bit memory locations starting at the effective address.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in the 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3 Unused

FPSR Stored into two memory double words as follows:

First memory double word:

FPSR(0-15) into memory(0-15).

If ANY is 0, memory(16-31) set to 0.

If ANY is 1, memory(16-27) set to 0; FPSR(28-31) into memory(28-31).

Second memory double word:

Memory(0) set to 0.

If ANY is 0, memory(1-31) undefined.

If ANY is 1, FPSR(33-63) into memory(1-31).

After execution, contents unchanged.

PC PC + 3

Stack Unchanged

Related Instructions

FSST Store Floating-Point Status

FLST, LFLST Load the floating-point status register from memory.

Exceptions

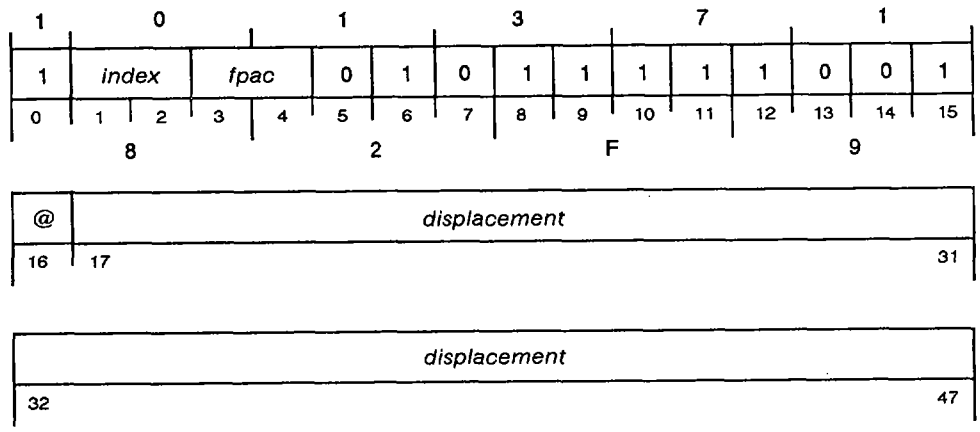
LFSST will not store a value with any combination of bit 5 (TE) and bits 1-4 concurrently set.

Example

```
LFSST STATUS        ;Store the wide FPSR in four memory words,
                    ;beginning at location STATUS.
```

Store Floating-Point Double (Long Displacement) LFSTD

LFSTD *fpac*,[@]*displacement*[,*index*]



Function: *fpac* -> (E)

Parameters: None

LFSTD stores a 64-bit floating-point number in a floating-point accumulator into two consecutive double words in memory starting at the effective address. **LFSTD** will move unnormalized data without change.

Arguments

fpac Before execution, contains 64-bit floating-point number.

After execution, contents unchanged.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be individually specified as *fpac*; otherwise unused.

FPSR Unchanged

PC PC + 3

Stack Unchanged

Related Instructions

LFSTS Store Floating-Point Single (Long Displacement)

LFLDD, LFLDS Load a floating-point accumulator with the contents of memory.

Exceptions

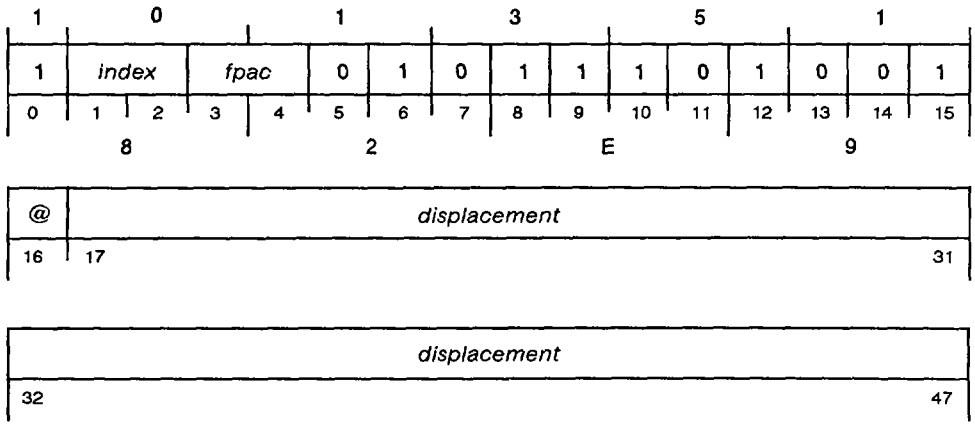
None

Example

```
FAD   2,3          ;Add FPAC2 to FPAC3, storing the double
LFSTD 3,RESULT     ;precision result at memory location RESULT.
```

Store Floating-Point Single (Long Displacement) LFSTS

LFSTS *fpac*,[@]*displacement*[,*index*]



Function: *fpac* -> (E)

Parameters: None

LFSTS stores the 32-bit floating-point number in a floating-point accumulator into a double word in memory at the effective address. **LFSTS** will move unnormalized data without change.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contents unchanged.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be individually specified as *fpac*; otherwise unused.

FPSR Unchanged

PC PC + 3

Stack Unchanged

Related Instructions

LFSTD Store Floating-Point Double (Long Displacement)

LFLDS, LFLDD Load a floating-point accumulator with the contents of memory.

Exceptions

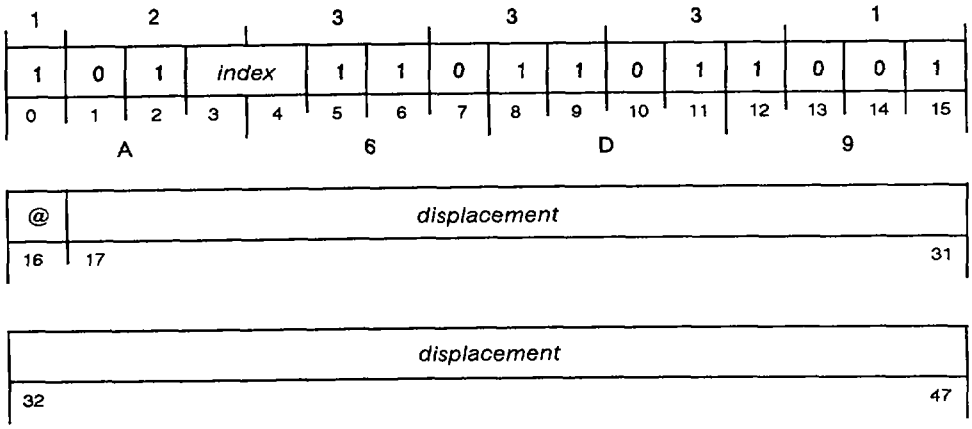
None

Example

```
FAS   0,1          ;Add FPAC0 to FPAC1, storing the single
LFSTS 1,SUM        ;precision result at memory location SUM.
```

Jump (Long Displacement) LJMP

LJMP [@]*displacement*[,*index*]



Function: E -> PC

Parameters: None

LJMP calculates an effective address and loads it into the program counter.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction confined to current segment.

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	Effective address
PSR	Unchanged
Stack	Unchanged

Related Instructions

JMP, EJMP, XJMP Place an effective address into the program counter.

Exceptions

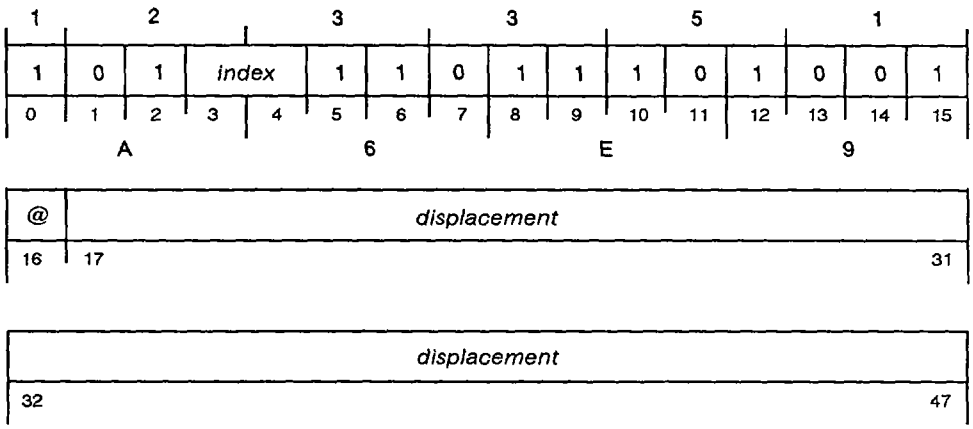
None

Example

```
    LJMP   CURR_SEG     ;Jump to a location anywhere in current segment.
    .
    .
CURR_SEG:
          . . .
```

Jump to Subroutine (Long Displacement) LJSR

LJSR [@]*displacement*[,*index*]



Function: E -> PC
PC + 3 -> AC3

Parameters: None

LJSR calculates the effective address E, loads AC3 with the current 31-bit value of the program counter plus three, and loads the program counter with E. Execution continues at the new value of the program counter.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction confined to current segment.

Registers, Flags, and Stacks

AC0-AC2	Unused
AC3	PC(before execution) + 3
CARRY	Unchanged
Overflow	0
PC	Effective address
PSR	Unchanged
Stack	Unchanged

Related Instructions

EJSR, XJSR Jump to subroutine

Exceptions

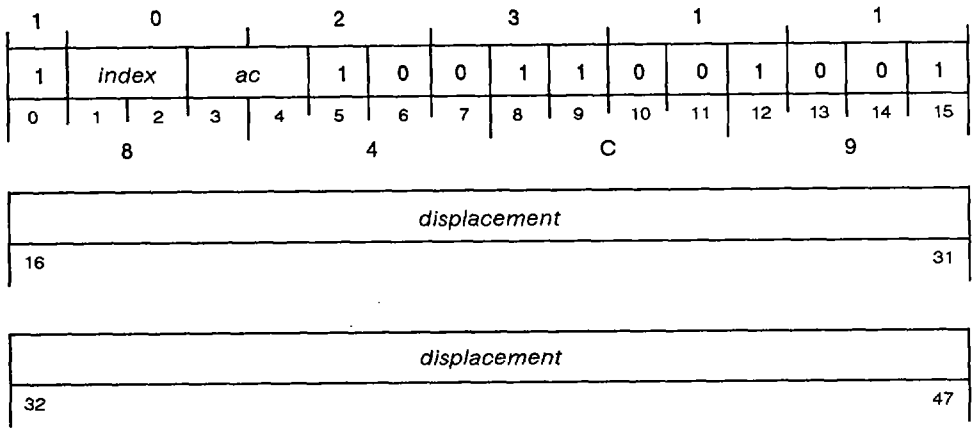
None

Example

```
LJSR   SUBROUT      ;Jump to subroutine. Return PC is put in AC3.
...
SUBROUT: WSSVR 0    ;Save ACs and return address.
         ...        ;Do the subroutine.
WRTN                ;Go back to the caller. ACs are restored.
```

Load Byte (Long Displacement) LLDB

LLDB *ac,displacement*[,*index*]



Function: (E)byte --> *ac* [bits 24–31, bits 0–23 set to 0]

Parameters: None

LLDB loads the byte at the effective byte address into an accumulator, and then zero-extends the value to 32 bits.

Arguments

ac(24–31) After execution, contains byte result with bits 0–23 set to 0.

displacement[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 3
PSR	Unchanged
Stack	Unchanged

Related Instructions

ELDB, XLDB Load a byte into an accumulator from memory.

Exceptions

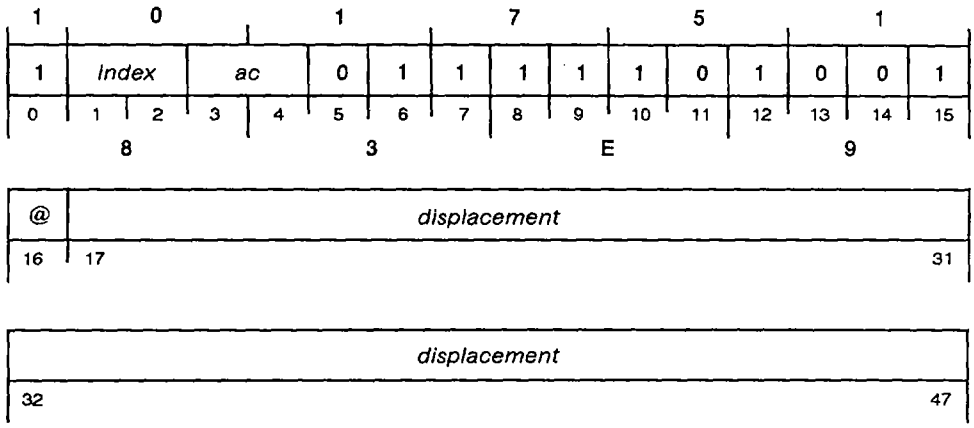
None

Example

```
    LLDB   2,(BYTE_PAIR*2)+1        ;Load AC2 with the low order byte
    ...                             ;from the word.
BYTE_PAIR: .WORD 0                  ;Location containing a pair of bytes.
```

Load Effective Address (Long Displacement) LLEF

LLEF *ac*,[@]*displacement*[,*index*]



Function: E --> *ac*

Parameters: None

LLEF calculates the effective address and loads it into the specified accumulator. Bit 0 of the result is guaranteed to be 0.

Arguments

ac After execution, contains effective address.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 3
PSR	Unchanged
Stack	Unchanged

Related Instructions

LEF, ELEF, XLEF Load an effective address into an accumulator.

Exceptions

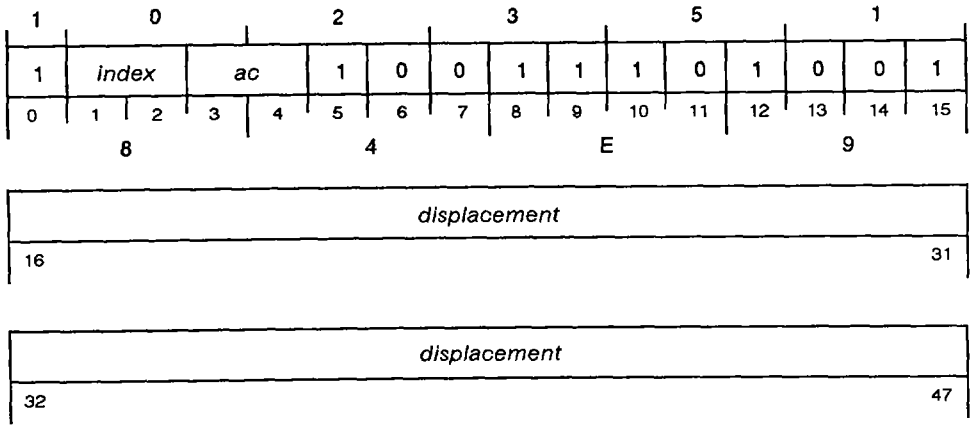
None

Example

```
    LLEF   2,WORD_ARRAY ;Get starting address of array of words.
    WADD   1,2          ;Add the word index from AC1.
    XNLDA  0,0,2        ;Get the word into AC0.
    ...
WORD_ARRAY: .BLK 16.    ;Array of 16 words.
```

Load Effective Byte Address (Long Displacement) LLEFB

LLEFB *ac,displacement[,index]*



Function: E[byte] --> *ac*

Parameters: None

LLEFB calculates the effective byte address and loads it into the specified accumulator.

Arguments

ac After execution, contains effective byte address.

displacement[,*index*]
Effective address generated by instruction can access any byte in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 3
PSR	Unchanged
Stack	Unchanged

Related Instructions

XLEFB Load Effective Byte Address (Extended Displacement)

Exceptions

None

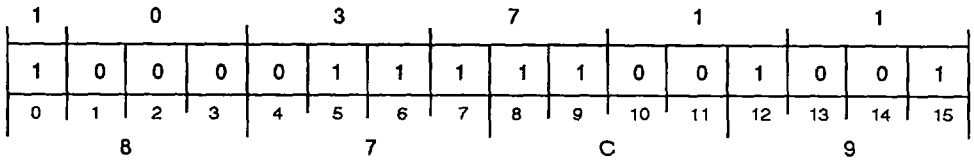
Example

```
LLEFB 2,DEST*2      ;Get the destination byte address.
LLEFB 3,SOURCE*2    ;Get the source byte address.
NLDAI 32.,0         ;Set up to move 32 bytes to destination.
WMOV  0,1           ;Also 32 bytes from the source.
WCMV                ;Move them all....
DEST: .BLK 16.      ;32 bytes.
SOURCE:   .BLK 16.  ;32 bytes.
```

Load Modified and Referenced Bits LMRF

Privileged Instruction

LMRF



Function: page(modified & referenced bit) -> AC0[right justified, 0-filled]
0 -> referenced bit
unchanged -> modified bit

Parameters: AC0 = ? -> 0(bits 0-29), Mod(bit 30), Ref(bit 31)
AC1 = pageframe # (13-31) -> unchanged

LMRF loads the modified and referenced bits of the pageframe, specified in AC1, into AC0. The referenced bit of the addressed pageframe is then set to 0.

Arguments

None

Registers, Flags, and Stacks

AC0	After execution receives modified (bit 30) and referenced bit (bit 31) with bits 0–29 set to 0.
AC1(13–31)	Before execution, contains pageframe number.
	After execution, contents unchanged.
AC2–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

SMRF	Store Modified and Referenced Bits
ORFB	OR Referenced Bits
RRFB	Reset Referenced Bits.

Exceptions

The results are indeterminate if:

Address translator not enabled

Nonexistent pageframe specified.

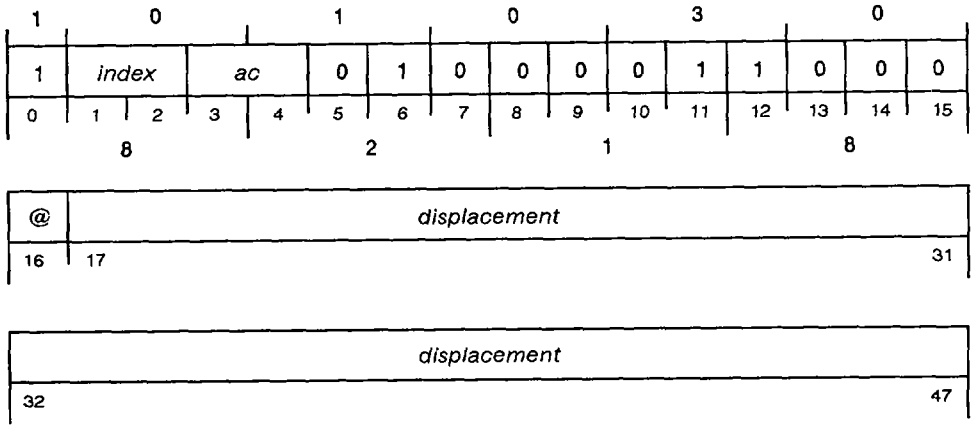
Example

```
XWLDA 1,PAGE_FRAME ;Get the page frame number of interest.
LMRF               ;Get the modified and referenced bits.
WSKBO 30.          ;Is the modified bit set?
WBR   NOT_MOD      ;No. Page frame has not been modified.
. . .              ;Yes. Page frame has been modified.
```

Narrow Add Memory Word to Accumulator LNADD

(Long Displacement)

LNADD *ac*,[@]*displacement*[,*index*]



Function: (E) + *ac* -> *ac*
ALU CRY -> CRY

Parameters: None

LNADD adds the signed 16-bit integer contained in the specified location to the signed 16-bit integer contained in the specified accumulator. Then it sign-extends the 16-bit result to 32 bits and loads it into the specified accumulator.

Arguments

ac(16-31) Before execution, contains signed 16-bit number.
After execution, contains result sign-extended to 32 bits.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Set with value of ALU CARRY
Overflow	1 if ALU overflow
PC	PC + 3
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

LWADD, XNADD, XWADD Add memory word to an accumulator.

Exceptions

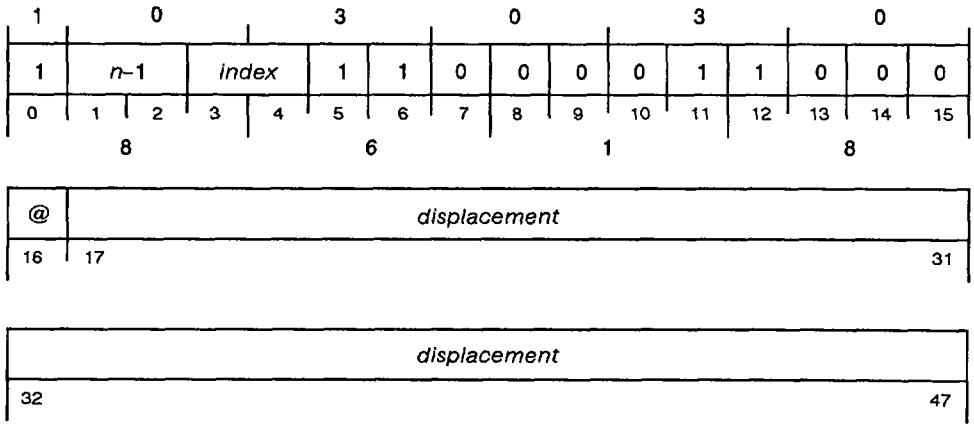
If the add produces a result greater than 32,767, PSR(OVR) is set to 1.

Example

```
LNLDA 0,FIRST      ;Get one value (only 16 bits).
LNADD 0,SECOND     ;Add the second value (16-bit arithmetic).
LNSTA 0,RESULT     ;Store the single word result.
```

Narrow Add Immediate (Long Displacement) LNADI

LNADI *n*,[@]*displacement*[,*index*]



Function: *n* + (E) -> (E)
ALU CRY -> CRY

Parameters: None

LNADI adds a value in the range of 1 to 4 to the signed 16-bit integer at the specified memory location.

Arguments

n Integer in range 1 to 4

Assembler takes coded value of *n* and subtracts one from it before placing it in immediate field. Thus, programmer should code exact value to be added.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Set with value of ALU CARRY
Overflow	1 if ALU overflow
PC	PC + 3
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

LWADI, XNADI, XWADI Add immediate value to a memory location.

Exceptions

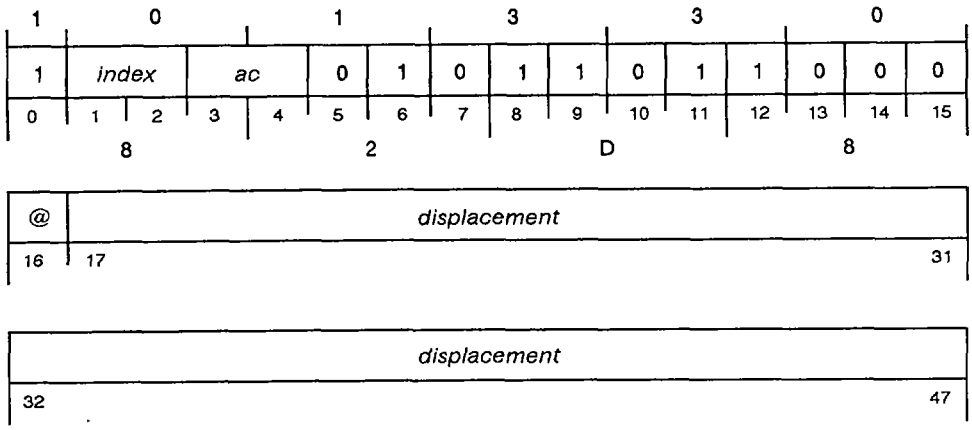
If the add produces a result greater than 32,767, PSR(OVR) is set to 1.

Example

```
    LNADI  4,COUNTER     ;Increment by 4 a counter in memory.....
COUNTER:  .WORD 0        ;16-bit counter.
```

Narrow Divide Memory Word (Long Displacement) LNDIV

LNDIV *ac*,[@]*displacement*[,*index*]



Function: *ac* / (E) -> *ac*

Parameters: None

NOTE: If (E) = 0 or result overflows; *overflow* = 1;

LNDIV sign extends the 16-bit integer contained in the specified accumulator to 32 bits and divides it by the signed 16-bit integer contained in memory at the effective address. It then sign extends the result to 32 bits and loads it into the specified accumulator.

Arguments

ac(16-31) Before execution, contains signed 16-bit integer, which processor sign-extends to 32 bits.

After execution, contains result sign-extended to 32 bits.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	1 if quotient is outside specified range or if memory word is 0; otherwise 0.
PC	PC + 3
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

LWDIV, XNDIV, XWDIV Divide an accumulator by the contents of memory.

Exceptions

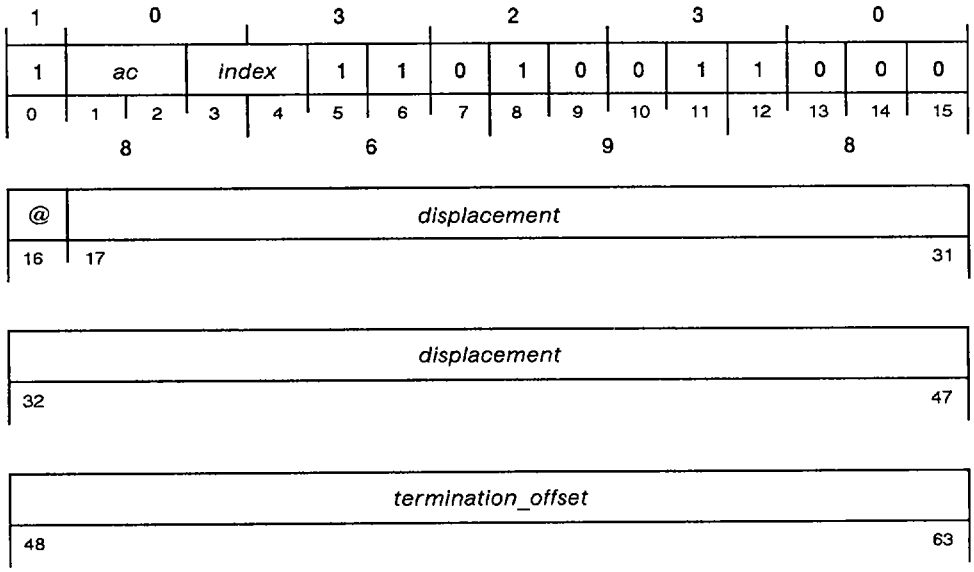
If the quotient is outside the range -32,768 to +32,767 inclusive, or if the memory location contains zero, an overflow occurs, PSR(OVR) is set to 1, and *ac* is unchanged.

Example

```
LNLDA 0,DIVIDEND   ;Get the dividend (16 bits wide).
LNDIV 0,DIVISOR    ;Divide by the divisor.
LNSTA 0,RESULT     ;Store the single word result.
```

Narrow Do Until Greater Than (Long Displacement) LNDO

```
LNDO ac,termination_offset,[@]displacement[,index]
 .              ;begin DO-loop
 . . .          ;
WBR             ;return to beginning of DO-loop
(normal return)
```



Function: (E) + 1 -> (E)
If (E) > *ac* then PC + 1 + *termination_offset* -> PC
ALU CRY -> CRY
(E) -> *ac*

Parameters: (E) = 2# -> 2# + 1

LNDO directs the processor to repeat a sequence of instructions until an incremented variable is greater than a loop count.

For each pass the through DO-loop, the processor increments the variable in memory and compares it to the loop count in *ac*. If the contents of the memory location are greater than the loop count, the loop ends, the processor moves the incremented value to *ac*, and adds the *termination_offset* plus one to the program counter. If the contents is equal to or less than the loop count, the processor moves the incremented value to *ac* and continues the DO-loop.

Instructions within the DO-loop (between **LNDO** and **WBR**) can use the loop count in *ac* for indexed addressing. With accumulator-relative indexed addressing, instructions must use absolute displacements.

Arguments

ac

Before execution, contains signed 32-bit integer for loop count. Processor increments value in memory and moves it to *ac*. Value can then be used for ac-relative addressing in DO-loop.

Although value in *ac* can be constant, DO-loop sequence can modify value in memory before restoring it to *ac*. Thus DO-loop sequence can test for condition and then prematurely terminate by modifying either variable or loop-count in memory.

Ac must be reloaded with loop count before processor returns to **LNDO**.

[*termination_offset*]
Specifies PC-relative address for normal return. Argument ranges from 0 to 64 Kwords.

[@]*displacement*[,*index*]
Specifies effective address of a double word in memory to be incremented during each pass of DO-loop. Word contains signed 16-bit integer which is sign-extended to 32-bits.

Registers, Flags, and Stacks

AC0-AC3	Can be initially specified as *ac*; otherwise unused.
CARRY	Set to value of CARRY after each DO-loop increment.
Overflow	1 if *ac* overflows.
PC	PC + 4 (Begin DO-loop) PC + 1 + *termination_offset* (Normal return)
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

LWLDA, XWLDA, WLDAI	Use these instructions to load *ac* with one more than actual loop count.
WBR	Use the Wide Branch instruction to end the DO-loop (to loop back to the **LNDO** instruction).

Exceptions

The contents of the specified memory location and the program counter value in any return block are both undefined.

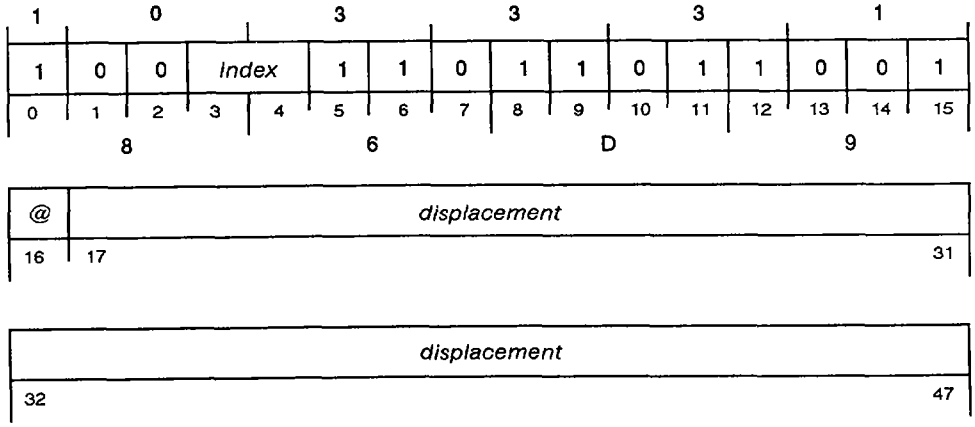
If a fixed-point overflow fault occurs while incrementing the DO-loop variable, contents of memory location and PC word in return block are undefined. (AC0 in fixed-point fault handler properly references address of DO-loop instruction.)

Example

```
    WSUB   0,0                    ;Get a 0.
    LNSTA  0,INDEX                ;Initialize the counter in memory.
LOOP:     NLDAI 5,0               ;Maximum index value.
          LNDO 0,END,INDEX        ;Start of the DO-loop.
          ...                     ;New index value is in AC0 and may be
          WBR LOOP                ;used by computations in the loop.
END:      . . .                   ;Loop was executed 5 times.
          ...
INDEX:    .WORD 0                 ;Index value.
```

Narrow Decrement and Skip if Zero (Long Displacement) LNDSZ

LNDSZ [@]*displacement*[,*index*]
(result ≠ 0 return)
(result = 0 return)



Function: (E) - 1 -> (E)
If resulting (E) = 0 then skip

Parameters: None

LNDSZ decrements by one the unsigned 16-bit integer in the memory location at the effective address. If the result is equal to zero, **LNDSZ** skips the next sequential word. The instruction is indivisible.

Arguments

[@]*displacement*[,*index*]

Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0–AC3	Unused	
CARRY	Unchanged	
Overflow	0	
PC	PC + 3	(result ≠ 0)
	PC + 4	(result = 0)
PSR	Unchanged	
Stack	Unchanged	

Related Instructions

ISZ, EISZ, LNISZ, LWISZ, XNISZ, XWISZ

Increment the contents of memory and skip if result is zero.

DSZ, EDSZ, LWDSZ, XNDSZ, XWDSZ

Decrement the contents of memory and skip if result is zero.

Exceptions

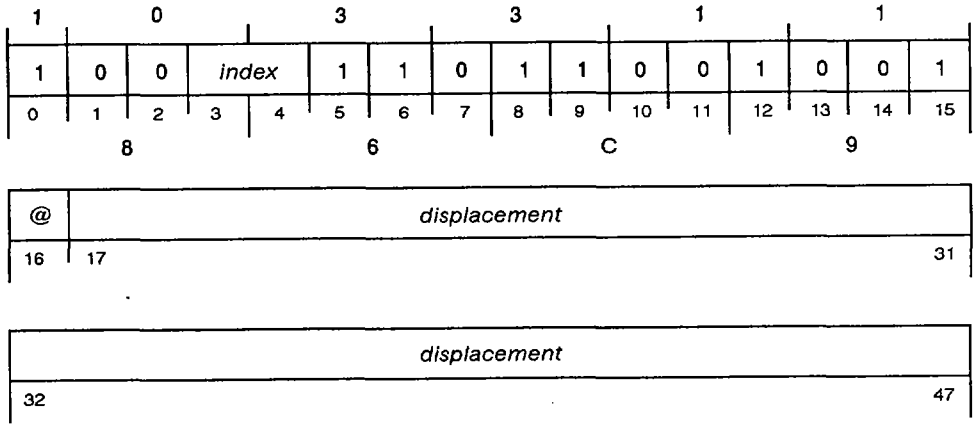
None

Example

```
NLDAI 5,0           ;Get a constant 5.
LNSTA 0,COUNTER     ;Initialize the loop counter.
LOOP: . . .         ;Beginning of loop.
.
.
.
LNDSZ COUNTER       ;Decrement counter and skip if zero.
WBR   LOOP          ;We're not done yet.
. . .               ;We did the loop 5 times.
.
.
.
COUNTER:   .WORD 0  ;Counter variable.
```

Narrow Increment and Skip if Zero (Long Displacement) LNISZ

LNISZ [@]*displacement*[,*index*]
(result ≠ 0 return)
(result = 0 return)



Function: (E) + 1 -> (E)
If resulting (E) = 0 then skip

Parameters: None

LNISZ increments by one the unsigned 16-bit integer in the memory location at the effective address. If the result is equal to zero, then the instruction skips the next sequential word. **LNISZ** is an indivisible instruction.

Arguments

[@]*displacement*[,*index*]

Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Unused	
CARRY	Unchanged	
Overflow	0	
PC	PC + 3	(result ≠ 0)
	PC + 4	(result = 0)
PSR	Unchanged	
Stack	Unchanged	

Related Instructions

ISZ, EISZ, LWISZ, XNISZ, XWISZ

Increment by one the contents of memory and skip if result is zero.

DSZ, EDSZ, LNDSZ, LWDSZ, XNDSZ, XWDSZ

Decrement by one the contents of memory and skip if result is zero.

Exceptions

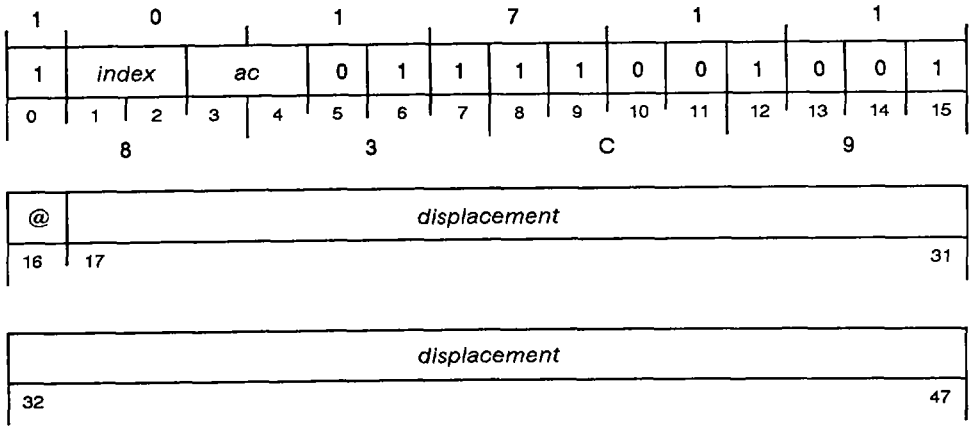
None

Example

```
NLDAI  -5,0          ;Get a constant -5.
LNSTA  0,COUNTER     ;Initialize the loop counter.
LOOP:  . . .         ;Beginning of loop.
.
.
.
LNISZ  COUNTER       ;Increment counter and skip if zero.
WBR    LOOP          ;We're not done yet.
. . .                ;We did the loop 5 times.
.
.
.
COUNTER:   .WORD 0   ;Counter variable.
```

Narrow Load Accumulator (Long Displacement) LNLDA

LNLDA *ac*,[@]*displacement*[,*index*]



Function: (E) -> *ac*

Parameters: None

LNLDA calculates the effective address and fetches the signed 16-bit integer contained in this location. Then it sign extends this integer to 32 bits and loads it into the specified accumulator.

Arguments

ac After execution, contains 32-bit result.

[@]*displacement*[,*index*]

Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 3
PSR	Unchanged
Stack	Unchanged

Related Instructions

LDA, ELDA, XNLDA, LWLDA, XWLDA

Load an accumulator with the contents of memory.

Exceptions

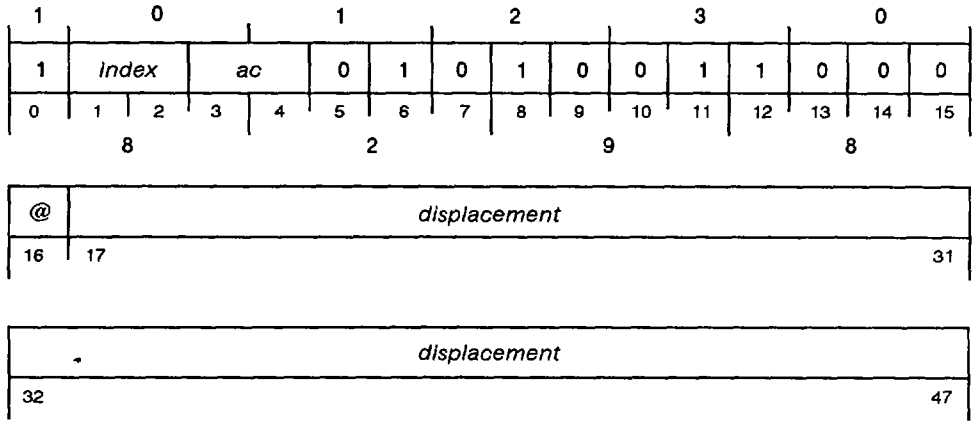
None

Example

```
LNLDA 0,SINGLE_WORD          ;Get 16 bit value and sign extend.
LWSTA 0,DOUBLE_WORD          ;Store the value as a double word.
```

Narrow Multiply Memory Word (Long Displacement) LNMUL

LNMUL *ac*,[@]*displacement*[,*index*]



Function: (E) * *ac* -> *ac*

Parameters: None

LNMUL multiplies the signed 16-bit integer contained in the specified memory location by the signed 16-bit integer contained in the specified accumulator. It then sign extends the result to 32 bits and places the result in the specified accumulator.

Arguments

ac(16-31) Before execution, contains signed 16-bit integer.

After execution, contains result sign-extended to 32 bits.

[@]*displacement*[,*index*]

Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	1 if result is outside specified range; otherwise 0.
PC	PC + 3
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

LWMUL, XNMUL, XWMUL

Multiply an accumulator by the contents of memory.

Exceptions

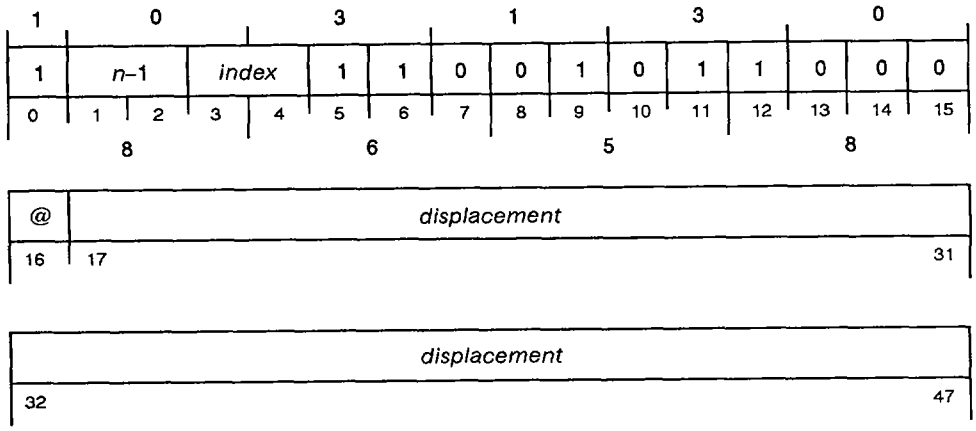
If the result is outside the range of -32,768 to +32,767 inclusive, an overflow occurs, and OVR is set to 1.

Example

```
LNLDA 0,FIRST      ;Get one value (only 16 bits).
LNMUL 0,SECOND     ;Multiply by the second value (16-bit arithmetic).
LNSTA 0,RESULT     ;Store the single word result.
```

Narrow Subtract Immediate (Long Displacement) LNSBI

LNSBI *n*,[@]*displacement*[,*index*]



Function: (E) - *n* -> (E)
ALU CRY -> CRY

Parameters: None

LNSBI subtracts an integer in the range of 1 to 4 from the signed 16-bit integer contained in the specified memory location.

Arguments

n Integer in range 1 to 4.

Assembler takes coded value of *n* and subtracts one from it before placing it in immediate field. Thus, programmer should code exact value to be subtracted.

[@]*displacement*[,*index*]

Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Unused
CARRY	Set with value of ALU CARRY
Overflow	1 if ALU overflow
PC	PC + 3
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

LWSBI, XNSBI, XWSBI

Subtract an immediate value from the contents of memory.

Exceptions

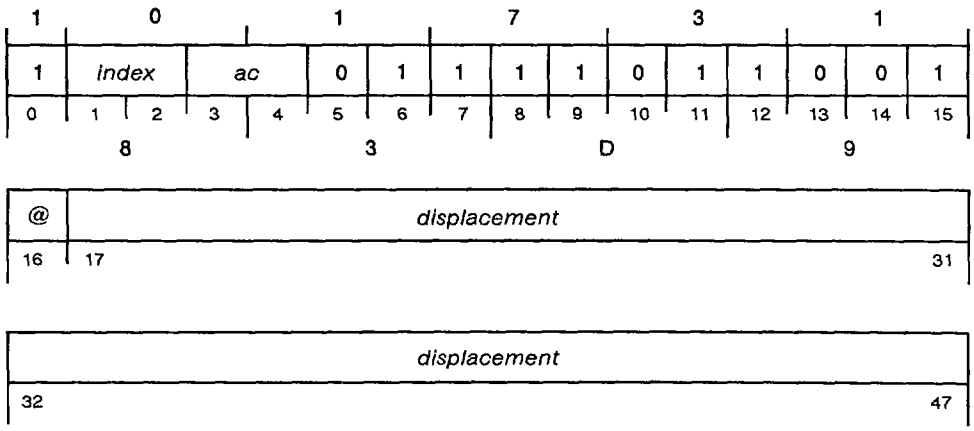
None

Example

```
LNSBI 2,COUNTER    ;Decrement by 2 a counter in memory....
COUNTER:  .WORD 0  ;16-bit counter.
```

Narrow Store Accumulator (Long Displacement) LNSTA

LNSTA *ac*,[@]*displacement*[,*index*]



Function: *ac* -> (E)

Parameters: None

LNSTA stores the low-order 16 bits of the specified accumulator into the specified memory location.

Arguments

ac(16-31) Before execution, contains 16-bit data.

After execution, contents unchanged.

[@]*displacement*[,*index*]

Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 3
PSR	Unchanged
Stack	Unchanged

Related Instructions

STA, ESTA, LWSTA, XNSTA, SWSTA

Store the contents of an accumulator into memory.

Exceptions

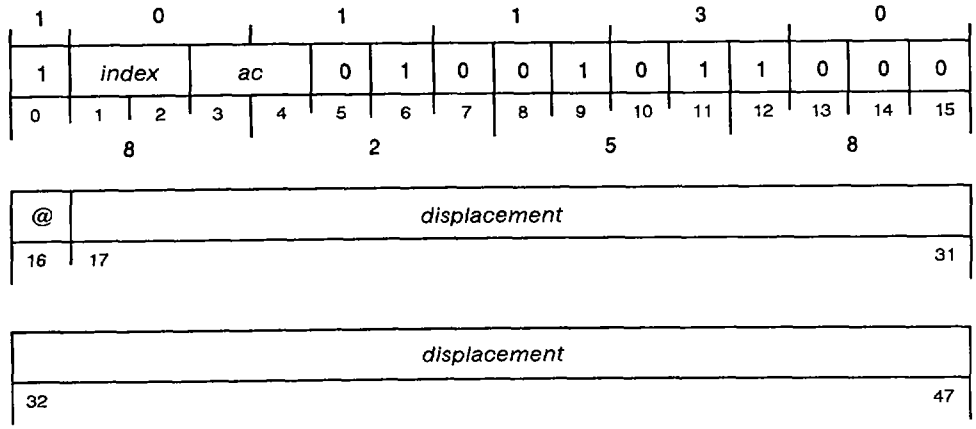
None

Example

```
LNLDA 0,FIRST      ;Get one value (only 16 bits).
LNADD 0,SECOND     ;Add the second value (16-bit arithmetic).
LNSTA 0,RESULT     ;Store the single word result.
```

Narrow Subtract Memory Word (Long Displacement) LNSUB

LNSUB *ac*,[@]*displacement*[,*index*]



Function: *ac* - (E) -> *ac*
ALU CRY -> CRY

Parameters: None

LNSUB subtracts the signed 16-bit integer contained in the specified memory location from the signed 16-bit integer contained in the specified accumulator. Then it sign extends the result to 32 bits and stores it in the specified accumulator.

Arguments

ac(16-31) Before execution, contains signed 16-bit integer.
After execution, contains result sign-extended to 32 bits.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Set with value of ALU CARRY
Overflow	1 if ALU overflow
PC	PC + 3
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

LWSUB, XNSUB, XWSUB
Subtract the contents of memory from an accumulator.

Exceptions

None

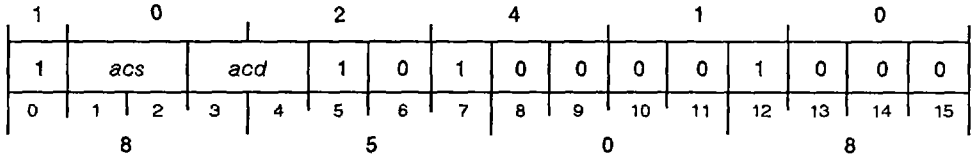
Example

```
LNLDA 0,FIRST     ;Get one value (only 16 bits).
LNSUB 0,SECOND    ;Subtract the second value (16-bit arithmetic).
LNSTA 0,RESULT    ;Store the single word result.
```

Locate Lead Bit LOB

ECLIPSE Instruction

LOB *acs,acd*



Function: *acs*(# of leading 0s) + *acd* -> *acd*

Parameters: None

LOB counts the number of high-order zeros in bits 16-31 of *acs* and adds this result to the contents of *acd*.

Arguments

acs(16-31) Before execution, contains 16-bit value.

After execution, contents unchanged unless *acs* and *acd* are same accumulator.

acd(16-31) Before execution, contains signed 16-bit integer.

After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

WLOB	Wide Locate Lead Bit
LRB, WLRB	Locate and reset the leading 1 bit to 0.

Exceptions

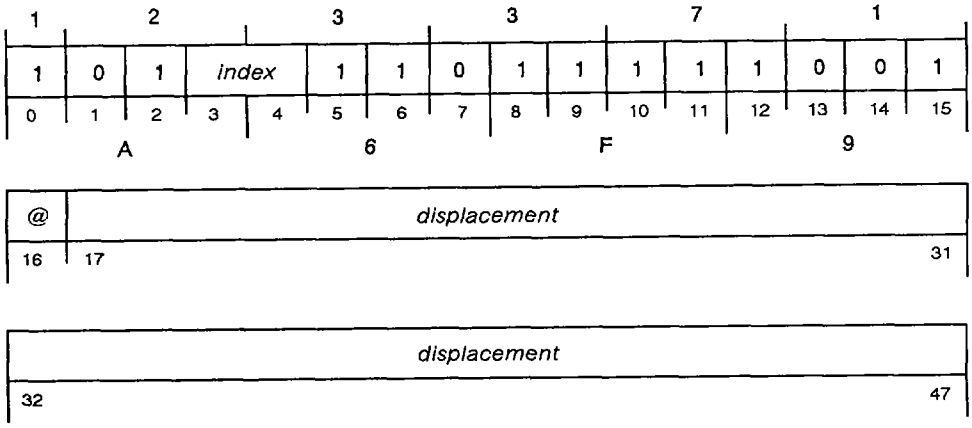
None

Example

```
NLDAI  0777,1        ;A bit pattern into AC1.
SUB    0,0           ;Set AC0 to zero.
LOB    1,0           ;Adds 7 to AC0.AC0[16-31] now is 7.
```

Push Address (Long Displacement) LPEF

LPEF [@]*displacement*[,*index*]



Function: E --> wide stack

Parameters: None

LPEF calculates the effective address and pushes it onto the wide stack, checking for stack overflow.

Arguments

[@]*displacement*[,*index*]

Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 3
PSR	Unchanged
Stack	Top double word of wide stack contains effective address with bit 0 guaranteed to be 0.

Related Instructions

XPEF	Push Address (Extended Displacement)

Exceptions

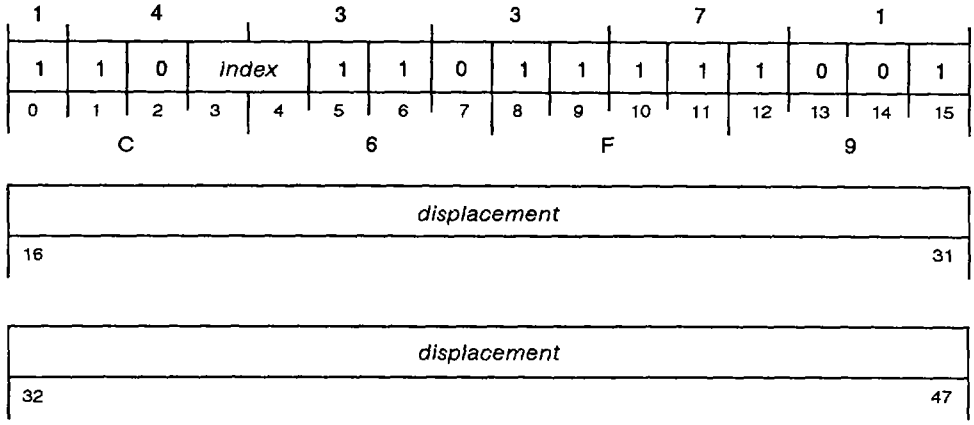
None

Example

```
LPEF   ARG_2          ;Push address of argument 2 onto the stack.
LPEF   ARG_1          ;Push address of argument 1 onto the stack.
LPEF   ARG_0          ;Push address of argument 0 onto the stack.
LCALL  SUBROUT,0,3    ;Call a subroutine with 3 arguments.
```

Push Byte Address (Long Displacement) LPEFB

LPEFB *displacement*[,*index*]



Function: E(byte) --> wide stack

Parameters: None

LPEFB calculates a 32-bit byte address and pushes it onto the wide stack, checking for stack overflow.

Arguments

displacement[,*index*]

Effective address generated by instruction can access any byte in 4-Gbyte range.

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 3
PSR	Unchanged
Stack	Top double word of wide stack contains byte address.

Related Instructions

XPEFB	Push Byte Address (Extended Displacement)

Exceptions

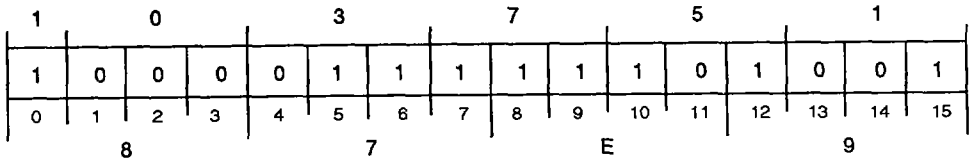
None

Example

```
LPEFB  ARG_1*2       ;Push byte address of argument 1 onto the stack.
LPEF   ARG_0         ;Push address of argument 0 onto the stack.
LCALL  SUBROUT,0,2   ;Call a subroutine with 2 arguments.
                     ;Subroutine must be expecting a byte
                     ;address to ARG_1 and a word address to ARG_2.
```

Load Physical Address And Skip LPHY

LPHY



Function: logical -> physical
PTE -> AC0
physical address -> AC2

Parameters: AC1 = logical address -> unchanged

NOTE: If PTE ≠ page or validity fault, next word skipped

LPHY translates the logical word address contained in AC1 to a physical address, stores it in AC2, and skips the next instruction.

Arguments

None

Registers, Flags, and Stacks

AC0	After execution, contains last resident page table entry (PTE).
AC1	Before execution, contains logical word address to be translated to physical address.
	After execution, contents unchanged.
AC2	After execution, contains 32–bit physical word address, translated from contents of AC1.
AC3	Unchanged
CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

XWLDA, LWLDA

Use these instructions to place data in AC1.

Exceptions

The instruction terminates and the next sequential instruction is executed if any of the following faults occur:

Address translator disabled

Ring field of logical address is less than the current ring field (protection fault 4 code stored in AC1)

Invalid or depth fault on SBR check

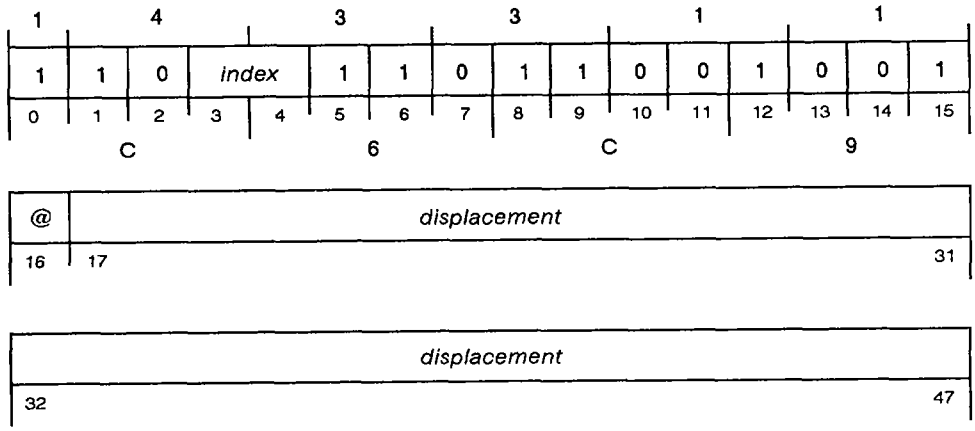
Page or invalid fault on PTE check

Example

```
XWLDA  1,LOGICAL_ADDR          ;Get the logical address.
LPHY                           ;Convert to physical address.
WBR    FAULT                   ;An error occurred.
XWSTA  0,PTE                   ;Store the PTE that was returned.
XWSTA  2,PHYSICAL_ADDR         ;Store the physical address.
```

Push Jump (Long Displacement) LPSHJ

LPSHJ [@]*displacement*[,*index*]



Function: PC +3 -> wide stack
E -> PC

Parameters: None

LPSHJ pushes a return address on the wide stack and jumps to a specified location. Sequential operation continues with the word addressed by the updated value of the program counter.

Arguments

[@]*displacement*[,*index*]

Effective address generated by this instruction is confined to current segment of execution.

Reisters, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	Effective address
PSR	Unchanged
Stack	Top double word of wide stack contains PC(before execution) + 3 (always points to location in current segment)

Related Instructions

PSHJ, XPSHJ

Push a return address onto stack and jump.

Exceptions

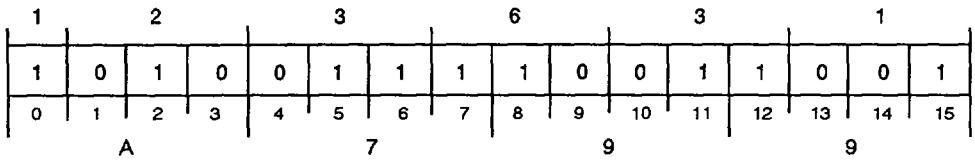
None

Example

```
LPSHJ  SUBROUT      ;Call a subroutine. Return PC is on the stack.
....
SUBROUT:....        ;Subroutine is implemented here.
WPOPJ               ;Pop return address and return to caller. ACs
                    ;modified in the subroutine are not restored.
```

Load Processor Status Register LPSR

LPSR



Function: PSR -> AC0

Parameters: None

LPSR loads the 16-bit Processor Status Register into bits 0–15 of AC0 and fills the bits 16–31 of AC0 with zeros. Reserved bits of the PSR are returned in AC0 as zero.

Arguments

None

Registers, Flags, and Stacks

AC0(0–15)	After execution, contains PSR bits. Reserved bits and bits 16–31 are filled with zeros.
AC1–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

SPSR	Store Processor Status Register

Exceptions

None

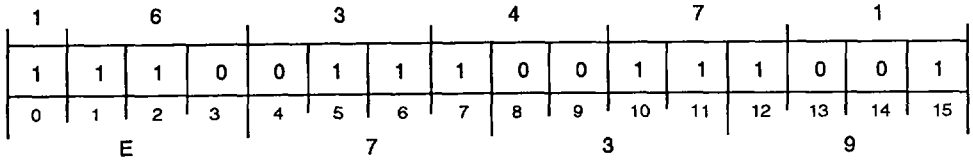
Example

```
LPSR              ;Put the PSR into AC0[0-15].
XWSTA 0,PSR       ;Store the PSR in memory.
```

Load Page Table Entry LPTE

Privileged Instruction

LPTE
(exception return)
(normal return)



Function: Returns PTE of corresponding logical address.
If no exceptions then,
PTE -> AC0
physical address of last PTE -> AC2
PC = PC + 2
Else
all ac -> unchanged
PC = PC + 1

Parameters: AC0 = x -> PTE
AC1 = logical address -> unch
AC2 = x -> physical address of PTE returned in AC0
AC3 = SBR table address -> unch

NOTE: The addresses in AC1 and AC3 may not specify indirection.

Exceptions, in order of detection are:
ATU is off, not in ring 0, invalid SBR, Page Table depth error, invalid second level PTE, non-resident first level PTE.
Conditions not considered errors are: invalid first level PTE, non-resident object page.

LPTE obtains the Page Table Entry (PTE) that corresponds to the translation of a given logical address contained in AC1. If no exceptions are encountered, the next sequential word is skipped and execution continues with the PTE returned to AC0 and its physical address to AC2.

Arguments

None

Registers, Flags, and Stacks

AC0	After execution, contains Page Table Entry.
AC1	Before execution, contains logical address whose PTE is desired. Address may not specify indirection. Bits 1 through 3 of accumulator (ring bits) used to index by double words into Segment Base Register (SBR) table pointed to by AC3 to obtain an SBR for use in translation.
	After execution, contents unchanged.
AC2	After execution, contains physical address of PTE.
AC3	Before execution, contains nonindirectable pointer to table of eight double words. Each double word contains an SBR.
	After execution, contents unchanged.
CARRY	Unchanged
Overflow	Unaffected

PC	PC + 1 (exception return) PC + 2 (normal return)
PSR	Unchanged
Stack	Unchanged

Related Instructions

SPTE	Store Page Table Entry

Load effective address

Use these instructions to put PTE logical address in AC1 or SBR table address in AC3.

Exceptions

If an exception does occur, execution continues with the next sequential instruction, and no accumulators are modified.

Exceptions that take the error path (in order of detection) are: ATU is off, not in ring 0,invalid SBR, Page Table depth error, invalid second-level PTE, non-resident first-level PTE.

The following conditions are not considered errors: invalid first-level PTE and non-resident object page.

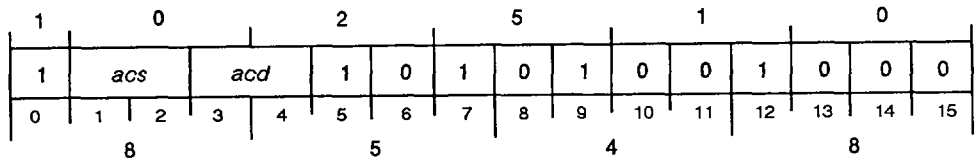
Example

```
XLEF   3,SBR_TABLE                ;Get the SBR table address.
XWLDA  1,LOGICAL_ADDRESS          ;Address for which we want PTE.
LPTE
WBR    FAULT                      ;A fault occurred.
XWSTA  0,PTE                      ;Store the PTE in memory.
XWSTA  2,PHYSICAL_ADDRESS         ;Store the physical address in memory.
       .
       .
       .
SBR_TABLE:
       .DWORD        ;Ring 0 SBR.
       .DWORD        ;Ring 1 SBR.
       .DWORD        ;Ring 2 SBR.
       .DWORD        ;Ring 3 SBR.
       .DWORD        ;Ring 4 SBR.
       .DWORD        ;Ring 5 SBR.
       .DWORD        ;Ring 6 SBR.
       .DWORD        ;Ring 7 SBR.
```

Locate and Reset Lead Bit — LRB

ECLIPSE Instruction

LRB *acs,acd*



Function: *acs*(# of leading 0s) + *acd* -> *acd*
0 -> high 1(*acs*)

Parameters: None

NOTE: If *acs* is *acd*; then nothing is added, but 0 -> leading 1

LRB counts the number of high-order zeros in *acs*, then adds this number to the signed 16-bit integer contained in *acd* and sets the leading 1 in *acs* to 0.

Arguments

acs(16-31) Before execution, contains a 16-bit value.

After execution, the leading 1 in bits 16-31 set to 0.

If *acs* and *acd* are specified to be the same accumulator, then **LRB** sets the leading 1 in the accumulator to 0 and no count is taken.

acd(16-31) Before execution, contains signed 16-bit integer.

After execution, contains sum of *acd* and number of high-order zeros found in *acs*.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

LOB Locate Lead Bit

Exceptions

None

Example

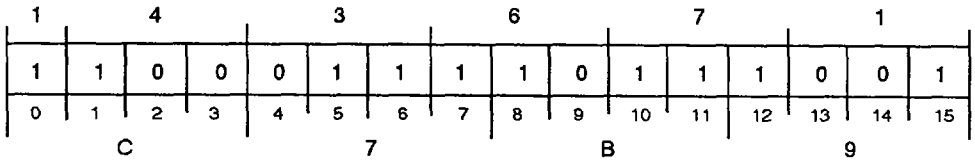
```
LRB   2,0          ;Counts the number of high-order zeros in AC2 and
                   ;adds this number to the contents of AC0, then
                   ;resets the high-order zero in AC2 to one.

....
```

Load All Segment Base Registers LSBRA

Privileged Instruction

LSBRA



Function: new values -> SBR(0-7)

Parameters: AC0 = E -> unchanged

LSBRA loads the eight segment base registers (SBRs) with new values. The instruction uses the contents of AC0 as the pointer to a table containing the new values. After loading the SBRs, the instruction purges the address translator. If the address translator was disabled at the beginning of the instruction cycle, the instruction then enables it.

Arguments

None

Registers, Flags, and Stacks

AC0 — Before execution, contains starting address of eight-double-word block containing new SBR values. Values arranged in table and moved to SBRs as follows:

Double Word In Block	Destination	Order Moved
1	SBR0	First
2	SBR1	Second
3	SBR2	Third
4	SBR3	Fourth
5	SBR4	Fifth
6	SBR5	Sixth
7	SBR6	Seventh
8	SBR7	Eighth

After execution, contents unchanged.

AC1–AC3 — Unused

CARRY — Unchanged

Overflow — 0

PC — PC + 1

PSR — Unchanged

Stack — Unchanged

Related Instructions

Load effective address
: Use these instructions to load AC0 with the address of the table containing the new SBR values.

LSBRS Load Segment Base Registers 1–7

Exceptions

If a fault occurs, SBRs 1–7 are not loaded from the block of double words. If an invalid address is loaded into SBR0, the processor disables the address translator and a protection fault occurs (code 3 is returned to AC1). This means that logical addresses are identical to physical addresses, and the fault is processed in physical address space.

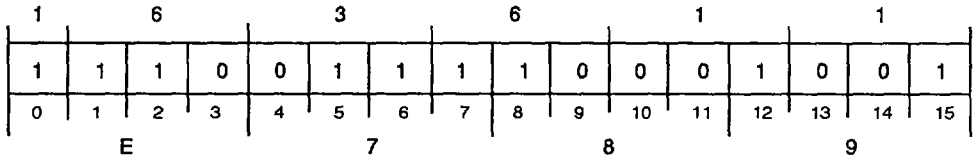
Example

```
XLEF   0,SBR_TABLE  ;Get the SBR table address.
LSBRA               ;Load all SBRs.
.                   ;Processing continues in new context.
.
SBR_TABLE:
       .DWORD       ;Ring 0 SBR.
       .DWORD       ;Ring 1 SBR.
       .DWORD       ;Ring 2 SBR.
       .DWORD       ;Ring 3 SBR.
       .DWORD       ;Ring 4 SBR.
       .DWORD       ;Ring 5 SBR.
       .DWORD       ;Ring 6 SBR.
       .DWORD       ;Ring 7 SBR.
```

Load Segment Base Registers LSBRS

Privileged Instruction

LSBRS



Function: new values -> SBR(1-7)

Parameters: AC0 = E -> unchanged

LSBRS loads segment base registers (SBRs) 1 through 7 with new values. The instruction uses the contents of AC0 as the pointer to a table containing the new values. After loading the SBRs, the instruction purges the address translator. If the address translator was disabled at the beginning of the instruction cycle, the instruction then enables it.

Arguments

None

Registers, Flags, and Stacks

AC0 — Before execution, contains starting address of seven-double-word block containing new SBR values. Values arranged in table and moved to SBRs as follows:

Double Word in Block	Destination	Order Moved
1	SBR1	First
2	SBR2	Second
3	SBR3	Third
4	SBR4	Fourth
5	SBR5	Fifth
6	SBR6	Sixth
7	SBR7	Seventh

After execution, contents unchanged.

AC1–AC3 — Unused

CARRY — Unchanged

Overflow — 0

PC — PC + 1

PSR — Unchanged

Stack — Unchanged

Related Instructions

Load effective address
: Use these instructions to load AC0 with the address of the table containing the new SBR values.

LSBRA Load All Segment Base Registers

Exceptions

None

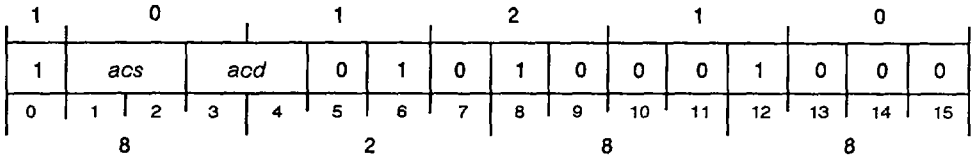
Example

```
XLEF   0,SBR_TABLE+2            ;Get the SBR table address.
LSBRS                           ;Load only SBRs 1-7.
.                               ;Processing continues in new context.
.
SBR_TABLE:
        .DWORD          ;Ring 0 SBR.
        .DWORD          ;Ring 1 SBR.
        .DWORD          ;Ring 2 SBR.
        .DWORD          ;Ring 3 SBR.
        .DWORD          ;Ring 4 SBR.
        .DWORD          ;Ring 5 SBR.
        .DWORD          ;Ring 6 SBR.
        .DWORD          ;Ring 7 SBR.
```

Logical Shift LSH

ECLIPSE Instruction

LSH *acs,acd*



Function: shift *acd*(*acs*(bits 24-31[+=left,-=right])) -> *acd*

Parameters: None

LSH shifts the contents of *acd* either left or right depending on the number contained in *acs*. Bits shifted out are lost, and the vacated bit positions are filled with zeros.

Arguments

acs(24-31) Before execution, contains signed 8-bit integer that determines direction of shift and number of bits to be shifted. Magnitude of number determines number of bits to be shifted.

If bit 24 is 0, shifting is to left; if bit 24 is 1, shifting is to right. If number is zero, no shifting is performed.

If magnitude is greater than 15, all bits of *acd* are set to 0.

After execution, contents unchanged unless *acs* and *acd* are same accumulator.

acd(16-31) Before execution, contains 16-bit value.

After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

WLSH Wide Logical Shift

Exceptions

None

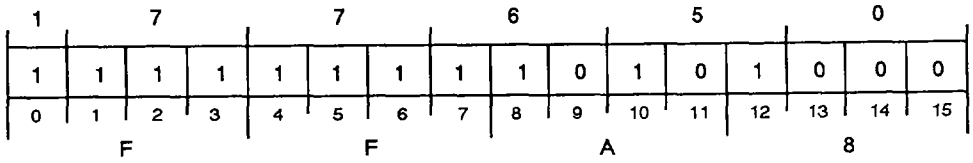
Example

```
ELDA   0,NFIVE      ;Get the shift count of -5.
ADC    1,1          ;Set AC1 to all ones.
LSH    0,1          ;Shift. AC1[16-31] now contains 003777(8).
...
NFIVE: .WORD -5     ;Constant -5.
```

Load Sign LSN

ECLIPSE Instruction

LSN



Function: (E)[decimal #] = (non-0 or 0, + or -)
AC3 -> AC2

Parameters: AC1 = x -> result:
+1 (+ non-0)
-1 (- non-0)
0 (+ 0)
-2 (- 0)
AC3 = byte pointer->?

LSN evaluates a decimal number in the specified memory location and returns a code that classifies the number as zero or nonzero and identifies its sign.

Arguments

None

Registers, Flags, and Stacks

AC0 — Unused

AC1(16-31) — Before execution, specifies type and length of data to be evaluated.

After execution, contains value code as follows:

Value of Number	Code
Positive nonzero	+1
Negative nonzero	-1
Positive zero	0
Negative zero	-2

AC2(16-31) — After execution, contains original contents of AC3.

AC3(16-31) — Before execution, contains logical address of byte to be evaluated. Effective address generated by instruction confined to current segment.

After execution, contents undefined.

CARRY — Unchanged

Overflow — 0

PC — PC + 1

PSR — Unchanged

Stack — Unchanged

Related Instructions

WLSN — Wide Load Sign

Exceptions

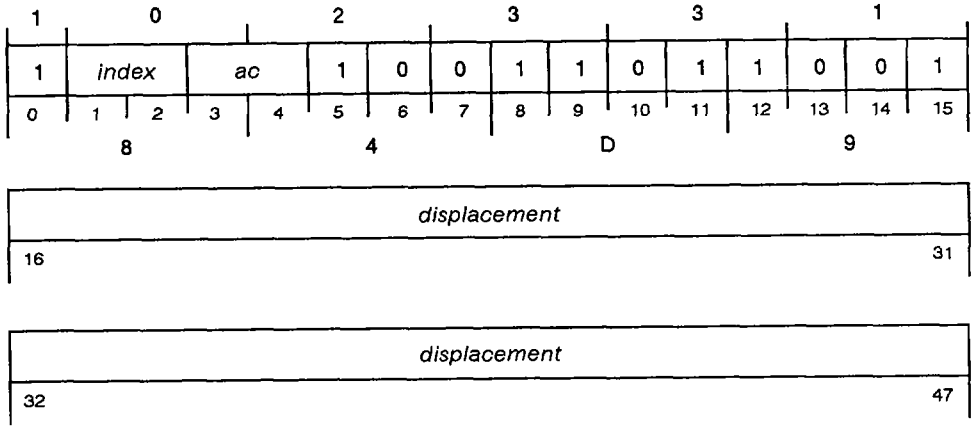
None

Example

```
XNLDA  1,DTYPE      ;AC1 contains the data type indicator.
XLEF   3,DATA       ;Word pointer to integer field.
WADD   3,3          ;AC3 is a byte pointer to the integer.
LSN                 ;Get a code into AC1 which reflects the
                    ;sign of the commercial integer.
```

Store Byte (Long Displacement) LSTB

LSTB *ac,displacement*[,*index*]



Function: *ac*[right byte] --> (E)byte

Parameters: None

LSTB calculates the effective byte address. The instruction then moves a copy of the contents of bits 24–31 of the specified accumulator into memory at the location specified by the byte address.

Arguments

ac(24–31) Before execution, contains 8–bit data.

After execution, contents unchanged.

displacement[,*index*]

Effective address generated by instruction can access any word in 4–Gbyte range.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 3
PSR	Unchanged
Stack	Unchanged

Related Instructions

ESTB, XSTB Store a byte from an accumulator into memory.

Exceptions

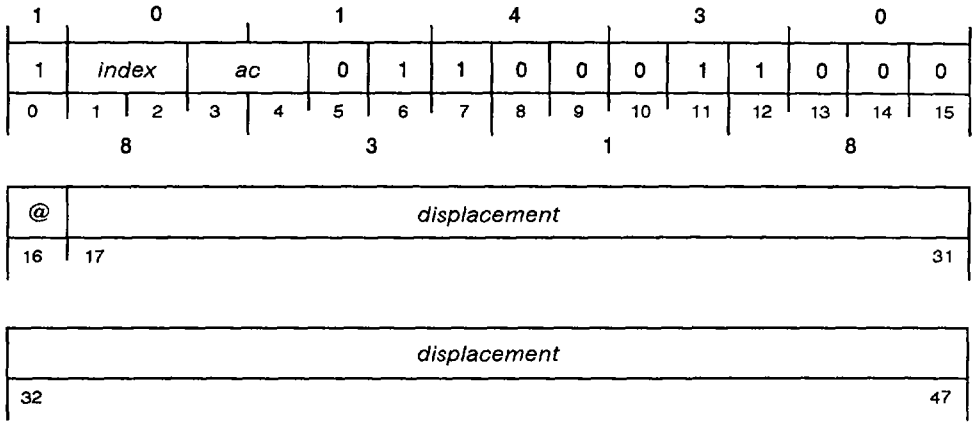
None

Example

```
LSTB   2,(BYTE_PAIR*2)+1        ;Store the byte in bits 24-31 of AC2.
                                ;into the low-order byte of the word
                                ;in memory.
.
.
BYTE_PAIR:
       .WORD 0                  ;Location containing a pair of bytes.
```

Wide Add Memory Word to Ac (Long Displacement) LWADD

LWADD *ac*,[@]*displacement*[,*index*]



Function: (E) + *ac* -> *ac*
ALU CRY -> CRY

Parameters: None

LWADD calculates the effective address and adds the signed 32-bit integer contained in this location to the signed 32-bit integer contained in the specified accumulator.

Arguments

ac Before execution, contains signed 32-bit number.
After execution, contains result.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Set with value of ALU CARRY
Overflow	1 if ALU overflow
PC	PC + 3
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

LNADD, XNADD, XWADD
Add memory contents to an accumulator.

Exceptions

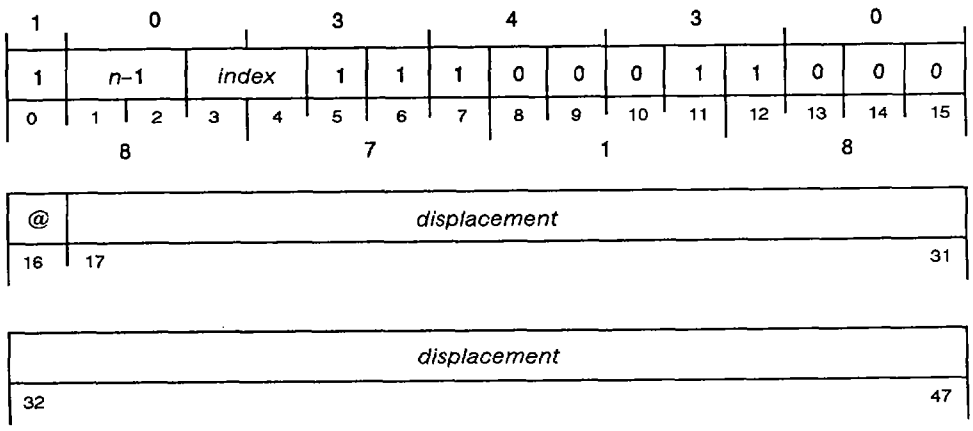
If the result of the add produces a result greater than 2,147,483,647, PSR(OVR) is set to 1.

Example

```
LWLDA 0,FIRST     ;Get one value (32 bits).
LWADD 0,SECOND    ;Add the second value (32-bit arithmetic).
LWSTA 0,RESULT    ;Store the double word result.
```

Wide Add Immediate (Long Displacement) LWADI

LWADI *n*,[@]*displacement*[,*index*]



Function: *n* + (E) -> (E)
ALU CRY -> CRY

Parameters: None

LWADI adds an integer in the range of 1 to 4 to the signed 32-bit integer contained in a memory location.

Arguments

n Integer in range 1 to 4

Assembler takes coded value of *n* and subtracts one from it before placing it in immediate field. Thus, programmer should code exact value to be added.

[@]*displacement*[,*index*]

Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Set with value of ALU CARRY
Overflow	1 if ALU overflow
PC	PC + 3
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

LNADI, XNADI, XWADI

Add immediate value to memory.

Exceptions

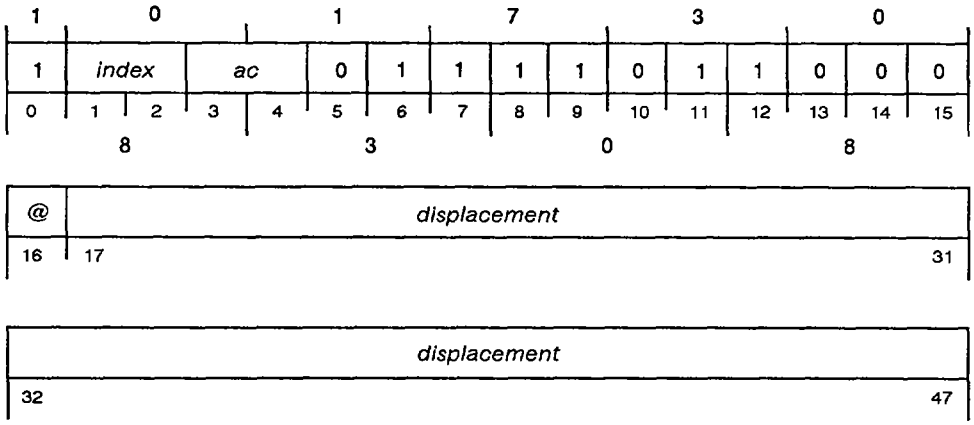
None

Example

```
LWADI 4,COUNTER    ;Increment by 4 a counter in memory....
COUNTER:  .DWORD 0 ;32-bit counter.
```

Wide Divide Memory Word (Long Displacement) LWDIV

LWDIV *ac*,[@]*displacement*[,*index*]



Function: *ac* / (E) -> *ac*

Parameters: None

NOTE: If (E)=0 or result overflows; *overflow* = 1; *ac* = unchanged

LWDIV sign extends the signed 32-bit integer contained in the specified accumulator to 64 bits and divides it by the signed 32-bit integer contained in the memory, placing the quotient into the specified accumulator.

Arguments

ac Before execution, contains signed 32-bit integer.

After execution, contains result.

[@]*displacement*[,*index*]

Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	1 if quotient outside specified range or if memory word is 0; otherwise 0.
PC	PC + 3
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

LNDIV, XNDIV, XWDIV

Divide an accumulator by the contents of memory.

Exceptions

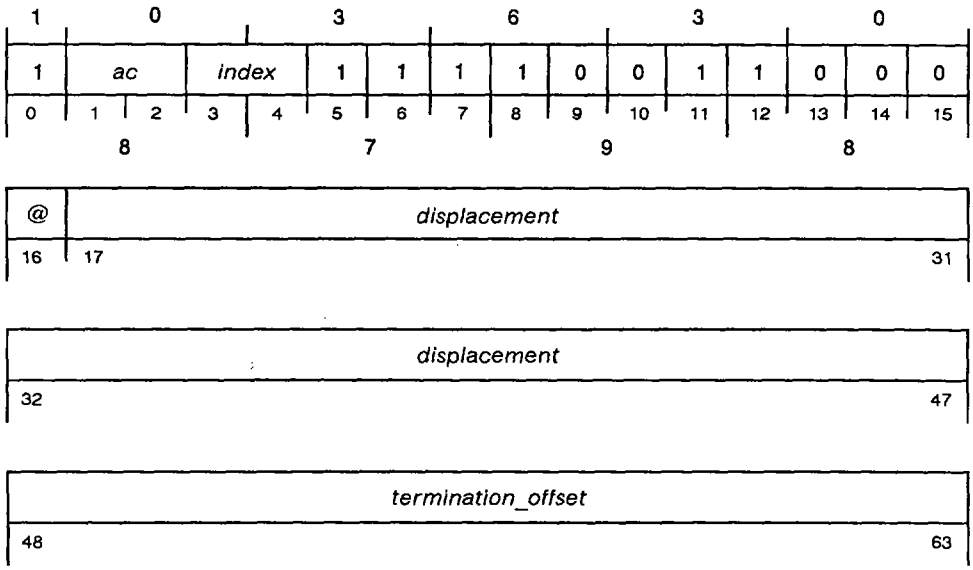
If the quotient is outside the range of -2,147,483,648 to +2,147,483,647 inclusive, or if the memory word is zero, an overflow occurs, PSR(OVR) is set to 1, and *ac* is unchanged.

Example

```
LWLDA 0,DIVIDEND   ;Get the dividend (32 bits wide).
LWDIV 0,DIVISOR    ;Divide by the divisor.
LWSTA 0,RESULT     ;Store the double word result.
```

Wide Do Until Greater Than (Long Displacement) LWDO

```
LWDO ac,termination_offset,[@]displacement[,index]
 .              ;begin DO-loop
 . . .          ;
WBR             ;return to beginning of DO-loop
(normal return)
```



Function: (E) + 1 -> (E)
If (E) > *ac*, PC + 1 + *termination_offset* -> PC
ALU CRY -> CRY
(E) -> *ac*

Parameters: (E) = 2# -> 2# + 1

LWDO directs the processor to repeat a sequence of instructions until an incremented variable is greater than a loop count.

For each pass the through DO-loop, the processor increments the variable in memory and compares it to the loop count in *ac*. If the contents of the memory location are greater than the loop count, the loop ends, the processor moves the incremented value to *ac*, and adds the *termination_offset* plus one to the program counter. If the contents is equal to or less than the loop count, the processor moves the incremented value to *ac* and continues the DO-loop.

Instructions within the DO-loop (between **LWDO** and **WBR**) can use the loop count in *ac* for indexed addressing. With accumulator-relative indexed addressing, instructions must use absolute displacements.

Arguments

ac Before execution, contains signed 32-bit integer for loop count. Processor increments value in memory and moves it to *ac*. Value can then be used for ac-relative addressing in DO-loop.

Although value in *ac* can be constant, DO-loop sequence can modify value in memory before restoring it to *ac*. Thus DO-loop sequence can test for condition and then prematurely terminate by modifying either variable or loop-count in memory.

Ac must be reloaded with loop count before processor returns to **LWDO**.

[*termination_offset*]
Specifies PC-relative address for normal return. Argument ranges from 0 to 64 Kwords.

[@]*displacement*[,*index*]
Specifies effective address of a double word in memory to be incremented during each pass of DO-loop. Double-word contains signed 32-bit integer.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Set to value of CARRY after each DO-loop increment.
Overflow	1 if *ac* overflows.
PC	PC + 4 (Begin DO-loop) PC + 1 + termination_offset (Normal return)
PSR	Sets OVR to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

LWLDA, XWLDA, WLDAI
Use these instructions to load *ac* with one more than actual loop count.

WBR Use the Wide Branch instruction to end the DO-loop (to loop back to the **LWDO** instruction).

Exceptions

The contents of the specified memory location and the program counter value in any return block are both undefined.

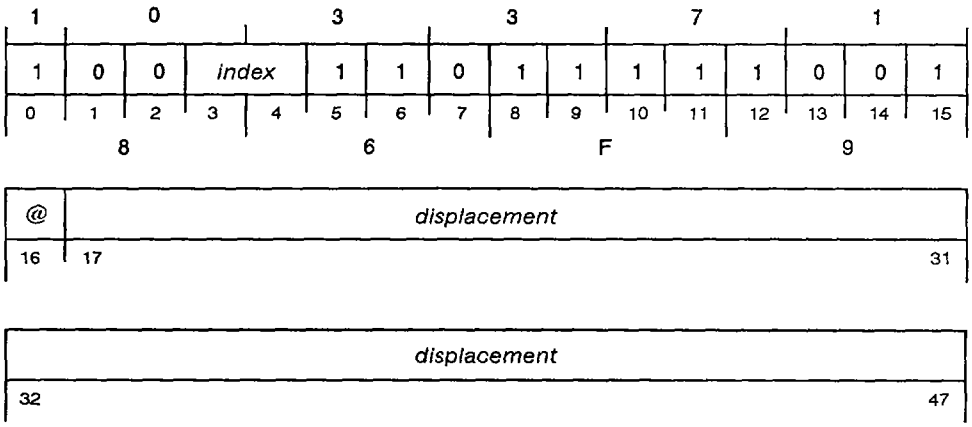
If a fixed-point overflow fault occurs while incrementing the DO-loop variable, contents of memory location and PC word in return block are undefined. (AC0 in fixed-point fault handler properly references address of DO-loop instruction.)

Example

```
WSUB   0,0            ;Get a 0.
LWSTA  0,INDEX        ;Initialize the counter in memory.
LOOP:  NLDAI 5,0      ;Maximum index value.
       LWDO 0,END,INDEX          ;Start of the DO-loop.
       .
       .              ;New index value is in AC0 and may be
       .              ;used by computations in the loop.
       .
       WBR LOOP
END:   . . .          ;Loop was executed 5 times.
       .
       .
       .
INDEX: .DWORD 0       ;Index value.
```

Wide Decrement and Skip if Zero (Long Displacement) LWDSZ

LWDSZ [@]*displacement*[,*index*]
(result ≠ 0 return)
(result = 0 return)



Function: (E) - 1 -> (E)
If resulting (E) = 0 then skip

Parameters: None

LWDSZ calculates the effective address and decrements by one the unsigned 32-bit integer in this location. If the result is equal to zero, then the instruction skips the next sequential word.

LWDSZ executes in one indivisible memory cycle if the word to be decremented is located on a double-word boundary.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Unused	
CARRY	Unchanged	
Overflow	0	
PC	PC + 3	(result ≠ 0)
	PC + 4	(result = 0)
PSR	Unchanged	
Stack	Unchanged	

Related Instructions

ISZ, EISZ, LNISZ, LWISZ, XNISZ, XWISZ
Increment the contents of memory and skip if result is zero.

DSZ, EDSZ, LNDSZ, XNDSZ, XWDSZ
Decrement the contents of memory and skip if result is zero.

Exceptions

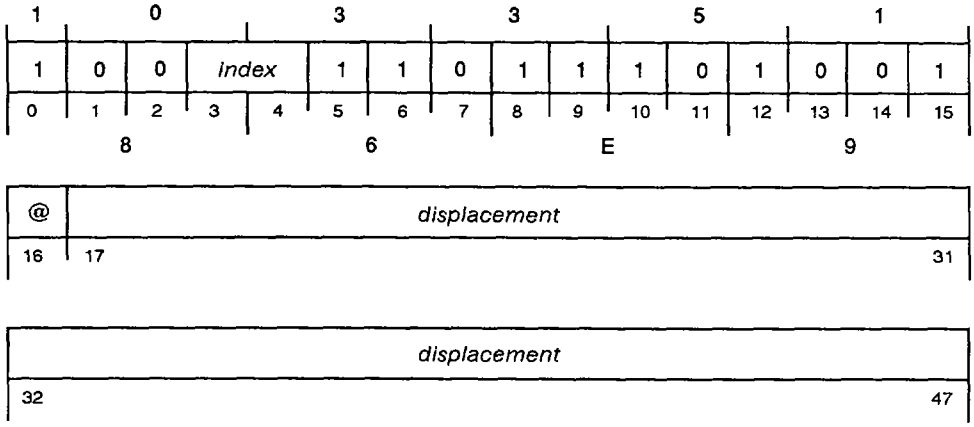
None

Example

```
NLDAI  5,0          ;Get a constant 5.
LWSTA  0,COUNTER    ;Initialize the loop counter.
LOOP:  . . .        ;Beginning of loop.
       .
       .
       .
LWDSZ  COUNTER      ;Decrement counter and skip if zero.
WBR    LOOP         ;We're not done yet.
. . .               ;We did the loop 5 times.
       .
       .
       .
COUNTER:.DWORD 0    ;Counter variable.
```

Wide Increment and Skip if Zero (Long Displacement) LWISZ

LWISZ [@]*displacement*[,*index*]
(result ≠ 0 return)
(result = 0 return)



Function: (E) + 1 -> (E)
If resulting (E) = 0 then skip

Parameters: None

LWISZ calculates the effective address and increments by one the unsigned 32-bit integer in this location. If the result is equal to zero, then the instruction skips the next sequential word.

LWISZ executes in one indivisible memory cycle if the word to be incremented is located on a double-word boundary.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 3 (result ≠ 0) PC + 4 (result = 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

ISZ, EISZ, LNISZ, XNISZ, XWISZ
Increment by one the contents of memory and skip if result is zero.

DSZ, EDSZ, LNDSZ, LWDSZ, XNDSZ, XWDSZ
Decrement by one the contents of memory and skip if result is zero.

Exceptions

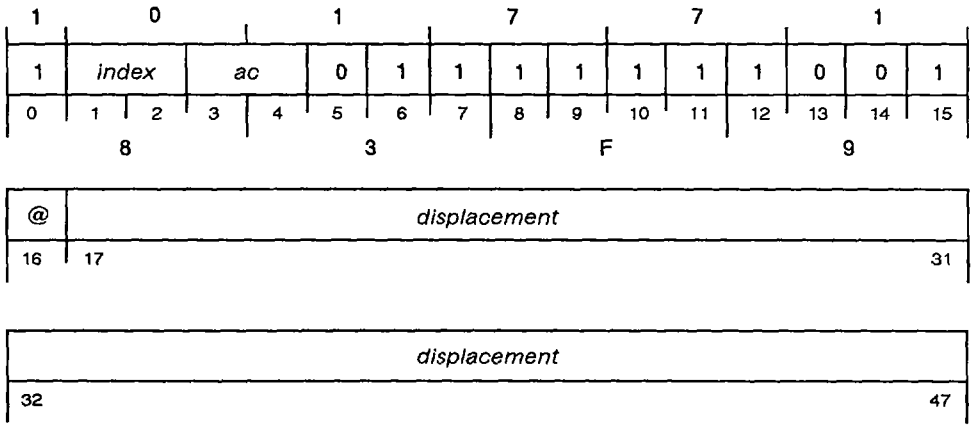
None

Example

```
    NLDAI  -5,0          ;Get a constant -5.
    LWSTA  0,COUNTER     ;Initialize the loop counter.
LOOP:. . .               ;Beginning of loop.
    .
    .
    .
    LWISZ  COUNTER       ;Increment counter and skip if zero.
    WBR    LOOP          ;We're not done yet.
    . . .      ;We did the loop 5 times.
    .
    .
    .
COUNTER:.DWORD 0         ;Counter variable.
```

Wide Load Accumulator (Long Displacement) LWLDA

LWLDA *ac*,[@]*displacement*[,*index*]



Function: (E) -> *ac*

Parameters: None

LWLDA calculates the effective address and loads the 32-bit value contained in this location into the specified accumulator.

Arguments

ac After execution, contains result.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 3
PSR	Unchanged
Stack	Unchanged

Related Instructions

LWSTA Wide Store Accumulator (Long Displacement)

Exceptions

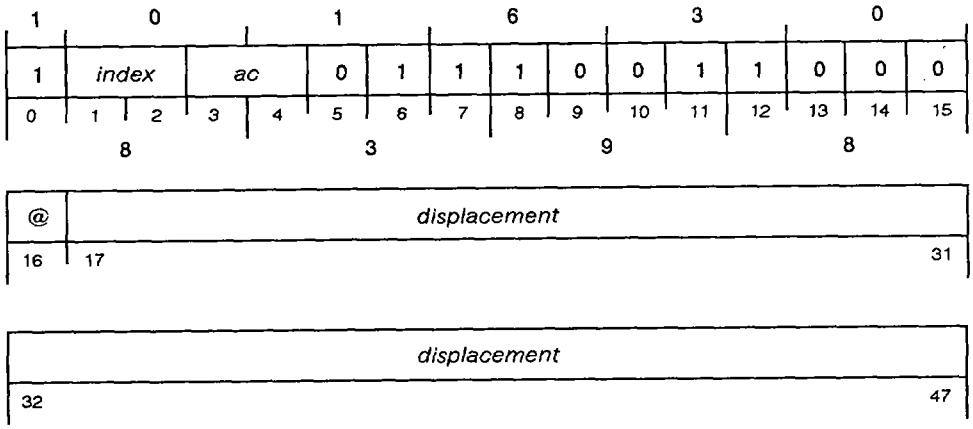
None

Example

```
LWLDA 0,DOUBLE_SOURCE          ;Get 32-bit value.
LWSTA 0,DOUBLE_DEST            ;Store 32 bit value.
```

Wide Multiply Memory Word (Long Displacement) LWMUL

LWMUL *ac*,[@]*displacement*[,*index*]



Function: (E) * *ac* -> *ac*

Parameters: None

LWMUL multiplies the signed 32-bit integer contained in the location specified by E by the signed 32-bit integer contained in the specified accumulator. Then it loads the least significant 32 bits of the result into the specified accumulator.

Arguments

ac Before execution, contains signed 32-bit integer.

After execution, contains least significant 32 bits of result.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3 Can be individually specified as *ac*; otherwise unused.

CARRY Unchanged

Overflow 1 if result outside specified range; otherwise 0.

PC PC + 3

PSR OVR set to 1 if overflow occurs.

Stack Unchanged

Related Instructions

LNMUL, XNMUL, XWMUL
Multiply an accumulator by the contents of memory.

Exceptions

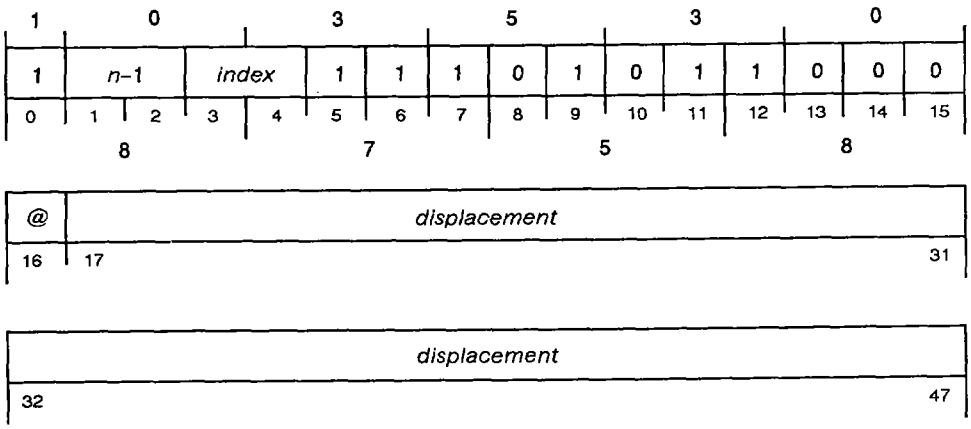
If the result is outside the range -2,147,483,648 to +2,147,483,647 inclusive, an overflow occurs, and PSR(OVR) is set to 1.

Example

```
LWLDA  0,FIRST     ;Get one value (32 bits).
LWMUL  0,SECOND    ;Multiply by the second value (32-bit
                   ;arithmetic).
LWSTA  0,RESULT    ;Store the double word result.
```

Wide Subtract Immediate (Long Displacement) LWSBI

LWSBI *n*,[@]*displacement*[,*index*]



Function: (E) - *n* -> (E)
ALU CRY -> CRY

Parameters: None

LWSBI subtracts an integer in the range of 1 to 4 from the signed 32-bit integer contained in the specified memory location.

Arguments

n Integer in range 1 to 4.

Assembler takes coded value of *n* and subtracts one from it before placing it in immediate field. Thus, programmer should code exact value to be subtracted.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Unused
CARRY	Set with value of ALU CARRY
Overflow	1 if ALU overflow
PC	PC + 3
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

LNSBI, XNSBI, XWSBI
Subtract an immediate value from the contents of memory.

Exceptions

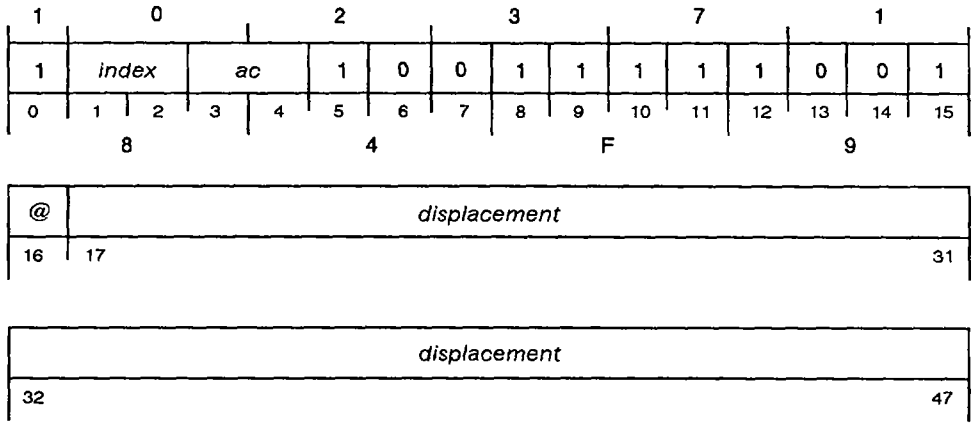
None

Example

```
    LWSBI 2,COUNTER     ;Decrement by 2 a counter in memory.
    ...
COUNTER: .DWORD 0       ;32-bit counter.
```

Wide Store Accumulator (Long Displacement) LWSTA

LWSTA *ac*,[@]*displacement*[,*index*]



Function: *ac* -> (E)

Parameters: None

LWSTA calculates the effective address and stores a copy of the 32-bit contents of the specified accumulator into this location.

Arguments

ac Contains 32-bit value.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 3
PSR	Unchanged
Stack	Unchanged

Related Instructions

LWLDA Wide Load Accumulator (Long Displacement)

Exceptions

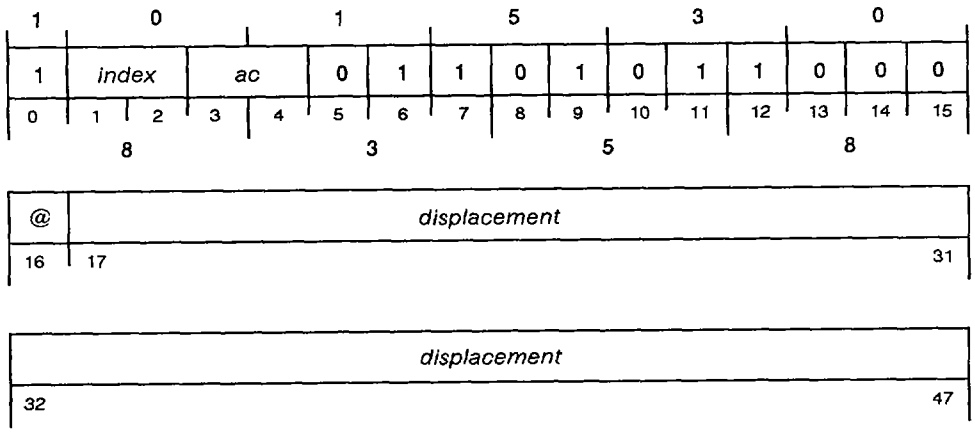
None

Example

```
LWLDA 0,DOUBLE_SOURCE          ;Get 32-bit value.
LWSTA 0,DOUBLE_DEST            ;Store 32-bit value.
```

Wide Subtract Memory Word (Long Displacement) LWSUB

LWSUB *ac*,[@]*displacement*[,*index*]



Function: *ac* - (E) -> *ac*
ALU CRY -> CRY

Parameters: None

LWSUB subtracts the signed 32-bit integer contained in the specified memory location from the signed 32-bit integer contained in the specified accumulator. Then it loads the result into the specified accumulator.

Arguments

ac Before execution, contains signed 32-bit integer.
After execution, contains result.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Set with value of ALU CARRY
Overflow	1 if ALU overflow
PC	PC + 3
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

LNSUB, XNSUB, XWSUB Subtract the contents of memory from an accumulator.

Exceptions

None

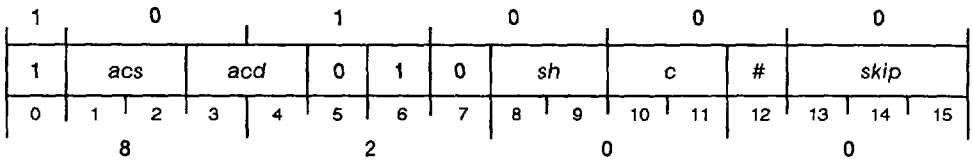
Example

```
LWLDA 0,FIRST     ;Get one value (32 bits).
LWSUB 0,SECOND    ;Subtract the second value (32-bit arithmetic).
LWSTA 0,RESULT    ;Store the double word result.
```

Move MOV

ECLIPSE Instruction

MOV[*c*][*sh*][*#*] *acs,acd*[,*skip*]
(*skip* false return)
(*skip* true return)



Function: *acs -> acd*

Parameters: None

MOV initializes carry to the specified value and places the contents of *acs* in the shifter. Then it performs the specified shift operation and loads the result of the shift into *acd* if the no-load bit is 0.

Arguments

[*c*] Processor determines effect of CARRY flag (*c*) on initial value of CARRY before performing operation (opcode). Following table gives values of *c* and bits 10 and 11 and specifies operation.

Symbol [*c*]	Bits 10-11	Operation
omitted	0 0	Leave CARRY unchanged
Z	0 1	Initialize CARRY to 0
O	1 0	Initialize CARRY to 1
C	1 1	Complement CARRY

[*sh*] Processor shifts CARRY flag and 16 data bits after performing operation. Processor can shift bits left or right one bit position or can swap the two bytes. Following table gives values of *sh* and bits 8 and 9 and specifies shift operation.

Symbol [*sh*]	Bits 8-9	Shift Operation
omitted	0 0	Do not shift the result
L	0 1	Shift left
R	1 0	Shift right
S	1 1	Swap the two 8-bit bytes

[*#*] Except with no-load option (#), processor loads result of shift operation into destination accumulator. No-load option can test result of operation without destroying destination accumulator contents. Following table gives values of no-load option and bit 12 and specifies operation.

Symbol [#]	Bit 12	Operation
omitted	0	Load result into *acd*
#	1	Do not load result; restore initial CARRY flag

acs(16-31) Before execution, contains 16-bit value.

After execution, contents unchanged unless *acs* and *acd* are same accumulator.

acd(16-31) Before execution, contains 16-bit value.

After execution, contains result if no-load bit (#) is 0.

[*skip*] Processor skips next instruction if condition test true. Following table gives test conditions as specified by bits 13 to 15 and specifies operation.

Symbol [*skip*]	Bits 13-15	Operation
omitted	0 0 0	Never skip
SKP	0 0 1	Always skip
SZC	0 1 0	Skip if CARRY is 0
SNC	0 1 1	Skip if CARRY is not 0
SZR	1 0 0	Skip if result is 0
SNR	1 0 1	Skip if result is not 0
SEZ	1 1 0	Skip if either CARRY or result 0
SBN	1 1 1	Skip if both CARRY and result not 0

Skip omits next sequential 16-bit word. Make sure that skip does not transfer control to point within 32-bit or longer instruction.

Registers, Flags, and Stacks

AC0–AC3 Can be individually specified as *acs* and *acd*; otherwise unused.

CARRY Operation leaves initial CARRY unchanged unless [*c*] option specified. If right or left shift occurs, final resulting CARRY is bit shifted into CARRY.

Overflow 0

PC PC + 1 (false exit)
PC + 2 (true exit)

PSR Unchanged

Stack Unchanged

Related Instructions

WMOV Wide Move

Exceptions

Do not specify **MOV** with no-load option (#) in combination with either never skip or always skip option. Instruction may not end in 1000_2 or 1001_2 (reserved for other instructions).

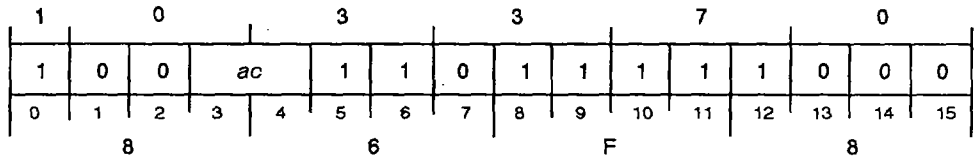
Example

```
    ;This subroutine appends one or two nulls to the end of a string,
    ;filling to a word boundary.
    ;
    ;Calling conventions:          XJSR NFILL
    ;                              <return>
    ;
    ;       AC1 = Byte pointer to start of string
    ;       AC2 = Length of string
    ;       AC3 = Return address
NFILL:  WPSH        3,3         ;Save return address.
        WSUB        3,3         ;Get a zero.
        WADD        2,1         ;Get end of string.
        WSTB        1,3         ;Append a null.
        WINC        1,1         ;Bump a pointer.
        MOVR#       1,1,SZC     ;Check if odd (middle of word).
        WSTB        1,3         ;Yes, append another null.
        LDAFP       3           ;AC3 contains frame pointer.
        WPOPJ                   ;Return.
```

Modify Stack Pointer MSP

ECLIPSE Instruction

MSP *ac*



Function: sp + *ac* -> sp

Parameters: None

MSP changes the value of the narrow stack pointer by the specified value and tests for potential stack overflow. The signed 16-bit integer in the specified accumulator is added to the current value of the stack pointer and the result is placed in reserved memory page zero location 40_8.

Arguments

ac(16-31)	Before execution, contains signed 16-bit integer to be added to stack pointer.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Narrow stack pointer updated by specified value.

Related Instructions

WMSP	Wide Modify Stack Pointer

Exceptions

If the computed result stored in location 40_8 is greater than the stack limit, the value in location 40_8 is changed back to its original value, and a standard return block is pushed, including current value of program counter (address of the **MSP** instruction). The stack pointer is updated with the value used to push the return block, the program counter is loaded with the starting address of the stack fault routine, and control transfers to the stack fault routine.

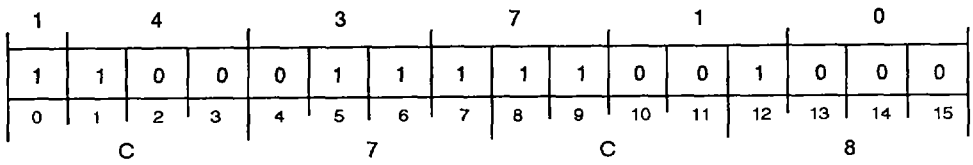
Example

```
ELDA   0,FIVE        ;Get a constant 5.
MSP    0             ;Increment the stack pointer by 5 words.
```

Unsigned Multiply MUL

ECLIPSE Instruction

MUL



Function: AC1 * AC2 + AC0 -> AC0[# high]&AC1[# low]

Parameters: None

MUL multiplies two unsigned 16-bit integers, each in an accumulator, and adds the 32-bit intermediate result to an unsigned 16-bit integer in a third accumulator. The result is an unsigned 32-bit integer in two accumulators.

Arguments

None

Registers, Flags, and Stacks

AC0(16-31)	Before execution, contains unsigned 16-bit number to be added to intermediate result to produce final result.
	After execution, contains high-order 16 bits of result.
AC1(16-31)	Before execution, contains unsigned 16-bit number.
	After execution, contains low-order 16 bits of result.
AC2(16-31)	Before execution, contains unsigned 16-bit number.
	After execution, contents unchanged.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

MULS	Signed Multiply
WMULS	Wide Signed Multiply
NMUL	Narrow Multiply
WMUL	Wide Multiply

Exceptions

None

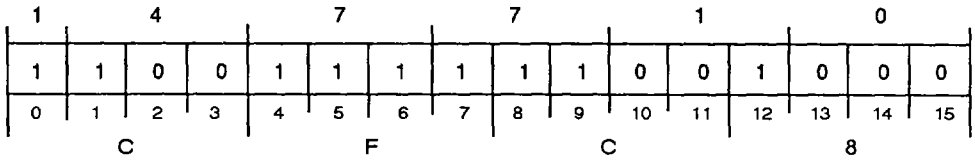
Example

```
ELDA   1,MULTIPLICAND      ;Get one number to multiply.
ELDA   2,MULTIPLIER        ;Get the other number to multiply.
ELDA   0,ADDEND            ;Get the number to add to the product.
MUL                        ;Multiply and add.
ESTA   0,HIGH_RESULT       ;Store the high order result.
ESTA   1,LOW_RESULT        ;Store the low order result.
```

Signed Multiply MULS

ECLIPSE Instruction

MULS



Function: AC1 * AC2 + AC0 -> AC0[2# high]&AC1[2# low]

Parameters: None

NOTE: Upon instruction completion, AC0 (bit 16) = sign

MULS multiplies two signed 16-bit integers, each in an accumulator, and adds the 32-bit intermediate result to the signed 16-bit contents of a third accumulator. The result is a signed 32-bit integer in two accumulators.

Arguments

None

Registers, Flags, and Stacks

AC0(16-31)	Before execution, contains signed 16-bit integer to be added to intermediate result to produce final result.
	After execution, contains high-order 16 bits of result. Bit 16 contains sign.
AC1(16-31)	Before execution, contains signed 16-bit integer.
	After execution, contains low-order 16 bits of result.
AC2(16-31)	Before execution, contains signed 16-bit integer.
	After execution, contents unchanged.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

MUL	Unsigned Multiply
WMULS	Wide Signed Multiply
NMUL	Narrow Multiply
WMUL	Wide Multiply

Exceptions

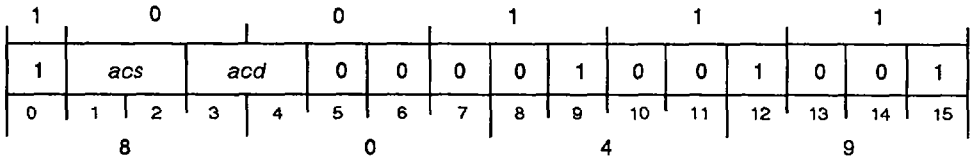
None

Example

```
ELDA   1,MULTIPLICAND          ;Get one number to multiply.
ELDA   2,MULTIPLIER            ;Get the other number to multiply.
ELDA   0,ADDEND                ;Get the number to add to the product.
MULS                           ;Multiply and add (signed).
ESTA   0,HIGH_RESULT           ;Store the high order result.
ESTA   1,LOW_RESULT            ;Store the low order result.
```

Narrow Add NADD

NADD *acs,acd*



Function: *acs* + *acd* -> *acd*

Parameters: None

NADD adds the signed 16-bit integer in *acs* to the signed 16-bit integer in *acd* and stores the 32-bit sign-extended result in *acd*.

Arguments

acs(16-31)	Before execution, contains signed 16-bit integer.
	After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd(16-31)	Before execution, contains signed 16-bit integer.
	After execution, contains result sign-extended to 32 bits.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Set with value of ALU CARRY (16-bit operation).
Overflow	1 if an ALU overflow (16-bit operation).
PC	PC + 1
PSR	OVR set to 1 if overflow occurs
Stack	Unchanged

Related Instructions

ADD	Add
WADD	Wide Add

Exceptions

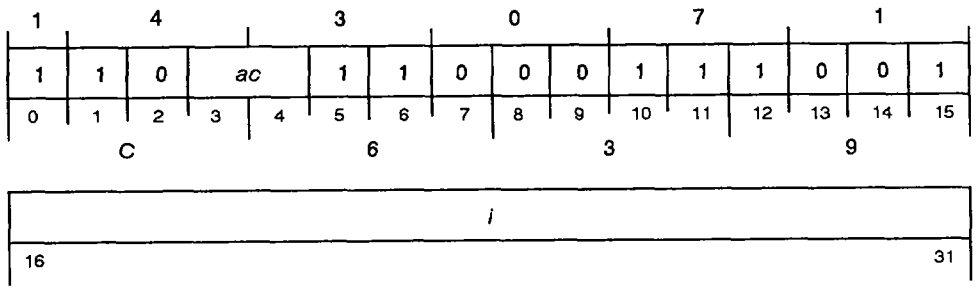
None

Example

```
XNLDA 0,FIRST      ;Get one value.
XNLDA 2,SECOND     ;Get second value.
NADD  0,2          ;Add.
XNSTA 2,RESULT     ;Store the result.
```

Narrow Extended Add Immediate NADDI

NADDI *i,ac*



Function: *i + ac -> ac*

Parameters: None

NADDI adds a signed 16-bit integer contained in the immediate field to the signed 16-bit integer in the specified accumulator.

Arguments

i	Signed 16-bit immediate value
ac(16-31)	Before execution, contains signed 16-bit integer.
	After execution, contains result sign-extended to 32-bits.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Set according to value of ALU CARRY (16-bit operation).
Overflow	1 if there is an ALU overflow.
PC	PC + 2
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

ADDI, WADDI, WNADI	Add a signed 16- or 32-bit immediate value to an accumulator
ADI, NADI, WADI	Add a 2-bit immediate value to an accumulator

Exceptions

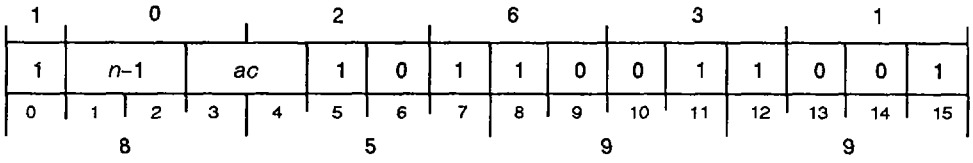
None

Example

```
XNLDA 3,FIRST      ;Get first value.
NADDI 300.,3       ;Add a constant 300. to AC3.
XNSTA 3,RESULT     ;Store the result.
```

Narrow Add Immediate NADI

NADI *n,ac*



Function: *n* + *ac* -> *ac*

Parameters: None

NADI adds an integer in the range of 1 to 4 to the signed 16-bit integer in the specified accumulator, sign-extending the result 32 bits.

Arguments

n	Integer in range 1-4. Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field. Thus, programmer should code exact value to be added.
ac(16-31)	Before execution, contains signed 16-bit integer. After execution, contains result sign-extended to 32 bits.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Set with value of ALU carry (16 bit operation).
Overflow	1 if there is ALU overflow (16 bit operation).
PC	PC + 1
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

ADI, WADI	Add a 2-bit immediate value to an accumulator.
ADDI, NADDI, WADDI, WNADI	Add a signed 16- or 32-bit immediate value to an accumulator.

Exceptions

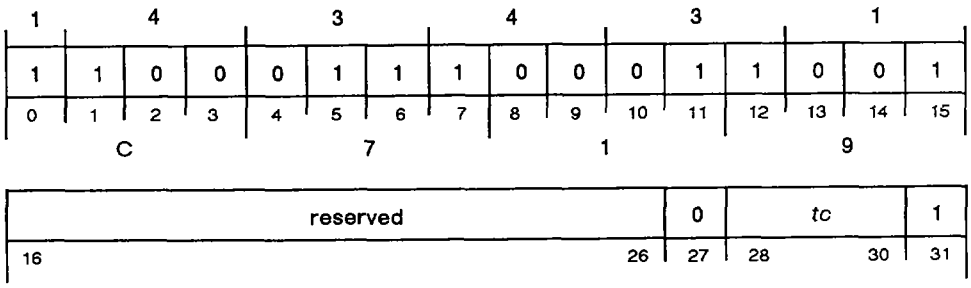
None

Example

```
XNLDA 3,FIRST      ;Get first value.
NADI  4,3          ;Add a constant 4 to AC3.
XNSTA 3,RESULT     ;Store the result.
```

Narrow Backward Search Queue and Skip NBStc

NBS*tc*
(unsuccessful return)
(interrupted return)
(successful return)



Function: Search from @(AC1) to Q head for @(AC1 + AC3)
= 16-bit test (*tc*)

= all 0	{AC}
= all 1	{AS}
= (WSP)	{E}
<= (WSP)	{GE}
>= (WSP)	{LE}
≠ (WSP)	{NE}
= some 0s	{SC}
= some 1s	{SS}

Parameters: AC1 = E(first queue data element - E(Q element)
AC3 = 2#(word offset) --> unchanged
(WSP) = mask --> unchanged

NBS*tc* searches backward through a queue, examining a 16-bit data field. The processor locates the beginning queue element by calculating the effective address (AC1). The data field examined in this element is located by adding to AC1 the word offset in AC3. The result is then compared to a 16-bit mask. The search continues until the processor reaches either the head of the queue or a data element that meets the test condition (*tc*).

Arguments

tc Bits 28-30 specify search condition.

tc Value	Bits 28-30 Encoding	Meaning
SS	0 0 0	Some of sampled test location bits 1.
SC	0 0 1	Some of sampled test location bits 0.
AS	0 1 0	All of sampled test location bits 1.
AC	0 1 1	All of sampled test location bits 0.
E	1 0 0	Mask and test location equal.
GE	1 0 1	Mask greater than or equal to test location.
LE	1 1 0	Mask less than or equal to test location.
NE	1 1 1	Mask and test location not equal.

For **E, GE, LE,** and **NE** test conditions, instruction treats values contained in mask and in test location as unsigned 16-bit integers.

Registers, Flags, and Stacks

AC0	Unused
AC1	Before execution, contains effective address. With AC3, identifies location in data field as beginning data element in queue search. Processor increments AC1 for each data element it tests. If search succeeds, contains effective address of data element. New beginning pointer must be placed in AC1 if search is to continue through rest of queue. (If this is not done, and both the search condition and the examined data field in the element remain unchanged, continuing the search will result in this element being found again.) If search fails, contains effective address of last data element searched. If processor interrupts search (after unsuccessful search or another interrrupt only), contains effective address of next data element to be searched.
AC2	Unused
AC3(16–31)	Before execution, contains signed 16-bit integer for word offset which is added to AC1 to identify location of data field in beginning data element in queue search. After execution, for all returns, contents unchanged.
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 2 (unsuccessful exit) Processor honors interrupts between completed search and execution of PC + 2 instruction. PC + 3 (interrupted exit) Processor honors interrupts between occurrence of interrupt and execution of PC + 3 instruction. PC + 4 (successful exit) Processor does not honor interrupts between completed search and execution of PC + 4 instruction.
PSR	Unchanged
Stack	Before execution, top wide stack double word bits 16–31 contain mask, identifying test location bits to sample. After execution, for all returns, wide stack unchanged.

Related Instructions

Queue Management
Use these instructions to insert, delete, and test queue entries.

Exceptions

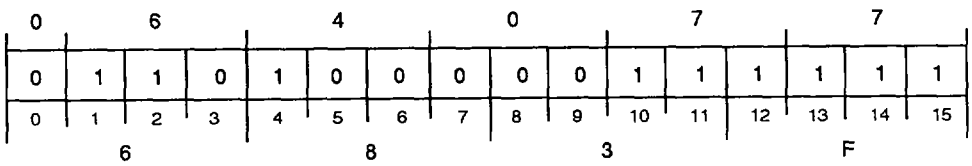
None

Example

```
NBSE
WBR QERROR          ;Unsuccessful exit. Jump to an error handler.
WBR ...             ;Interrupt exit. Jump to fix it.
...                 ;Successful exit. Continue instruction stream.
```

Narrow Load CPU Identification NCLID

NCLID



Function: CPU id -> AC0&AC1&AC2
model number -> AC0
microcode revision -> AC1
memory size -> AC2

Parameters: None

NCLID loads CPU identification into three accumulators (LEF mode and I/O protection must be disabled).

Arguments

None

Registers, Flags, and Stacks

AC0(16-31)	After execution, contains model number.
AC1(16-31)	After execution, contains microcode revision.
AC2(16-31)	After execution, contains memory size.
AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

ECLID, LCPID Return CPU identification information.

Exceptions

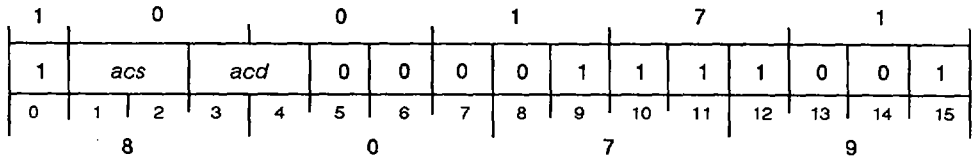
None

Example

```
NCLID               ;Get the various parts of the CPU id.
XNSTA 0,MODEL       ;Store the model number in memory.
XNSTA 1,UCODE       ;Store the microcode revision number in
                    ;memory.
XNSTA 2,MEM         ;Store the memory size number in memory.
```

Narrow Divide NDIV

NDIV *acs,acd*



Function: *acd* / *acs* -> *acd*(quotient)

Parameters: None

NOTE: If *acs* = 0, or result overflows; *overflow* = 1 and *acd* = unchanged

NDIV sign-extends the signed 16-bit integer in *acd* to 32 bits, and divides it by the signed 16-bit integer in *acs*. If the quotient is within the range of -32,768 to +32,767 inclusive, it sign-extends the lower 16 bits of the result to 32 bits and places these in *acd*.

Arguments

acs(16-31)	Before execution, contains signed 16-bit divisor.
	After execution, contents unchanged.
acd	Before execution, contains signed 16-bit dividend.
	After execution, contains sign-extended 32-bit result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	1 if quotient exceeds specified range or *acs* is 0; otherwise, 0.
PC	PC + 1
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

DIV, DIVS, DIVX, WDIV, WDIVS
Divide an accumulator by an accumulator.

Exceptions

If quotient is outside the range, -32,768 to +32,767, or if *acs* initially contains zero, an overflow occurs. PSR(OVR) is set to 1, and *acd* remains unchanged.

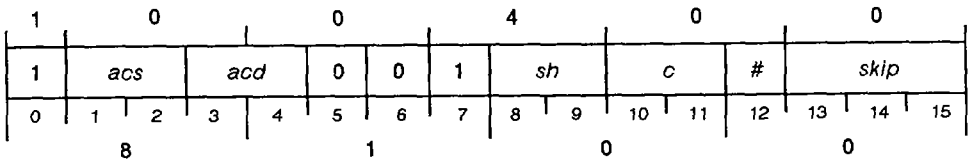
Example

```
XNLDA 2,DIVIDEND   ;Get the dividend.
XNLDA 3,DIVISOR    ;Get the divisor.
NDIV  3,2          ;Divide.
XNSTA 2,RESULT     ;Store the result.
```

Negate NEG

ECLIPSE Instruction

NEG[*c*][*sh*][*#*] *acs*,*acd*[,*skip*]
(*skip* false return)
(*skip* true return)



Function: *-acs -> acd*

Parameters: None

NOTE: If $\overline{acs} = 0$, CRY -> CRY

NEG initializes CARRY to specified value and places the negative of the signed 16-bit integer in *acs* into shifter. Then it performs specified shift operation and places result in *acd* if no-load bit is 0.

Arguments

[*c*] Processor determines effect of CARRY flag (*c*) on initial value of CARRY before performing operation (opcode). Following table gives values of *c* and bits 10 and 11 and specifies operation.

Symbol [*c*]	Bits 10-11	Operation
omitted	0 0	Leave CARRY unchanged
Z	0 1	Initialize CARRY to 0
O	1 0	Initialize CARRY to 1
C	1 1	Complement CARRY

[*sh*] Processor shifts CARRY flag and 16 data bits after performing operation. Processor can shift bits left or right one bit position or can swap the two bytes. Following table gives values of *sh* and bits 8 and 9 and specifies shift operation.

Symbol [*sh*]	Bits 8-9	Shift Operation
omitted	0 0	Do not shift the result
L	0 1	Shift left
R	1 0	Shift right
S	1 1	Swap the two 8-bit bytes

[*#*] Except with no-load option (*#*), processor loads result of shift operation into destination accumulator. No-load option can test result of operation without destroying destination accumulator contents. Following table gives values of no-load option and bit 12 and specifies operation.

Symbol [*#*]	Bit 12	Operation
omitted	0	Load result into *acd*
#	1	Do not load result; restore initial CARRY flag

acs(16-31) Before execution, contains signed 16-bit integer.

After execution, contents unchanged unless *acs* and *acd* are same accumulator.

acd(16-31) After execution, contains result if no-load bit (#) is 0.

[*skip*] Processor skips next instruction if condition test true. Following table gives test conditions as specified by bits 13 to 15 and specifies operation.

Symbol [*skip*]	Bits 13-15	Operation
omitted	0 0 0	Never skip
SKP	0 0 1	Always skip
SZC	0 1 0	Skip if CARRY is 0
SNC	0 1 1	Skip if CARRY is not 0
SZR	1 0 0	Skip if result is 0
SNR	1 0 1	Skip if result is not 0
SEZ	1 1 0	Skip if either CARRY or result 0
SBN	1 1 1	Skip if both CARRY and result not 0

Skip omits next sequential 16-bit word. Make sure that skip does not transfer control to point within 32-bit or longer instruction.

Registers, Flags, and Stacks

AC0-AC3 Can be individually specified as *acs* and *acd*; otherwise unused.

CARRY If *acs* contains 0 (so that operation produces a carry out of the high-order bit), then initial CARRY will be complemented. Then, if left or right shift occurs, final resulting CARRY is bit shifted into CARRY.

Overflow 0

PC PC + 1 (false exit)
PC + 2 (true exit)

PSR Unchanged

Stack Unchanged

Related Instructions

NNEG Narrow Negate

WNEG Wide Negate

Exceptions

If *acs* initially contains zero, CARRY is complemented. (See also CARRY)

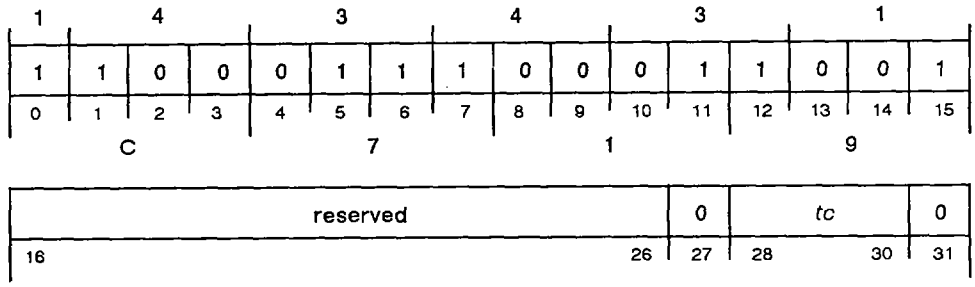
Do not specify **NEG** with no-load option (#) in combination with either never skip or always skip option. Instruction may not end in 1000_2 or 1001_2 (reserved for other instructions).

Example

```
NEG   0,0          ;Negate the value in AC0[16-31].
```

Narrow Forward Search Queue and Skip NFStc

NFS*tc*
(unsuccessful return)
(interrupted return)
(successful return)



Function: Search from @(AC1) to Q tail for @(AC1 + AC3)
= 16-bit test (*tc*)

=all 0	{AC}
=all 1	{AS}
=(wsp)	{E}
<=(wsp)	{GE}
>=(wsp)	{LE}
≠(wsp)	{NE}
=some 0s	{SC}
=some 1s	{SS}

Parameters: AC1 = E(first queue data element – E(Q element -- See Note)
AC3 = 2#(word offset) --> unchanged
(wsp) = mask word --> unchanged

NOTE: The call sequence for the Search Queue instruction is:
Search Queue instruction
Unsuccessful Return E(last element searched) --> AC1
Interrupt Return E(next element to search) --> AC1
Successful Return E(last element searched) --> AC1

NFS*tc* searches forward through a queue, examining a 16-bit data field. The processor locates the beginning queue element by calculating the effective address (AC1). The data field examined in this element is located by adding to AC1 the word offset in AC3. The result is then compared to a 16-bit mask. The search continues until the processor reaches either the tail of the queue or a data element that meets the test condition (*tc*).

Arguments

tc Bits 28–30 specify search condition.

tc Value	Bits 28-30 Encoding	Meaning
SS	0 0 0	Some of sampled test location bits 1.
SC	0 0 1	Some of sampled test location bits 0.
AS	0 1 0	All of sampled test location bits 1.
AC	0 1 1	All of sampled test location bits 0.
E	1 0 0	Mask and test location equal.
GE	1 0 1	Mask greater than or equal to test location.
LE	1 1 0	Mask less than or equal to test location.
NE	1 1 1	Mask and test location not equal.

NOTE: For **E, GE, LE,** *and* **NE** *test conditions, instruction treats values contained in mask and in test location as unsigned 16-bit integers.*

Registers, Flags, and Stacks

AC0	Unused
AC1	Before execution, contains effective address. With AC3, identifies location in data field as beginning data element in queue search. Processor increments AC1 for each data element it tests. If search succeeds, contains effective address of data element. New beginning pointer must be placed in AC1 if search is to continue through rest of queue. (If this is not done, and both the search condition and the examined data field in the elment remain unchanged, continuing the search will result in this element being found again.) If search fails, contains effective address of last data element searched. If processor interrupts search (after unsuccessful search or another interrrupt only), contains effective address of next data element to be searched.
AC2	Unused
AC3(16-31)	Before execution, contains signed 16-bit integer for word offset which is added to AC1 to identify location of data field in beginning data element in queue search. After execution, for all returns, contents unchanged.
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 2 (unsuccessful exit) Processor honors interrupts between completed search and execution of PC + 2 instruction. PC + 3 (interrupted exit) Processor honors interrupts between occurrence of interrupt and execution of PC + 3 instruction. PC + 4 (successful exit) Processor does not honor interrupts between completed search and execution of PC + 4 instruction.
PSR	Unchanged
Stack	Before execution, top wide stack double word bits 16-31 contain mask, identifying test location bits to sample. After execution, for all returns, wide stack unchanged.

Related Instructions

Queue Management
Use these instructions to insert, delete, and test queue entries.

Exceptions

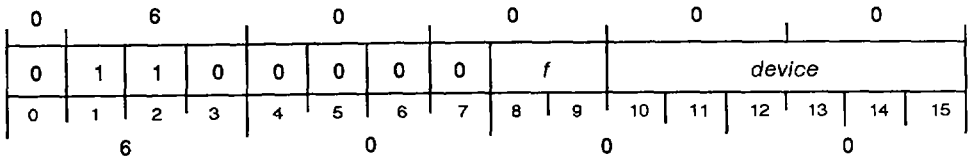
None

Example

```
NFSE
WBR QERROR          ;Unsuccessful exit. Jump to an error handler.
WBR ...             ;Interrupt exit. Jump to fix it.
...                 ;Successful exit. Continue instruction stream.
```

No I/O Transfer NIO

NIO[*f*] *device*



Function: [*f*] -> BUSY,DONE flags

Parameters: None

NIO sets the BUSY and DONE flags in the specified *device* on the default I/O channel (IOC) according to the function specified by *f*; no other operations take place.

Arguments

f Specify from **S**, **C**, and **P** for desired I/O device flag control as follows:

f	BUSY	DONE
(option omitted)	No effect	No effect
S	Set to a 1	Set to a 0
C	Set to a 0	Set to a 0
P	Pulses a special I/O bus control line	

device Specify either mnemonic or device code for desired I/O device.

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

DIA, DIB, DIC Transfer data from buffer of I/O device to an accumulator.

DOA, DOB, DOC
Transfer data from an accumulator to the buffer of an I/O device.

Exceptions

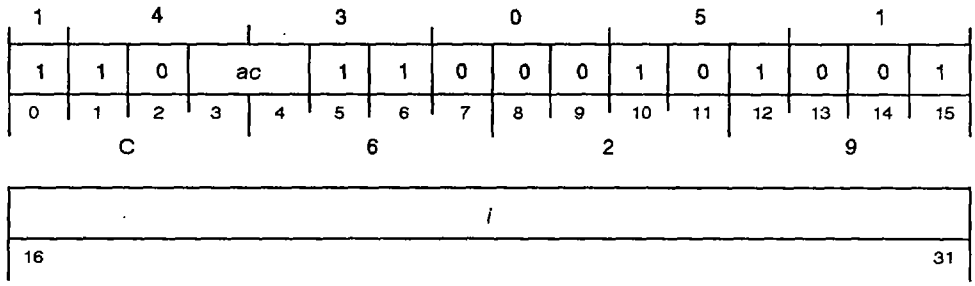
The **NIO[*f*] CPU** instructions are reserved or assigned specific functions. For instance, **NIOS CPU (INTEN)** is the Interrupt Enable instruction.

Example

```
NIOS  27        ;Start an operation on device 27 of the
                ;default IOC by setting the BUSY flag to one
                ;and the DONE flag to zero.
```

Narrow Load Immediate NLDAI

NLDAI *i,ac*



Function: *i -> ac*

Parameters: None

NLDAI sign-extends the signed 16-bit integer contained in the immediate field to 32 bits. Then it loads this result into the specified accumulator.

Arguments

i	Contains signed 16-bit integer.
ac	After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

WLDAI	Wide Load with Wide Immediate

Exceptions

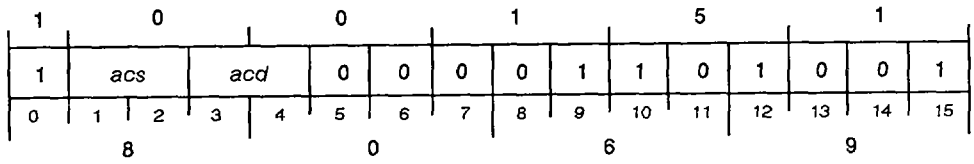
None

Example

```
NLDAI  -4,3          ;Load a -4 into AC3, then
WLSH   3,1           ;Divide contents of AC1 by 16.
```

Narrow Multiply NMUL

NMUL *acs,acd*



Function: *acs * acd -> acd*

Parameters: None

NMUL multiplies the signed 16-bit integer contained in *acd* by the signed 16-bit integer contained in *acs*. If the result is within the range of -32,768 to +32,767 inclusive, it sign extends the lower 16 bits of result to 32 bits and places the result in *acd*.

Arguments

acs(16-31)	Before execution, contains signed 16-bit integer.
	After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd(16-31)	Before execution, contains signed 16-bit integer.
	After execution, contains result sign-extended to 32 bits.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	1 if result exceeds specified range; otherwise 0.
PC	PC + 1
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

MUL, MULS, WMULS, WMUL
Multiply an accumulator by an accumulator.

Exceptions

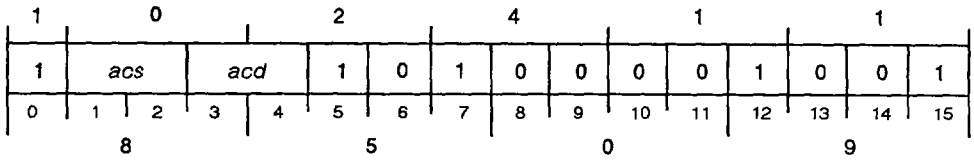
If result exceeds specified range (-32,768 to +32,767 inclusive), PSR(OVR) is set to 1.

Example

```
XNLDA 2,MULTIPLICAND          ;Get the multiplicand.
XNLDA 3,MULTIPLIER            ;Get the multiplier.
NMUL  3,2                     ;Multiply.
XNSTA 2,RESULT                ;Store the result.
```

Narrow Negate NNEG

NNEG *acs,acd*



Function: *-acs -> acd*

Parameters: None

NOTE: ALU carry -> CRY
If *acs* = 100000_8, overflow = 1

NNEG negates the signed 16-bit integer in *acs* by performing a two's complement subtract from zero. Then it sign-extends the 16 bit result to 32 bits and loads this into *acd*.

Arguments

acs(16-31)	Before execution, contains signed 16-bit integer.
	After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd	After execution, contains sign-extended 32-bit integer.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Set according to value of ALU carry.
Overflow	1 if the largest negative 16-bit integer (100000_8) is negated; otherwise 0.
PC	PC + 1
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

NEG	Negate
WNEG	Wide Negate

Exceptions

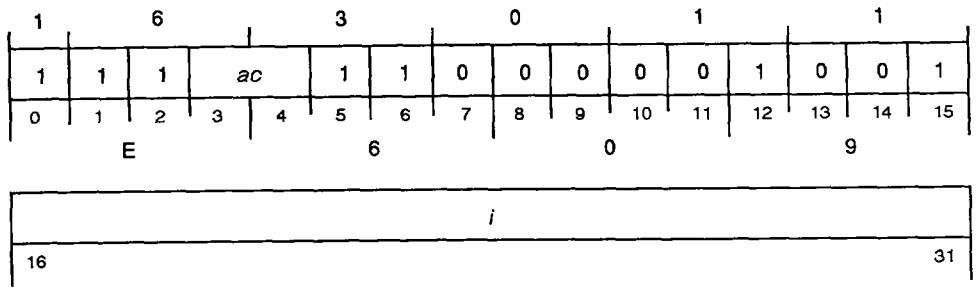
If largest negative 16-bit integer (100000_8) is negated, PSR(OVR) is set to 1.

Example

```
NNEG  0,0          ;Negate the value in AC0[16-31] and sign
                   ;extend the result to 32 bits.
```

Narrow Skip on All Bits Set in Accumulator NSALA

NSALA *i,ac*
(AND = non0 return)
(AND = 0 return)



Function: If *i* AND *ac* = 0 then skip

Parameters: None

NSALA logically ANDs the contents of the 16-bit immediate field with the complement of the least significant 16 bits contained in the specified accumulator and skips depending on the result.

Arguments

i	16-bit value
ac(16-31)	16-bit value

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2 (AND = non0) PC + 3 (AND = 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WSALA, NSALM, WSALM
Skip on all bits set in an accumulator or memory.

Exceptions

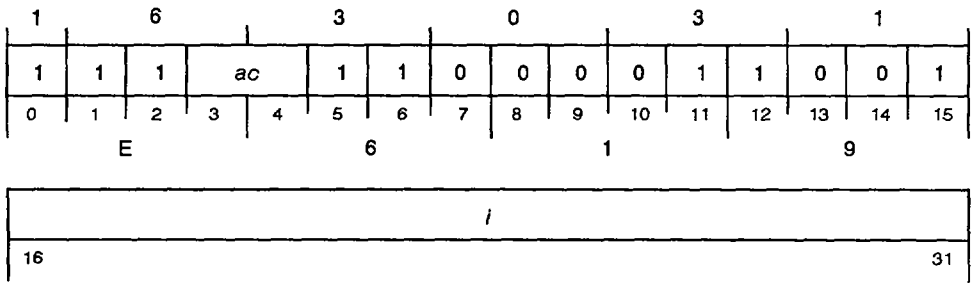
None

Example

```
    XNLDA 2,FLAGS       ;Get the flags word.
    NSALA 140002,2      ;Are bits 0, 1, and 14 all set?
    WBR   FAIL          ;No. One or more are zero.
    . . .               ;Yes. All bits are set.
    ...
FLAGS:  .WORD 0         ;Flags word.
```

Narrow Skip on All Bits Set in Memory Location — NSALM

NSALM *i,ac*
(AND = 0 return)
(AND ≠ 0 return)



Function: If *i* AND ($\overline{ac}$) = 0 then skip

Parameters: None

NSALM logically ANDs the contents of the 16-bit immediate field with the complement of the word addressed by the specified accumulator and skips the next sequential word on a zero result.

Arguments

i	16-bit value
ac	Contains word address of 16-bit value.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2 (AND = 0) PC + 3 (AND = non0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

NSALA, WSALM, WSALA
Skip on all bits set in accumulator or memory.

Exceptions

None

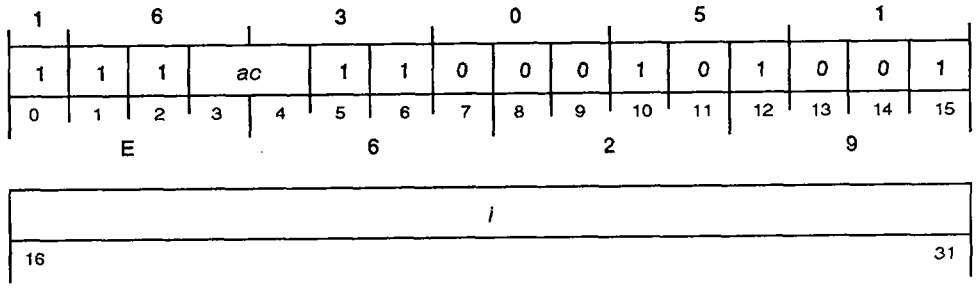
Example

```
        XLEF   2,FLAGS      ;Get address of the flags word.
        NSALM  140002,2     ;Are bits 0, 1, and 14 all set?
        WBR    FAIL         ;No. One or more are zero.
        . . .               ;Yes. All bits are set.
        ...

FLAGS:  .WORD  0            ;Flags word.
```

Narrow Skip on Any Bit Set in Accumulator NSANA

NSANA *i,ac*
(AND = 0 return)
(AND ≠ 0 return)



Function: If *i* AND *ac* ≠ 0 then skip

Parameters: None

NSANA logically ANDs the contents of the 16-bit immediate field with the least significant 16 bits contained in the specified accumulator and skips depending on the result.

Arguments

i	16-bit value
ac(16-31)	16-bit value

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2 (AND = 0) PC + 3 (AND ≠ 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

NSANM, WSANA, WSANM
Skip on any bit set in an accumulator or memory.

Exceptions

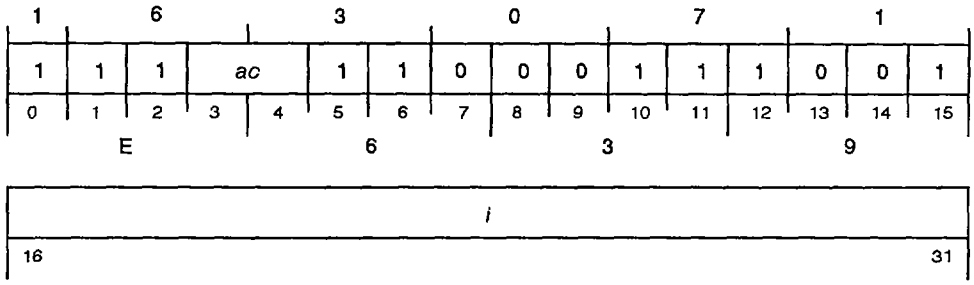
None

Example

```
        XNLDA  2,FLAGS      ;Get the flags word.
        NSANA  140002,2     ;Are any of bits 0, 1, and 14 set?
        WBR    FAIL         ;No. All three bits are zero.
        . . .               ;Yes. One or more of the three are set.
        ....
FLAGS:  .WORD  0            ;Flags word.
```

Narrow Skip on Any Bit Set in Memory Location — NSANM

NSANM *i,ac*
(AND = 0 return)
(AND ≠ 0 return)



Function: If *i* AND (*ac*) ≠ 0 then skip

Parameters: None

NSANM logically ANDs the contents of the 16-bit immediate field with the contents of the word addressed by the specified accumulator and skips depending on the result.

Arguments

i	16-bit value
ac	Contains word address of 16-bit value.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2 (AND = 0) PC + 3 (AND ≠0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

NSANA, WSANM, WSANA
Skip on any bit set in accumulator or memory.

Exceptions

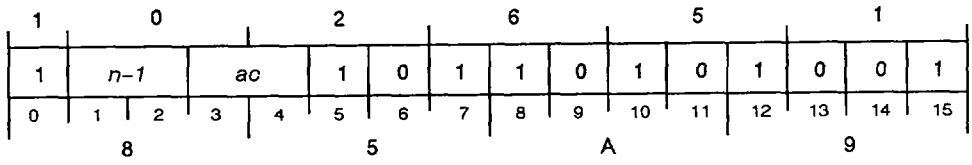
None

Example

```
    XLEF   2,FLAGS      ;Get the flags word.
    NSANM  140002,2     ;Are any of bits 0, 1, and 14 set?
    WBR    FAIL         ;No. All three bits are zero.
           . . .        ;Yes. One or more of the three are set.
    ....
FLAGS:     .WORD 0      ;Flags word.
```

Narrow Subtract Immediate NSBI

NSBI *n,ac*



Function: *ac* − *n* -> *ac*
ALU CRY -> CRY

Parameters: None

NSBI subtracts an integer in the range of 1 to 4 from the signed 16-bit integer contained in the specified accumulator. Then it stores the result in the specified accumulator, sign-extending it to 32 bits.

Arguments

n	Integer in range 1–4. Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field. Thus, programmer should code exact value to be subtracted.
ac(16–31)	Before execution, contains signed 16-bit integer. After execution, contains result, sign-extended to 32 bits.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Set with value of ALU CARRY.
Overflow	1 if an ALU overflow.
PC	PC + 1
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

SBI	Subtract Immediate
WSBI	Wide Subtract Immediate

Exceptions

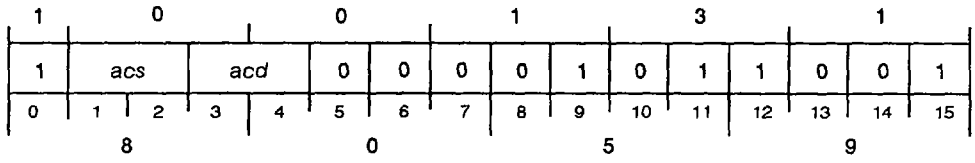
None

Example

```
XNLDA 3,FIRST      ;Get first value.
NSBI  4,3          ;Subtract a constant 4 from AC3.
XNSTA 3,RESULT     ;Store the result.
```

Narrow Subtract NSUB

NSUB *acs,acd*



Function: *acd - acs -> acd*

Parameters: None

NSUB subtracts the signed 16-bit integer contained in *acs* from the signed 16-bit integer contained in *acd*, placing the sign-extended result in *acd*.

Argument

acs(0-31) Before execution, contains signed 16-bit integer.

After execution, contents unchanged unless *acs* and *acd* are same accumulator.

acd(0-31) Before execution, contains signed 16-bit integer.

After execution, contains result sign-extended to 32-bits.

Registers, Flags, and Stacks

AC0-AC3 Can be individually specified as *acs* and *acd*; otherwise unused.

CARRY Set with value of ALU CARRY.

Overflow 1 if there is an ALU overflow.

PC PC + 1

PSR OVR set to 1 if overflow occurs.

Stack Unchanged

Related Instructions

SUB Subtract

WSUB Wide Subtract

Exceptions

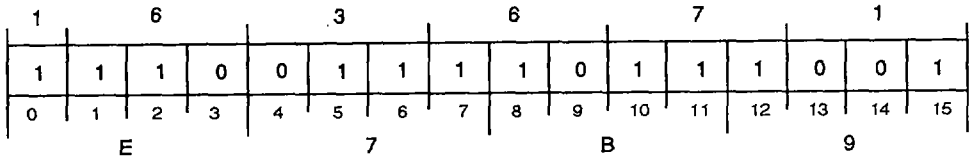
None

Example

```
XNLDA 0,FIRST      ;Get one value.
XNLDA 2,SECOND     ;Get second value.
NSUB  2,0          ;Subtract second value from first.
XNSTA 0,RESULT     ;Store the result.
```

OR Referenced Bits ORFB

ORFB



Function: Operate on groups of 16 referenced bits:
referenced bits ((AC1), (AC1) + 1,...,(AC1) + 15)
OR @(AC2) -> @(AC2)
0 -> referenced bits ((AC1), (AC1) + 1,...,(AC1) + 15)
Do until AC0 < 0

Parameters: AC0 = pageframe count -> all 1s
AC1 = pageframe #(13-31) -> (AC1 + (AC0 + 1) * 16)
AC2 = E(word string) -> (AC0 + 1) + AC2

NOTE: If AC0 = > 0 or = largest negative #, then pageframes reset.

ORFB reads and resets a specified string of framepage referenced bits. It inclusively ORs a string of framepage referenced bits with a string of words in memory; stores the result back in the memory string; and resets the string of referenced bits to 0.

Arguments

None

Registers, Flags, and Stacks

AC0	Before execution, contains signed 32-bit integer specifying number of words minus 1 to be processed in referenced bit string. A count of 0 or any negative count except 8000000_{16} causes one 16-bit word to be reset. With each word processed, count decrements by 1. After execution, count equals -1.
AC1(13-27)	Before execution, contains value specifying starting pageframe number for referenced bit string; bits 28-31 set to 0 (thus, each increment of pageframe number a multiple of 16). With each word processed, contents incremented by 16. After execution, contains original starting pageframe number plus referenced bit word count initially stored in AC0 plus 1.
AC2	Before execution, contains word starting address of compare word string; string inclusively ORed with reference bit string; results stored back in string. After each word processed, address incremented by 1. After execution, contains last address of string plus 1.
AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load with immediate	Use these instructions to place values into AC0 or AC1.
Load effective address	Use these instructions to place an address into AC2.
RRFB	Reset Referenced Bits

Exceptions

If AC1 contains a nonexistent pageframe number or ATU is off, then **ORFB** produces undefined results.

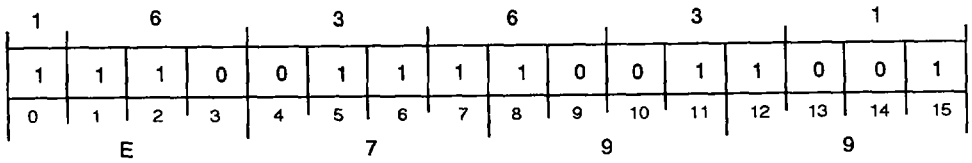
Example

```
ORFB
```

Purge the Address Translator PATU

Privileged Instruction

PATU



Function: Purges address translator
unchanged -> OVR
unchanged -> CRY

Parameters: None

PATU purges the entire address translator of all entries. It should be used only when page table entries are either invalidated or changed.

Arguments

None

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

LPTE	Load Page Table Entry

Exceptions

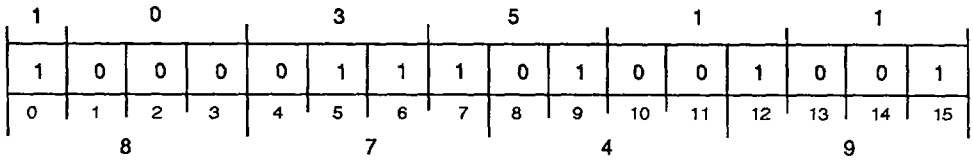
None

Example

```
PATU        ;Purge the Address Translation Unit.
```

Pop Block and Execute PBX

PBX



Function: Disables interrupt system for one instruction execution
AC0(16-bit opcode) -> temporary location
6 double words -> registers, PSR, PC, CRY
(wide return block)
after popped block, PC of **BKPT** -> PC
after execute, PC + length of executed opcode -> PC
execute(temporarily-saved opcode)

Parameters: None

NOTE: Popped PC must reference a **BKPT** instruction

PBX is used in conjunction with **BKPT** to return program control from the breakpoint handler.

PBX disables the interrupt system for one instruction execution. Then it temporarily saves the one-word opcode and performs a modified WPOPB.

Finally, it temporarily replaces the **BKPT** instruction with the temporarily-saved opcode and continues normal program flow.

Arguments

None

Registers, Flags, and Stacks

AC0(16-31) Before execution, contains 16-bit opcode.

If AC0 contains first word of multi-word instruction, processor locates remainder of multi-word instruction beginning at PC(after pop) + 1.

After execution, contents unchanged unless modified by instruction in AC0.

AC1-AC3 Unused

CARRY Before AC0 executed, contains popped value.

After AC0 executed, contents determined by instruction in AC0.

Overflow After AC0 executed, determined by instruction in AC0.

PC During execution, references **BKPT**, effectively substituting 16-bit instruction in AC0 (before pop) for **BKPT** referenced by PC after pop.

After execution, PC + (opcode length of instruction in AC0).

PSR If execution of **PBX**ed instruction is interrupted, processor sets IXCT and pushes opcode of saved instruction onto wide stack. Upon returning from interrupt handler, **BKPT** tests IXCT. If set, **BKPT** resets it to 0. Then it pops saved opcode of interrupted instruction off wide stack and executes it.

Instruction executing in AC0 determines values of other flags.

Stack — Before execution, top double word of wide stack contains value referencing **BKPT** instruction. (If value popped off stack and loaded into PC does not reference **BKPT** instruction, results undefined.)

Related Instructions

BKPT — Breakpoint

Exceptions

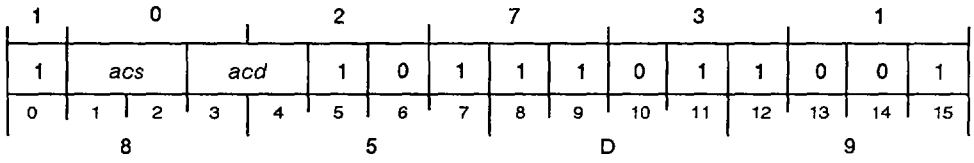
None

Example

```
    .ENT   START          ;Locations 10 - 11 (octal) contains BKPT
    .LOC   10             ;handler address.
BKPTAD:    .BLK 2         ;The linker overwrites this location. It must
                          ;be initialized at runtime.
    .NREL
    ;This program adds all integers from 0 to the integer present in
    ;MAXNUM together and places the result in AC1.
    ;
    ;BKPT will push a wide return block and PBX will pop this same block.
    ;PBX will result in the execution of the opcode present in AC0.
    ;Program flow then continues following the BKPT instruction in the
    ;program.
MAXNUM:    10.
START:     XLEF       3,BKPTH      ;Initialize the BKPT Handler
           XWSTA      3,BKPTAD     ;
           XWLDA      2,MAXNUM     ;Move the maximum number in the series
           WSUB       1,1          ;into AC2 and zero AC1.
ADDLOP:    WSNEI      0,2          ;If the additions are complete then
                                   ;exit,
           WBR EXIT                ;else execute the BKPT instruction.
           BKPT                    ;
           WSBI       1,2          ;Obtain the next lowest number in the
           WBR ADDLOP              ;series to be added and repeat the
                                   ;loop.
EXIT:      WSUB       2,2          ;End of Program - RETURN TO CALLER
           ?RETURN
    ;      BREAKPOINT HANDLER
BKPTH:     XNLDA      0,OPCOD      ;Load the opcode of the insturction
                                   ;WADD 2,1 into AC0. The PBX will
                                   ;cause the execution of this
                                   ;instruction in place of the BKPT.
           WSUB       1,1          ;Load the remaining registers with
           WSUB       2,2          ;garbage to prove the point that the
           WSUB       3,3          ;PBX will do a wide block return.
           PBX                     ;PBX will result in the execution of
                                   ;WADD 2,1, therefore, continuing the
                                   ;series additions.
OPCOD:     WADD       2,1
    ;
    ;The result of this program will be to add the numbers
    ;
    ;10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 55. ( = 67 octal)
    ;
    ;NOTE: You cannot use the system debugger when you have your
    ;own BKPT handler.
    .END   START
```

Program I/O PIO

PIO *acs,acd*



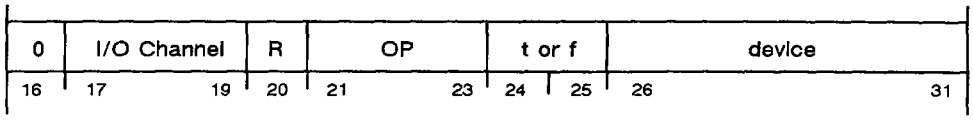
Function: command(*acs*) -> device

Parameters: *acs*(16-31) = command -> unchanged
acd(16-31) = source -> destination

PIO issues a programmed I/O command (in *acs*) to an I/O device on the specified I/O channel. Transferred data is stored in (or loaded into) *acd*.

Arguments

acs(16-31) Before execution, contains I/O command to be sent to I/O channel. I/O command must be formatted as follows:



Bits	Function*
16	0
17-19	I/O channel number
20	Reserved; set to 0
21-23	I/O operation code
24,25	t or f control flags
26-31	Device code

* Refer to the "Device Management" chapter for additional information on I/O functions.

After execution, contents unchanged.

acd(16-31) Contains data dependent upon I/O command (read or write). When *acd* is specified as other than AC0, execution of an **NIO** or an I/O Skip instruction produces results specific to the implementation only.

Registers, Flags, and Stacks

AC0-AC3 Can be individually specified as *acs* or *acd*; otherwise unused.

CARRY Unchanged

Overflow 0

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

DIA, DIB, DIC Transfer data from buffer of I/O device to an accumulator.

DOA, DOB, DOC
Transfer data from an accumulator to the buffer of an I/O device.

Exceptions

None

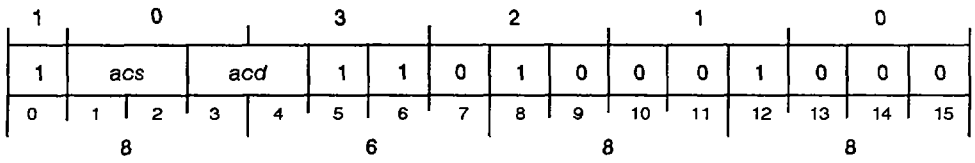
Example

```
NLDAI  30427,0      ;Command to DIAC 27 of IOC 3.
PIO    0,1          ;Equivalent to DIAC 1,27 on IOC 3.
```

Pop Multiple Accumulators POP

ECLIPSE Instruction

POP *acs,acd*



Function: stack -> *acs* to *acd*
-n(1-4) words -> stack
1st stack word -> *acs*
nth stack word -> *acd*
sp-n -> sp

Parameters: None

NOTE: If *acs* is *acd*, 1 word is popped

POP pops words off the narrow stack and loads them into the specified accumulators. The number of words popped is equal to the number of accumulators specified by *acs* through *acd* inclusive. The accumulators are filled in descending order, starting with *acs* and continuing downward numerically through the accumulators to and including *acd*, wrapping around, if necessary, with AC3 following AC0. If the same accumulator is specified for *acs* and *acd*, only one word is popped and it is placed in the specified accumulator.

A check for underflow is made after the entire pop operation is completed.

Arguments

acs(16-31) Starting accumulator of set; receives first word popped from stack.

acd(16-31) Ending accumulator of set; receives last word popped from stack.

Registers, Flags, and Stacks

AC0-AC3 Individually included or excluded by beginning and ending boundaries specified by *acs* and *acd*; data stored in bits 16-31; bits 0-15 undefined. If excluded from set, contents unchanged.

CARRY Unchanged

Overflow 0

PC PC + 1

Stack Narrow stack pointer decremented by number of accumulators popped; frame pointer unchanged.

Related Instructions

PSH Push Multiple Accumulators

Exceptions

None

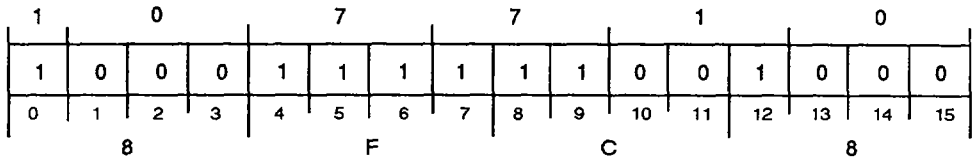
Example

```
PSH    2,0        ;Push bits 16-31 of AC2, AC3, and AC0 onto the
                  ;narrow stack.
...
POP    0,2        ;Pop words off the stack and restore bits 16-31 of
                  ;AC0, AC2, and AC3 to their values at the time of
                  ;the PSH.
```

Pop Block POPB

ECLIPSE Instruction

POPB



Function: stack -> registers
stack -> -5 words (narrow return block)
sp -> sp-5

Parameters: None

POPB returns control from an **XOP0** extended operation or an I/O interrupt handler that does not use the stack change facility of the ECLIPSE C/350 Vector instruction (VCT).

POPB pops five words off the narrow stack and places them in the following locations:

Word Popped	Destination
1	Bit 0 is loaded into carry Bits 1-15 are loaded into the PC
2	AC3
3	AC2
4	AC1
5	AC0

Underflow is checked after the entire pop operation is completed.

Sequential operation continues with the word addressed by the updated value of the program counter.

Arguments

None

Registers, Flags, and Stacks

AC0-AC3	After execution, bits 16-31 contain words popped from stack as specified above with bits 0-15 undefined.
CARRY	Bit 0 of first word popped from stack
Overflow	0
PC	Bits 1-15 of first word popped from stack
Stack	Narrow stack pointer decremented by five words; narrow frame pointer unchanged.

Related Instructions

WPOPB	Wide Pop Block

Exceptions

If a stack underflow occurs, the stack fault handler is executed.

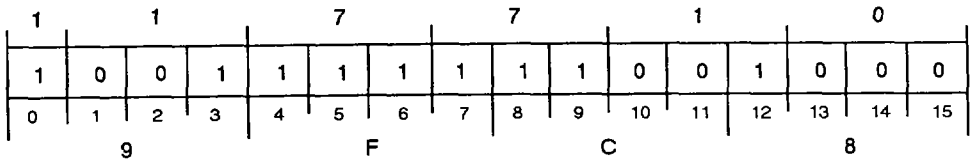
Example

```
POPB      ;Pop 5 words off the stack, setting AC0-3 and the PC.
```

Pop PC and Jump POPJ

ECLIPSE Instruction

POPJ



Function: top stack word -> PC
sp -> sp-1

Parameters: None

POPJ pops the top word off the narrow stack and places it in the program counter. Sequential operation continues with the word addressed by the updated value of the program counter. Underflow is checked after the pop is completed.

The resolved effective address is confined to the first 64 Kbytes of the current segment.

Arguments

None

Registers, Flags, and Stacks

AC0-AC3	Unused
CARRY	Unchanged
Overflow	0
PC	Word popped from stack
Stack	Narrow stack pointer decremented by one; narrow frame pointer unchanged.

Related Instructions

WPOPJ	Wide Pop PC and Jump

Exceptions

If a stack underflow occurs, the stack fault handler is executed.

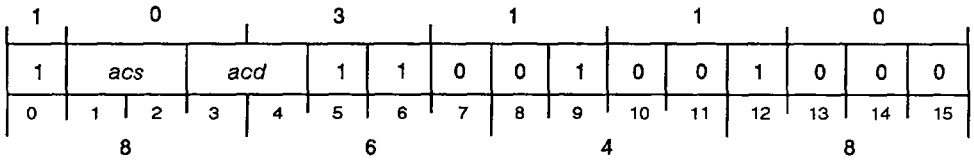
Example

```
    PSHJ   SUBROUT      ;Call a subroutine. Return PC is on the stack.
    ...
SUBROUT:...             ;Subroutine is implemented here.
    ....
    POPJ        ;Pop return address and return to caller.
                ;ACs modified in the subroutine are not
                ;restored.
```

Push Multiple Accumulators PSH

ECLIPSE Instruction

PSH *acs,acd*



Function: *acs* to *acd* -> stack
+(1-4) words -> stack
sp -> sp + (1-4)

Parameters: None

NOTE: If *acs* is *acd*, 1 ac is pushed

PSH pushes words from the specified accumulators onto the narrow stack. The number of words pushed is equal to the number of accumulators specified by *acs* through *acd* inclusive. The accumulators are pushed in ascending order, starting with *acs* and continuing upward numerically through the accumulators to and including *acd*, wrapping around, if necessary, with AC0 following AC3. If the same accumulator is specified for *acs* and *acd*, only one accumulator, the one specified, is pushed.

Stack overflow is checked after the entire push operation is completed.

Arguments

acs(16-31)	Starting accumulator of set; contains first word pushed onto stack.
acd(16-31)	Ending accumulator of set; contains last word pushed onto stack.

Registers, Flags, and Stacks

AC0-AC3	Individually included or excluded by beginning and ending boundaries specified by *acs* and *acd*. After execution, contents unchanged.
CARRY	Unchanged
Overflow	0
PC	PC + 1
Stack	Narrow stack pointer incremented by number of accumulators pushed; narrow frame pointer unchanged.

Related Instructions

POP	Pop Multiple Accumulators

Exceptions

None

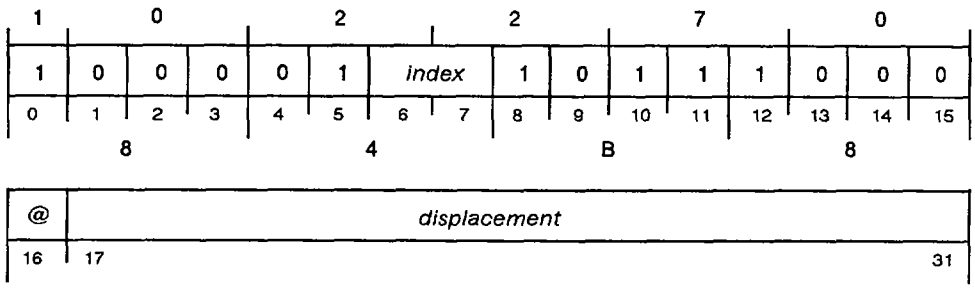
Example

```
PSH    2,0          ;Push bits 16-31 of AC2, AC3, and AC0
...                 ;onto the narrow stack.
POP    0,2          ;Pop words off the stack and restore bits
                    ;16-31 of AC0, AC2, and AC3 to their values
                    ;at the time of the PSH.
```

Push Jump PSHJ

ECLIPSE Instruction

PSHJ [@]*displacement*[,*index*]



Function: PC + 1 -> narrow stack
E -> PC

Parameters: None

PSHJ pushes the address of the next sequential instruction onto the narrow stack and loads the program counter with the specified address. Sequential operation continues with the instruction addressed by the updated value of the program counter. Stack overflow is checked after the push operation finishes.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	Effective address resolved from argument
Stack	Narrow stack pointer incremented by one; narrow frame pointer unchanged.

Related Instructions

XPSHJ, LPSHJ Push a return address and jump to subroutine.

Exceptions

If a stack overflow occurs, the stack fault handler is executed.

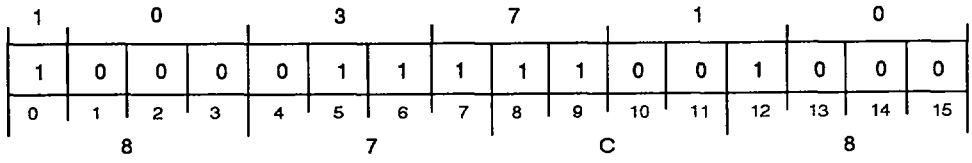
Example

```
    PSHJ   SUBROUT      ;Call a subroutine. Return PC is on the stack.
    ...
SUBROUT:...             ;Subroutine is implemented here.
    ...
    POPJ                ;Pop return address and return to caller.
                        ;ACs modified in the subroutine are not
                        ;restored.
```

Push Return Address PSHR

ECLIPSE Instruction

PSHR



Function: PC + 2 -> narrow stack

Parameters: None

PSHR adds 2 to current program counter and pushes the result onto the narrow stack. Stack overflow is checked after the push operation finishes.

Arguments

None

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1
Stack	Narrow stack pointer incremented by 1; narrow frame pointer unchanged.

Related Instructions

POPJ	Pop PC and Jump

Exceptions

If a stack overflow occurs, the stack fault handler is executed.

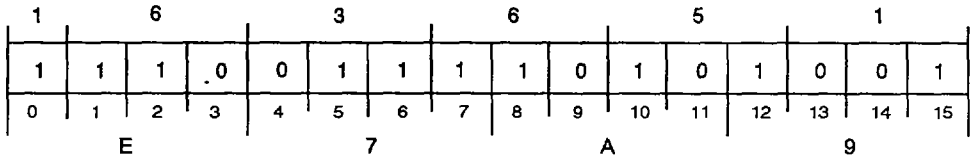
Example

```
    PSHR              ;Push return PC on the stack.
    JMP    SUBROUT    ;Jump to a subroutine.
    ...
SUBROUT:...           ;Subroutine is implemented here.
    ...
    POPJ              ;Pop return address and return to caller.
                      ;ACs modified in the subroutine are not
                      ;restored.
```

Reset Referenced Bits RRFB

Privileged Instruction

RRFB



Function: Operate on groups of 16 referenced bits:
0 -> referenced bits ((AC1), (AC1) + 1,...,(AC1) + 15)
Do until AC0 < 0

Parameters: AC0 = pageframe count -> all 1s
AC1 = pageframe #(13-31) -> (AC1 + (AC0 + 1) * 16)

NOTE: If AC0 => 0 or = largest negative #, then reset pageframes.
If initial AC0 = -value, then final AC0 = AC0 -1.

RRFB resets a specified string of pageframe referenced bits to 0. The string is processed as 16-bit words, each word representing the referenced bits for 16 contiguous pageframes.

Arguments

None

Registers, Flags, and Stacks

AC0	Before execution, specifies number of words minus 1 to be reset in referenced bit string. With each word processed, count decrements by 1. A count of 0 or any negative count except 8000000_{16}) causes one 16-bit word to be reset. After execution, count equals -1. (If initial AC0 value is negative, then final value equals initial AC0 minus one.
AC1(13-27)	Before execution, specifies starting pageframe number for referenced bit string; bits 28-31 set to 0 (thus, each increment of pageframe number is a multiple of 16). With each word processed, contents incremented by 16. After execution, contains original starting pageframe number plus referenced bit word count initially stored in AC0 plus 1, times 16
AC2-AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

ORFB OR Referenced Bits

Exceptions

If AC0 specifies a nonexistent pageframe or if the address translator is not enabled, the result is undefined.

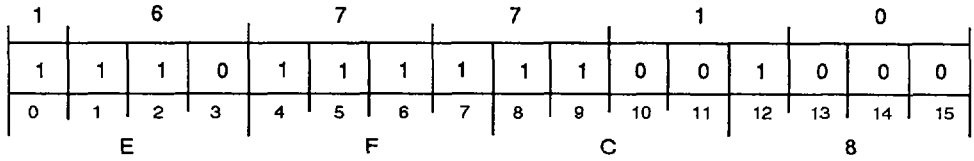
Example

```
    .TITLE    RRFB_EXAMPLE
    .ENT  START
    .NREL
    ;RRFB EXAMPLE
    ;This example resets the referenced bits for 48 pages, starting ;
    ;with pageframe 32. AC0 must be loaded with an origin zero count
    ;of the number of groups of 16 referenced bits to reset, so for
    ;this example:
    ;
    ;     AC0 <- (48/16 - 1) = 2
    ;
    ;AC1 must be loaded with a pageframe number in bits 13-31. Also,
    ;the last four bits must be zero since the referenced bits are
    ;always reset in groups of 16, so the starting pageframe number
    ;must be a multiple of 16. In this case:
    ;
    ;AC1 <- 32 decimal = 20 hex = 100000 binary
    ;
    ;Note that to execute RRFB, you must in ring 0, since it is
    ;a privileged instruction.
START:    NLDAI       2,0
          NLDAI       32.,1
          RRFB
          ?RETURN
          .END        START
```

Restore RSTR

ECLIPSE Instruction

RSTR



Function: stack -> locations
stack -> -9 words (narrow return block)
6th word -> stack fault address
7th word -> sl
8th word -> fp
9th word -> sp

Parameters: None

RSTR returns control from an interrupt by popping nine words off the narrow stack and placing them into the following locations:

Word Popped	Destination
1	Bit 0 into carry; bits 1-15 into the PC
2	AC3
3	AC2
4	AC1
5	AC0
6	Stack fault address
7	Stack limit
8	Frame pointer
9	Stack pointer

Sequential operation continues with the instruction addressed by the updated value of the program counter. The return environment must have been previously saved on the stack in proper sequence. Stack underflow is not checked.

Arguments

None

Registers, Flags, and Stacks

AC0–AC3	After execution, bits 16–31 contain words popped from stack as specified above with bits 0–15 undefined.
CARRY	Bit 0 of first word popped from stack
Overflow	0
PC	Bits 1–15 of first word popped from stack
PSR	Unchanged
Stack	Narrow stack parameters updated as specified by popped values.

Related Instructions

WRSTR Wide Restore

Exceptions

None

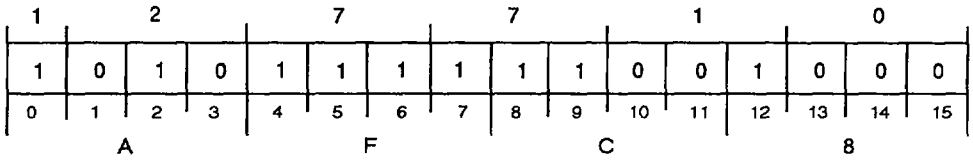
Example

```
RSTR    ;Pop 9 words off the narrow stack, restoring
        ;the PC, ACs, and stack registers to their
        ;values at the time an interrupt occurred.
```

Return RTN

ECLIPSE Instruction

RTN



Function: stack -> registers
stack -> -5 words (narrow return block)
fp-5 -> sp
AC3(popped) -> fp

Parameters: None

RTN returns control from a subroutine by popping 5 words off the narrow stack and placing them into the following locations:

Word Popped	Destination
1	Bit 0 into carry; bits 1-15 into the PC
2	AC3
3	AC2
4	AC1
5	AC0

Sequential operation continues with the instruction addressed by the updated value of the program counter. The return environment must have been previously placed on the stack using a **SAVE** instruction or the equivalent.

Arguments

None

Registers, Flags, and Stacks

AC0-AC3	After execution, bits 16-31 contain words popped from stack as specified above with bits 0-15 undefined.
CARRY	Bit 0 of first word popped from stack
Overflow	0
PC	Bits 1-15 of first word popped from stack
PSR	Unchanged
Stack	Narrow stack pointer updated to decremented value of narrow frame pointer; narrow frame pointer updated to value from popped AC3

Related Instructions

WRTN	Wide Return

Exceptions

None

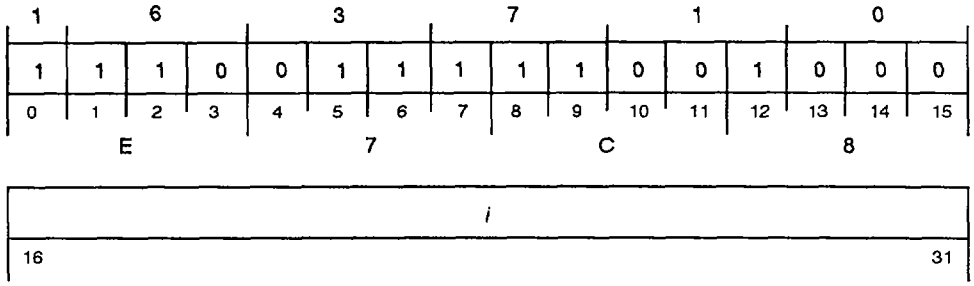
Example

```
...
SAVZ    ;    Save all AC's on the stack
...
RTN          ;Return to caller
.END START
```

Save SAVE

ECLIPSE Instruction

SAVE *i*



Function: *5 + i* words -> narrow stack
AC0 -> 1st word
AC1 -> 2nd word
AC2 -> 3rd word
fp(before instruction execution) -> 4th word
CRY -> 5th word(bit 0)
AC3 -> 5th word(bits 1-15)
sp(before push) + 5 + *i* -> sp
sp(before push) + 5 -> fp
sp(before push) + 5 -> AC3

Parameters: None

SAVE pushes a 5-word return block onto the narrow stack and then allocates an additional frame area in the stack (number of words specified by *i*) to be written to by the current procedure. After execution of **SAVE**, the frame space designated by *i* can be used to store arguments/data to be carried forward by the new subroutine. (Since the stack pointer will already encompass this space, use store instructions for loading.)

The return block is formatted as follows:

Word Pushed	Contents
1	AC0
2	AC1
3	AC2
4	Frame pointer before the SAVE
5	Bit 0 = CARRY; Bits 1-15 = Initial value of AC3 (return value for PC)

The effective address generated by **SAVE** is confined to the first 64 Kbytes of the current segment.

Arguments

i Unsigned 16-bit integer specifying size of frame area (in words) for storing data on stack after return block frame.

Registers, Flags, and Stacks

AC0(16-31) First word pushed onto stack.

AC1(16-31) Second word pushed onto stack.

AC2(16-31) Third word pushed onto stack.

AC3(17–31)	Before execution, contains return value for PC. After execution, contains updated frame pointer.
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 2
PSR	Unchanged
Stack	Narrow frame pointer = original stack pointer + 5; Narrow stack pointer = original stack pointer + 5 + *i*.

Related Instructions

SAVZ	Save Without Arguments
RTN	Use the Return instruction to pass control via a return block.

Exceptions

If the stack overflows, a stack fault is generated.

Example

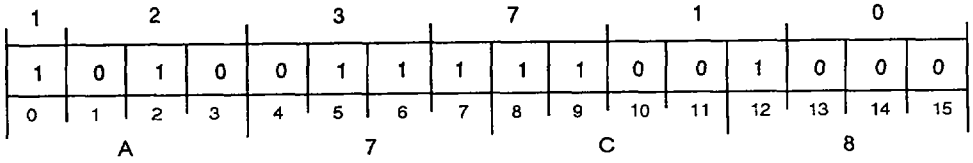
```
    .TITLE SAVE
    .ENT   START, RESULT, XTEL, STKER, RESOK, ARADD, ARAY0, ARAY1
    .ENABLE UWORD
    .LOC   14
    STKER                ;Stack Fault Handler
    .LOC   40
SP:       400            ;Stack Pointer
FP:       400            ;Frame Pointer
SL:       436            ;Stack Limit
          .LOC           401
          .BLK 50        ;allocate 50 octal words for the stack
          .NREL
    ;
    ;
    ;     Main program calls a subroutine to add together 2 arrays
    ;
START:    ELDA           0,ALEN       ;Get length of ARRAYS into AC0.
          ELDA           1, A1
          ELDA           2, A2
          EJSR           ARADD
          ELDA           3,A1         ;Make sure AC1 hasn't changed!
          SUB#           3,1,SZR
          JMP STKER                   ;IF not zero then AC1 changed
          ELDA           3,A2
          SUB#           3,2,SNR
          JMP RESOK                   ;If not zero then AC2 changed
                                      ;AC Changed -- Error occurred
STKER:    LDA 2,ERFLG    ;Come here also for stack fault error
          ?RETURN
          ?RFCF+?RFER
```

```
RESOK:  SUB 2,2
        ?RETURN
ALEN:   4
A1:     25.             ;Random numbers
A2:     99.
ARAY0:  1
        3
        5
        7
ARAY1:  1
        2
        3
        4
RESULT: .BLK            10.
   ;
   ;    This subroutine Adds together two arrays of length from AC0
   ;    the result goes into RESULT
   ;
   ;    RESULT <- ARAY0 + ARAY1
LCNT:   0
ARADD:  SAVE            ;Save All ACs on stack. Note return address was in
                        ;AC3 from the JSR. SAVE pushes AC3 onto stack, and
                        ;puts the Stack Pointer into AC3.
        LDA 1,-4,3      ;Get Length of Arrays from AC0 on
        STA 1,LCNT      ;stack (SP - 4) put copy of LENGTH in LCNT
        SUB 2,2         ;Set AC2 to 0
NXTEL:  ELDA            0,ARAY0,2    ;Get ARAY0[AC2]
        ELDA            1,ARAY1,2    ;Get ARAY1[AC2]
        ADD 0,1         ;Add them
        ESTA            1,RESULT,2   ;Store: RESULT[AC2] <- ARAY0[AC2] +
   ARAY1[AC2]
        INC 2,2         ;move to next element
        DSZ LCNT        ;If LCNT = 0 then ALL DONE.
        JMP NXTEL
        RTN             ;Return to caller
        .END START
```

Save Without Arguments SAVZ

ECLIPSE Instruction

SAVZ



Function:
5 words -> narrow stack
AC0 -> 1st word
AC1 -> 2nd word
AC2 -> 3rd word
fp(before instruction execution) -> 4th word
CRY -> 5th word(bit 0)
AC3 -> 5th word(bits 1-15)
sp + 5 -> sp
sp(before push) + 5 -> fp
sp(before push) + 5 -> AC3

Parameters: None

SAVZ pushes a 5-word return block onto the narrow stack. The return block is formatted as follows:

Word Pushed	Contents
1	AC0
2	AC1
3	AC2
4	Frame pointer before the SAVE
5	Bit 0 = CARRY; Bits 1-15 = initial value of AC3 (return value for PC)

The effective address generated is confined to the first 64 Kbytes of the current segment.

Arguments

None

Registers, Flags, and Stacks

AC0(16-31)	First word pushed onto stack.
AC1(16-31)	Second word pushed onto stack.
AC2(16-31)	Third word pushed onto stack.
AC3(17-31)	Before execution, contains return value for PC. After execution, contains updated frame pointer.
CARRY	Unaffected
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Narrow frame pointer and narrow stack pointer = original stack pointer + 5.

Related Instructions

SAVE Save

RTN Use the Return instruction to pass control via a return block.

Exceptions

If the stack overflows, a stack fault is generated.

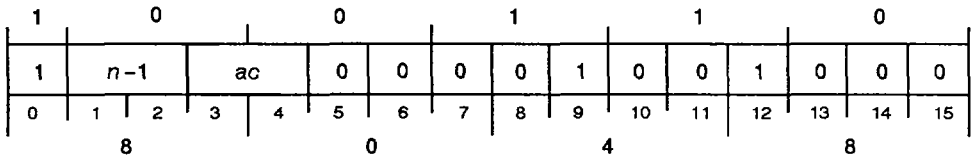
Example

```
    .TITLE SAVZ
    .ENT  START, RESULT, NXTEL, STKER, RESOK, ARADD, ARAY0, ARAY1
    .ENABLE UWORD
    .LOC  14
    STKER     ;Stack Fault Handler
    .LOC  40
SP:       400          ;Stack Pointer
FP:       400          ;Frame Pointer
SL:       436          ;Stack Limit
          .LOC         401
          .BLK 50      ;allocate 50 octal words for the stack
          .NREL
    ;
    ;
    ;     Main program calls a subroutine to add together 2 arrays
    ;
START:    ELDA         0,ALEN       ;Get length of ARRAYS into AC0.
          ELDA         1, A1
          ELDA         2, A2
          EJSR         ARADD
          ELDA         3,A1         ;Make sure AC1 hasn't changed!
          SUB#         3,1,SZR
          JMP STKER    ;IF not zero then AC1 changed
          ELDA         3,A2
          SUB#         3,2,SNR
          JMP RESOK                 ;If not zero then AC2 changed
                                    ;AC Changed -- Error occurred
STKER:    LDA 2,ERFLG               ;Come here also for stack fault error
          ?RETURN
ERFLG:    ?RFCF+?RFER
RESOK:    SUB 2,2
          ?RETURN
ALEN:     4
A1:       25.                       ;Random numbers
A2:       99.
ARAY0:    1
          3
          5
          7
ARAY1:    1
          2
          3
          4
```

```
RESULT:   .BLK        10.
   ;This subroutine Adds together two arrays of length from AC0
   ;the result goes into RESULT
   ;
   ;RESULT <- ARAY0 + ARAY1
LCNT:     0
ARADD:    SAVZ                        ;Save All ACs on stack. Note return
                                      ;address was in AC3 from the JSR. SAVz
                                      ;pushes AC3 onto stack, and puts the
                                      ;Stack Pointer into AC3.
          LDA 1,-4,3                  ;Get Length of Arrays from AC0 on
                                      ;stack
          STA 1,LCNT   ;(SP - 4)      ;put copy of LENGTH in LCNT
          SUB 2,2      ;Set AC2 to 0
NXTEL:    ELDA         0,ARAY0,2      ;Get ARAY0[AC2]
          ELDA         1,ARAY1,2      ;Get ARAY1[AC2]
          ADD 0,1      ;Add them
          ESTA         1,RESULT,2     ;Store: RESULT[AC2] <- ARAY0[AC2] +
   ARAY1[AC2]
          INC 2,2                     ;move to next element
          DSZ LCNT                    ;If LCNT = 0 then ALL DONE.
          JMP NXTEL
          RTN                         ;Return to caller
   .END START
```

Subtract Immediate SBI

SBI *n,ac*



Function: *ac* - *n* -> *ac*
ALU CRY -> CRY

Parameters: None

SBI subtracts an integer in the range 1 to 4 from the unsigned 16-bit integer contained in the specified accumulator, placing the result in the accumulator.

Arguments

n	Integer in range 1-4. Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field. Thus, programmer should code exact value to be subtracted.
ac(16-31)	Before execution, contains unsigned 16-bit integer. After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Set with value of ALU CARRY
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

NSBI	Narrow Subtract Immediate
WSBI	Wide Subtract Immediate

Exceptions

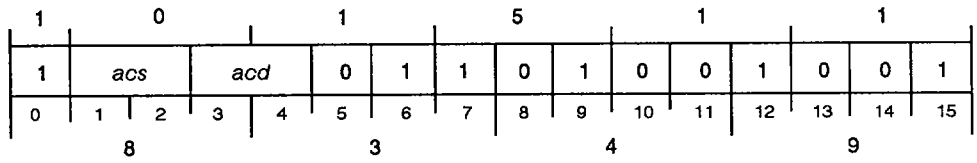
None

Example

```
LDA   0,FIVE        ;Get a constant 5.
SBI   4,0           ;Subtract 4 from AC0. AC0[16-31] is now 1.
```

Sign Extend SEX

SEX *acs,acd*



Function: *acs*[16 bit #] -> *acd*[32 bit #] (sign-extended)

Parameters: None

SEX sign-extends the signed 16-bit integer contained in *acs* to 32 bits and loads the result into *acd*.

Arguments

acs(16-31)	Before execution, contains signed 16-bit integer.
	After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd	After execution, contains *acs* sign-extended to 32 bits.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

ZEX	Zero Extend

Exceptions

None

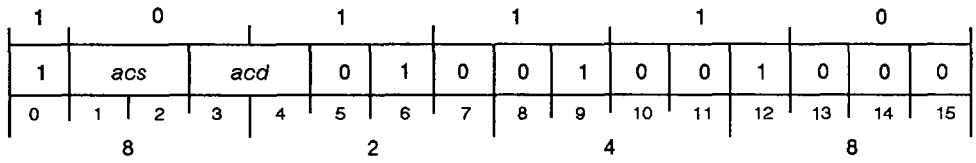
Example

```
ADC   3,3          ;Set AC3[16-31] to ones. AC3[0-15] undefined.
SEX   3,1          ;AC3 unchanged. AC1[0-31] is now all ones.
```

Skip if ACS Greater than or Equal to ACD SGE

ECLIPSE Instruction

SGE *acs,acd*
(*acs* < *acd*)
(*acs* >= *acd*)



Function: If *acs* > = *acd* then skip

Parameters: None

NOTE: Compares signed numbers

SGE compares two signed 16-bit integers in two accumulators and skips if the first is greater than or equal to the second.

Arguments

acs(16-31)	Before execution, contains signed 16-bit integer.
	After execution, contents unchanged.
acd(16-31)	Before execution, contains signed 16-bit integer.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1 (*acs* < *acd*) PC + 2 (*acs* >= *acd*)
PSR	Unchanged
Stack	Unchanged

Related Instructions

SGT	Skip if ACS Greater Than ACD
SUB, ADC	Use these instructions to compare unsigned integers.

Exceptions

None

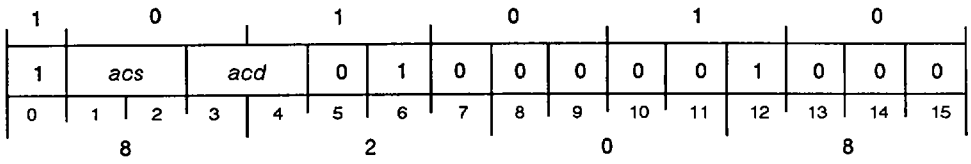
Example

```
LDA   2,FIRST       ;Get first value.
LDA   1,SECOND      ;Get second value.
SGE   1,2           ;Skip if second value is >= first.
JMP   FIRST_GREATER;First value is greater.
. . .               ;Second value is greater or equal.
```

Skip if ACS Greater than ACD — SGT

ECLIPSE Instruction

SGT *acs,acd*
(*acs* <= *acd*)
(*acs* > *acd*)



Function: If *acs* > *acd* then skip

Parameters: None

NOTE: Compares signed numbers

SGT compares two signed 16-bit integers in two accumulators and skips if the first is greater than the second.

Arguments

acs(16-31)	Before execution, contains signed 16-bit integer.
	After execution, contents unchanged.
acd(16-31)	Before execution, contains signed 16-bit integer.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1 (*acs* <= *acd*) PC + 2 (*acs* > *acd*)
PSR	Unchanged
Stack	Unchanged

Related Instructions

SGE	Skip if ACS Greater Than or Equal to ACD
SUB, ADC	Use these instructions to compare unsigned integers.

Exceptions

None

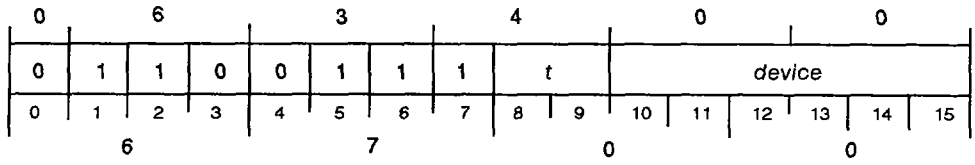
Example

```
LDA   2,FIRST        ;Get first value.
LDA   1,SECOND       ;Get second value.
SGT   1,2            ;Skip if second value is > first.
JMP   FIRST_GREATER;First value is greater or equal.
. . .                ;Second value is greater.
```

I/O Skip SKPt

ECLIPSE Instruction

SKP*t* *device*
(test result not true return)
(test result true return)



Function: If *t* = true then skip
BUSY, DONE flags -> unchanged

Parameters: None

SKP*t* tests the BUSY and DONE status flags of an I/O *device*. If the test condition specified by *t* is true, the instruction skips the next sequential instruction.

Arguments

t Specifies test function as follows:

t Symbol	Bit Value	Test
BN	0 0	BUSY = 1
BZ	0 1	BUSY = 0
DN	1 0	DONE = 1
DZ	1 1	DONE = 0

device Specify either mnemonic or device code for desired I/O device.

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 1 (test result not true) PC + 2 (test result true)
PSR	Unchanged
Stack	Unchanged

Related Instructions

DIA, DIB, DIC Transfer data from buffer of I/O device to an accumulator.

DOA, DOB, DOC Transfer data from an accumulator to the buffer of an I/O device.

Exceptions

None

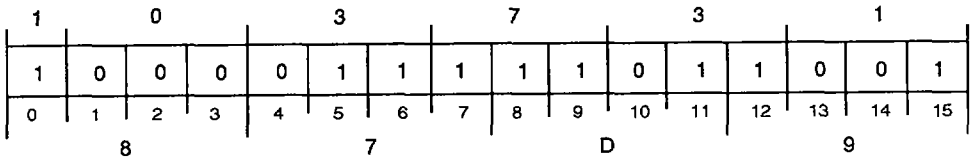
Example

```
    DOAS   0,11        ;Send a character to the op console.
LOOP:      SKPDN 11    ;Has the character finished being sent?
           WBR LOOP    ;No. Test again.
           NIOC 11     ;Yes. Clear the Done flag.
```

Store Modified and Referenced Bits SMRF

Privileged Instruction

SMRF



Function: New values -> modified & referenced bits

Parameters: AC0 = Mod(bit 30), Ref(bit 31) [new values] -> unchanged
AC1 = pageframe # -> unchanged

SMRF stores new specified values into the modified and referenced bits of the specified pageframe.

Arguments

None

Registers, Flags, and Stacks

AC0(30–31)	Before execution bits 30 and 31 specify new values to be stored in respective modified and referenced bit locations. After execution, contents unchanged.
AC1(13–31)	Before execution specifies pageframe number for affected bits. After execution contents unchanged.
AC2–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

LMRF	Load Modified and Referenced Bits

Exceptions

If the address translator is not enabled or a nonexistent pageframe is specified, the results are undefined.

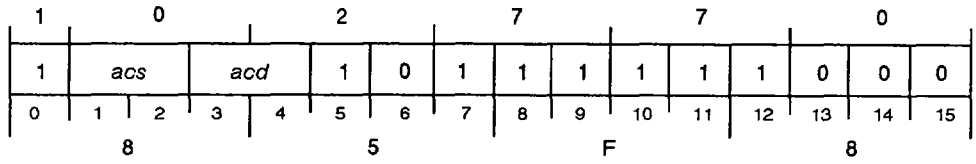
Example

```
WSUB  0,0 ;Modified and referenced both zero.
XWLDA 1,FRAME      ;Get the page frame number.
SMRF               ;Clear the modified and referenced bits.
```

Skip on Nonzero Bit SNB

ECLIPSE Instruction

SNB *acs,acd*
(test bit is 0 return)
(test bit is 1 return)



Function: If (E)bit = 1 then skip

Parameters: *acs* = base word pointer -> unchanged
acd = word offset & bit identifier -> unchanged

NOTE: If *acs* is *acd*, base word pointer = 0 (of the current segment)

SNB tests the specified bit for 1 and skips the next sequential word if test result is true. The effective address generated by the instruction is confined to the first 64 Kbytes of the current segment. If *acs* and *acd* are the same accumulator, then high-order word of address defaults to 0.

Arguments

acs(16-31)	Contains high-order word of 32-bit address.
acd(16-31)	Contains low-order word of 32-bit address (includes bit pointer).

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* or *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1 (test bit is 0) PC + 2 (test bit is 1)
PSR	Unchanged
Stack	Unchanged

Related Instructions

SZB	Skip on Zero Bit
SZBO	Skip on Zero Bit and Set to One

Exceptions

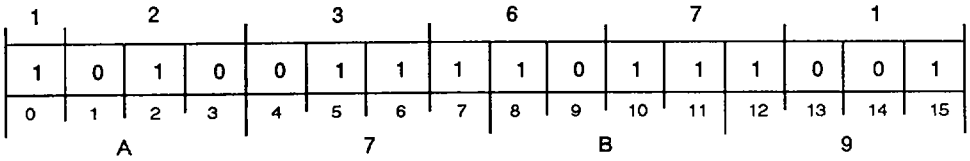
None

Example

```
    ELEF   0,FLAGS      ;Get word address of flags word.
    SUB    1,1          ;Get a zero in AC1.
    ADI    3,1          ;Get a 3 in AC1.
    SNB    0,1          ;Is bit 3 of the flags word set?
    JMP    NOT_SET      ;No.
    . . .               ;Yes.
    ...
FLAGS:    .WORD 0       ;Flags word.
```

Skip on OVR Reset SNOVR

SNOVR
(OVR = 1 return)
(OVR = 0 return)



Function: If OVR = 0 then skip

Parameters: None

SNOVR tests the value of the Processor Status Register overflow flag (OVR). If the flag has the value 0, the next sequential word is skipped. If the flag has the value 1, the next sequential word is executed.

Arguments

None

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1 (OVR = 1) PC + 2 (OVR = 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

LPSR	Load Processor Status Register
SPSR	Store Processor Status Register

Exceptions

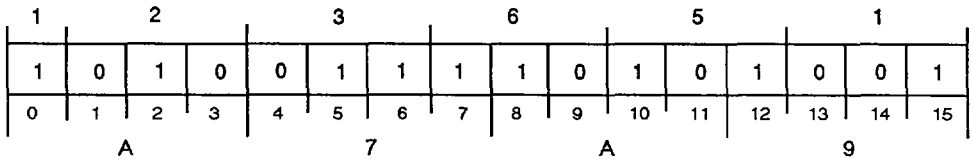
None

Example

```
SNOVR              ;Is the OVR flag in the PSR set?
WBR   OVR_SET      ;Yes, it is set.
. . .              ;No. OVR is not set.
```

Store Processor Status Register SPSR

SPSR



Function: AC0 -> PSR
unchanged -> CRY

Parameters: None

SPSR stores bits 0–15 of AC0 into the Processor Status Register.

Arguments

None

Registers, Flags, and Stacks

AC0(0–15)	Before execution, contains bits to be loaded into PSR. AC0(0) corresponds to PSR(0), AC(1) to PSR(1), etc. Undefined PSR bits are ignored. After execution, contents unchanged.
AC1–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	After execution, contains values from AC0(0–15).
Stack	Unchanged

Related Instructions

LPSR	Load Processor Status Register
Load with immediate	Use these instructions to place a value into AC0.

Exceptions

None

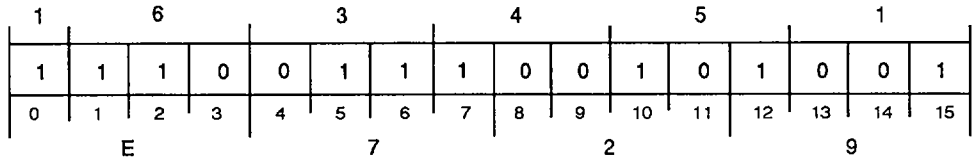
Example

```
XWLDA 0,PSR        ;Get saved PSR from memory.
SPSR               ;Store into the PSR register.
```

Store Page Table Entry SPTE

Privileged Instruction

SPTE



Function: AC0[data] -> @(AC2)
updates address translator (machine specific)

Parameters: AC0 = new PTE data -> unchanged
AC1 = logical address for PTE -> unchanged
AC2 = physical address of PTE slot -> unchanged

NOTE: You should make sure that the specified logical address in AC1 will produce the physical address in AC2 of the last PTE accessed.

SPTE stores Page Table Entry (PTE) data contained in AC0 to the physical address specified in AC2. AC1 contains the logical address that uses the new PTE data. The instruction updates the hardware address translation mechanism for consistency (machine-dependent).

Arguments

None

Registers, Flags, and Stacks

AC0	Before execution, contains new Page Table Entry data.
	After execution, contents unchanged.
AC1	Before execution, contains logical address to use new PTE data. Specified logical address in AC1 must produce physical address in AC2 of last PTE accessed.
	After execution, contents unchanged.
AC2	Before execution, contains physical address of PTE slot to update.
	After execution, contents unchanged.
AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load effective address
Use these instructions to load a logical address into AC1.

LPTE Load Page Table Entry

LPHY Use this instruction to load a physical address into AC2 and the last resident PTE into AC0.

Exceptions

If the ATU is off, then no operation occurs.

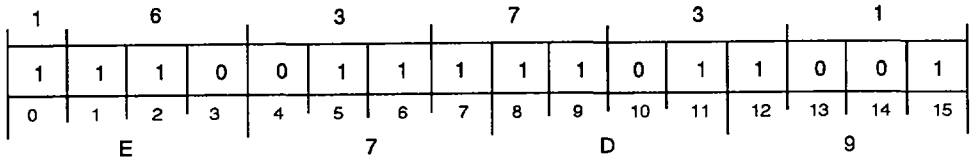
Example

```
XWLDA 0,NEW_PTE                ;Get the new PTE.
XWLDA 1,LOGICAL_ADDRESS        ;Get the logical address.
XWLDA 2,PTE_PHYSICAL_ADDR      ;Get the PTE physical address.
SPTE                           ;Store the PTE and flush caches as
                               ;necessary.
```

Store State Pointer SSPT

Privileged Instruction

SSPT



Function: If AC0 ≠ -1,
then (AC0) = state pointer
and state pages assigned --> AC1
If AC0 = -1, then state area size --> AC1

Parameters: AC0 = definition/status value --> unchanged
AC1 = ? --> state area size

SSPT, depending on the contents of AC0, either defines or requests the status of the state area in memory. The operating system must execute **SSPT** at system initialization time, before the address translator is enabled.

Arguments

None

Registers, Flags, and Stacks

AC0	Before execution, contains value indicating operation of instruction. If -1, required state area size is returned to AC1; **SSPT** does not allocate the state area. In this case, execute **SSPT** twice. If other than -1, bits 20-31 contain physical pageframe number of state area base to be moved to state pointer. After execution, contents unchanged.
AC1	Before execution, unused. After execution, contents dependent upon initial value of AC0: If AC0 contains other than -1, AC1 contains number of state pages assigned. If AC0 contains -1, AC1 contains number of consecutive physical pages that operating system should reserve in memory.
AC2-AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load with immediate
Use these instructions to place a value into AC0.

WADC 0,0 Use this Wide Add Complement instruction to create a -1 in AC0.

Exceptions

If AC0 initially contains a -1, **SSPT** returns the number of memory pages required by the processor in AC1 and does not allocate the state area. In this case, execute **SSPT** twice.

If it becomes necessary to move the state area (i.e., as a result of a hard memory failure within the state area), the operating system should stop operations that may change the contents of the state area, such as **CIO** or **WLMP** operations (this procedure is machine-dependent). It may then perform the move and reload the state pointer by re-executing **SSPT**.

If the processor does not implement a state area, **SSPT** is a no-op instruction.

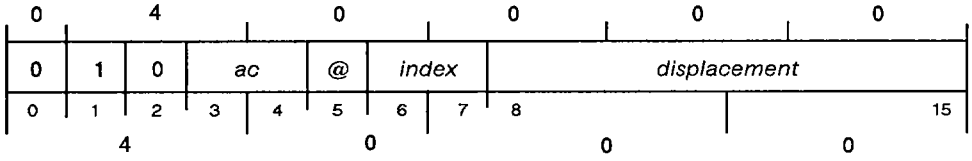
Example

```
WSUB   1,1                    ;In case SSPT is a no-op.
XWLDA  0,PAGENUM              ;Get the first page frame to be used.
SSPT                          ;Do it.
XWSTA  1,PAGES_TAKEN          ;Store the number of pages needed.
```

Store Accumulator STA

ECLIPSE Instruction

STA *ac*,[@]*displacement*[,*index*]



Function: *ac* -> (E)

Parameters: None

STA stores the contents of specified accumulator into the specified memory location.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

ac(16–31) Before execution, contains data word to be stored in memory.

After execution, contents unchanged.

Registers, Flags, and Stacks

AC0–AC3 Can be individually specified as *ac*; otherwise unused.

CARRY Unchanged

Overflow 0

PC PC + 1

Stack Unchanged

Related Instructions

ESTA, LNSTA, LWSTA, XNSTA, XWSTA
Store the contents of an accumulator into memory.

Exceptions

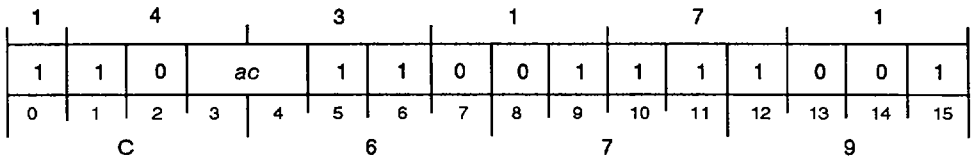
None

Example

```
    STA   0,@ACOUNT    ;Store AC0[16-31] into memory.
    .                  ;Example shows use of indirection.
    ...
ACOUNT: .WORD COUNTER  ;Address of counter.
    ...
    COUNTER:.WORD 0    ;Counter.
```

Store Accumulator in WFP STAFP

STAFP *ac*



Function: *ac* -> wfp

Parameters: None

STAFP stores a copy of the contents of the specified accumulator into the wide frame pointer.

Arguments

ac	Before execution, contains 32-bit value.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Wide frame pointer contains value from *ac*.

Related Instructions

STASB, STASL, STASP
Store the contents of an accumulator into wide stack parameters.

Exceptions

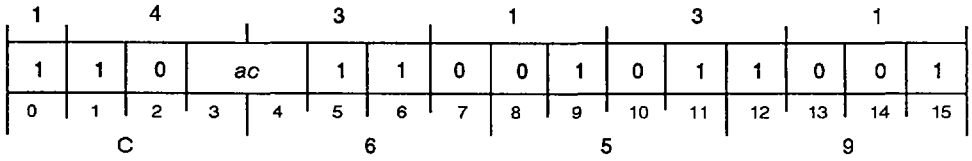
None

Example

```
    LLEF   0,NEW_STACK-2          ;Get starting address of new stack.
    STASB  0                      ;Set the wide stack base.
    STAFP  0                      ;Set the wide frame pointer.
    STASP  0                      ;Set the wide stack pointer.
    LLEF   0,END_STACK            ;Get address of end of new stack.
    STASL  0                      ;Set the wide stack limit.
    ...
NEW_STACK: .BLK 500.              ;500 words for the stack.
END_STACK: .BLK 50.               ;50 words for stack overflow area.
```

Store Accumulator in WSB STASB

STASB *ac*



Function: *ac* -> wsb

Parameters: None

STASB stores the contents of an accumulator into the wide stack base and updates locations 26_8–27_8 in page zero of reserved memory.

Arguments

ac	Before execution, contains 32-bit value.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Wide stack base contains new value from *ac*.

Related Instructions

STAFP, STASL, STASP
Store the contents of an accumulator into wide stack parameters.

Exceptions

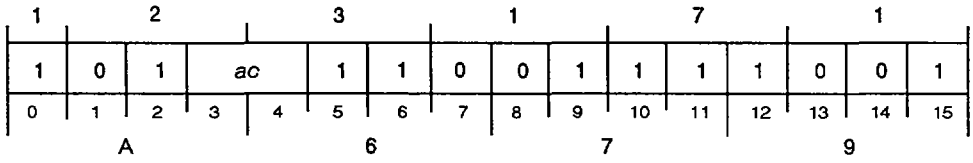
None

Example

```
    LLEF   0,NEW_STACK-2          ;Get starting address of new stack.
    STASB  0                      ;Set the wide stack base.
    STAFP  0                      ;Set the wide frame pointer.
    STASP  0                      ;Set the wide stack pointer.
    LLEF   0,END_STACK            ;Get address of end of new stack.
    STASL  0                      ;Set the wide stack limit.
    ...
NEW_STACK: .BLK 500.              ;500 words for the stack.
END_STACK: .BLK 50.               ;50 words for stack overflow area.
```

Store Accumulator in WSL STASL

STASL *ac*



Function: *ac* -> wsl

Parameters: None

STASL stores the contents of an accumulator into the wide stack limit and updates locations 24_8–25_8 in page zero of reserved memory.

Arguments

ac	Before execution, contains 32-bit value.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Wide stack limit contains value from *ac*.

Related Instructions

STAFP, STASB, STASP
Store the contents of an accumulator into wide stack parameters.

Exceptions

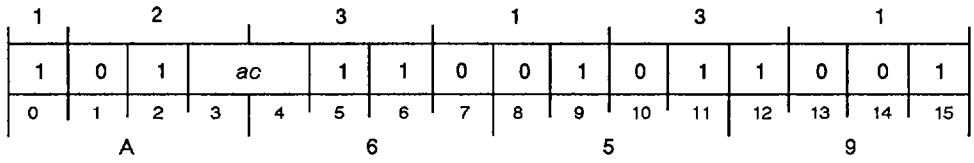
None

Example

```
    LLEF   0,NEW_STACK-2          ;Get starting address of new stack.
    STASB  0                      ;Set the wide stack base.
    STAFP  0                      ;Set the wide frame pointer.
    STASP  0                      ;Set the wide stack pointer.
    LLEF   0,END_STACK            ;Get address of end of new stack.
    STASL  0                      ;Set the wide stack limit.
    ...
NEW_STACK: .BLK 500.              ;500 words for the stack.
END_STACK: .BLK 50.               ;50 words for stack overflow area.
```

Store Accumulator in WSP STASP

STASP *ac*



Function: *ac* -> wsp

Parameters: None

STASP stores the contents of an accumulator into the wide stack pointer.

Arguments

ac	Before execution, contains 32-bit value.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Wide stack pointer contains value from *ac*.

Related Instructions

STAFP, STASB, STASL
Store the contents of an accumulator into wide stack parameters.

Exceptions

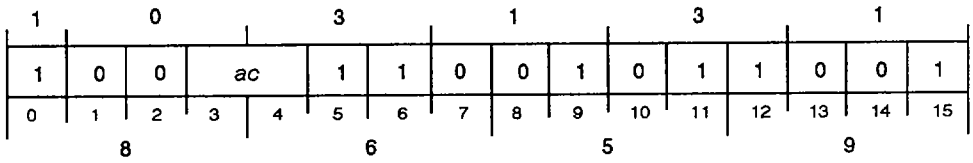
None

Example

```
    LLEF   0,NEW_STACK-2          ;Get starting address of new stack.
    STASB  0                      ;Set the wide stack base.
    STAFP  0                      ;Set the wide frame pointer.
    STASP  0                      ;Set the wide stack pointer.
    LLEF   0,END_STACK            ;Get address of end of new stack.
    STASL  0                      ;Set the wide stack limit.
    ...
NEW_STACK: .BLK 500.              ;500 words for the stack.
END_STACK: .BLK 50.               ;50 words for stack overflow area.
```

Store Accumulator into Stack Pointer Contents STATS

STATS *ac*



Function: *ac* -> (wsp)

Parameters: None

STATS uses the contents of the wide stack pointer as the address of a double word. It stores a copy of the contents of the specified accumulator at the address contained in WSP.

Arguments

ac	Before execution, contains 32-bit value.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

LDATS	Load Accumulator with Double Word

Exceptions

None

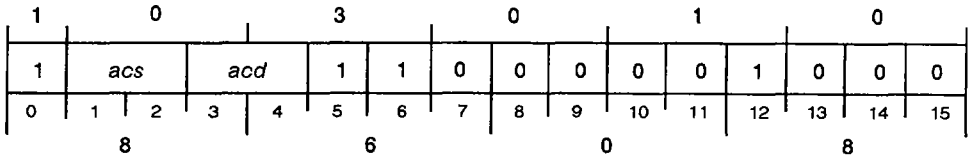
Example

```
WPSH  1,1          ;Save AC1 on the stack.
...
STATS 0            ;Change the pushed value to what is
                   ;currently in AC0.
```

Store Byte STB

ECLIPSE Instruction

STB *acs,acd*



Function: *acd*[right byte] -> (E)byte

Parameters: *acs* = byte pointer -> unch

STB stores a byte from the specified accumulator into the specified memory byte location.

Arguments

acs(16–31) Before execution, contains byte address. Effective address generated by instruction confined to first 64 Kbytes of current segment.

After execution, contents unchanged.

acd(24–31) Before execution, contains byte to be placed in memory.

After execution, contents are unchanged.

Registers, Flags, and Stacks

AC0–AC3 Can be individually specified as *acs* or *acd*; otherwise unused.

CARRY Unchanged

Overflow 0

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

WSTB Wide Store Byte

Exceptions

None

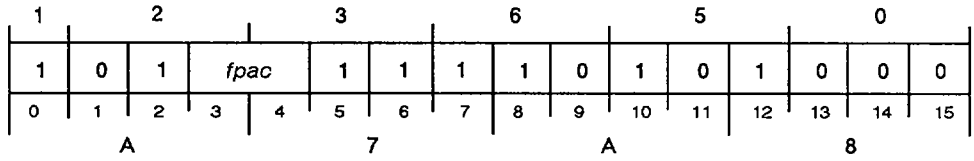
Example

```
    ELEF   2,(BYTE_PAIR*2)+1        ;Get byte address of low order byte.
    STB    2,0                      ;Store AC0[24-31] into the low order
    ...                             ;byte of the word.
    ...
BYTE_PAIR: .WORD 0                  ;Location containing a pair of bytes.
```

Store Integer STI

ECLIPSE Instruction

STI *fpac*



Function: *fpac*[fp#] -> @(AC3)[#]
AC3 -> AC2
0 -> CRY

Parameters: AC1 = data-type indicator -> unchanged
AC2 = x -> AC3
AC3 = byte pointer -> last byte pointer + 1
fpac = fp# -> unchanged

STI converts the contents of a floating-point accumulator to an integer of the specified data type and length, and stores the result as a string in memory.

For data types 0 through 6, the digits are stored right-aligned with the least significant digit stored at the highest address location of the string. If the number of significant digits is insufficient to fill the string, the remaining high-order digits are set to 0 (for data type 6, the sign bit is extended leftward to fill the string).

For data type 7, the digits are left-aligned and remaining low-order bytes are set to 0.

Arguments

fpac Before execution, contains 64-bit floating-point number to be converted.

After execution, contents unchanged.

Registers, Flags, and Stacks

AC0 Unused

AC1(16-31) Before execution, defines data type and string size of converted data. **STI** does not use the scale factor in the data type indicator.

After execution, contents unchanged.

AC2(16-31) After execution, contains initial value of AC3.

AC3(16-31) Before execution, contains starting byte address for high-order byte; contents incremented by 1 with each byte stored. Effective address generated is confined to first 64 Kbytes of current segment.

After execution, contains address of next byte following last byte of string.

CARRY Set to 1 if number of significant digits to be stored is larger than specified string length; otherwise set to 0.

FPAC0–FPAC3	Can be individually specified for *fpac*; otherwise not used.
FPSR	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

STIX	Store Integer Extended
WSTI	Wide Store Integer
WSTIX	Wide Store Integer Extended

Exceptions

If the number in *fpac* has any fractional part, the result of **STI** is undefined. Use the Integerize instruction (**FINT**) to clear any fractional part.

If the destination field cannot contain the entire number being stored, digits are discarded until the number will fit into the destination. The remaining digits are stored and CARRY is set to 1.

For data types 0 through 6, high–order digits are discarded and low–order digits are stored.

For data type 7, low–order digits are discarded and high–order digits are stored.

If the number being stored will not fill the destination field:

For data types 0 through 5, the high–order bytes to the right of the sign are set to 0.

For data type 6, the sign bit is extended to the left to fill the field.

For data type 7, the low–order bytes are set to 0.

If the number in *fpac* is too large to be converted to the specified data type, a commercial fault is initiated.

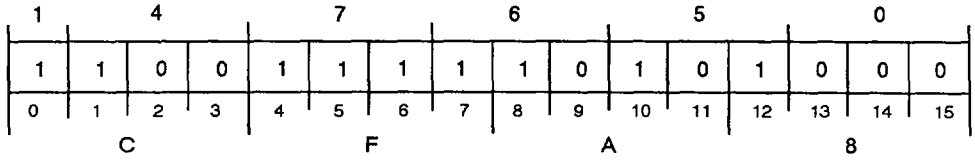
Example

```
XNLDA  1,DTYPE      ;AC1 contains the data type indicator.
XLEF   3,RESULT     ;Word pointer to integer field.
WADD   3,3          ;AC3 is a byte ponter to the integer.
STI    2            ;Convert the contents of FPAC2 into
                    ;a commercial integer, with the type
                    ;specified by AC1, and at the location
                    ;specified by AC3.
```

Store Integer Extended STIX

ECLIPSE Instruction

STIX



Function: fpac(0-3)[fp#] -> (E)[#]
AC3 -> AC2
0 -> CRY

Parameters: AC1 = data indicator -> unch
AC2 = x -> AC3
AC3 = byte pointer -> last bp + 1

NOTE: If E is not large enough, 1 -> CRY

STIX converts the contents of the four floating-point accumulators to an integer of the specified data-type format, and stores the result as a string in memory beginning at the specified byte location.

The string is structured from four 8-digit frames, each frame comprising the low-order 8 digits from an fpac conversion. The digits are stored right-aligned and in sequence with the least significant 8 digits (derived from FPAC3) stored at the higher address locations of the string. The digits derived from FPAC2, FPAC1, and FPAC0 (most-significant digits) are stored sequentially downward in the string. If the number of digits is not sufficient to fill the string, the remaining high-order digits are set to 0.

The sign of the stored integer is the logical OR of the signs of all four fpacs.

Arguments

None

Registers, Flags, and Stacks

AC0 Unused

AC1(16-31) Before execution, contains data type and number of digits for converted data. Specify from data types 0 through 5. **STIX** does not use the scale factor in the data type indicator.

After execution, contents unchanged.

AC2(16-31) After execution, contains initial value of AC3.

AC3(16-31) Before execution, contains starting memory location for high order byte. Effective address generated is confined to first 64 Kbytes of current segment.

After execution, contains address of next byte following last byte of string.

CARRY	Set to 1 if converted number to be stored is larger than specified; otherwise set to 0.
FPAC0–FPAC3	Before execution, each fpac holds floating–point double–precision word to be converted; FPAC0 contains the high eight digits; FPAC3 contains the low eight digits. After execution, contents unchanged.
FPSR	Undefined
Overflow	0
PC	PC + 1
Stack	Unchanged

Related Instructions

STI	Store Integer
WSTI	Wide Store Integer
WSTIX	Wide Store Integer Extended

Exceptions

If the value in any fpac is larger than $(10_{16})-1$, a commercial fault is initiated.

If the number in an fpac has any fractional part, the result is undefined. Use the Integerize (**FINT**) instruction to clear any fractional part.

If the destination field is not large enough to contain the number being stored, **STIX** disregards high–order digits until the number will fit in the destination. The instruction stores the low–order digits remaining and sets CARRY to 1.

If the number being stored will not fill the destination field, **STIX** sets the high–order bytes to 0.

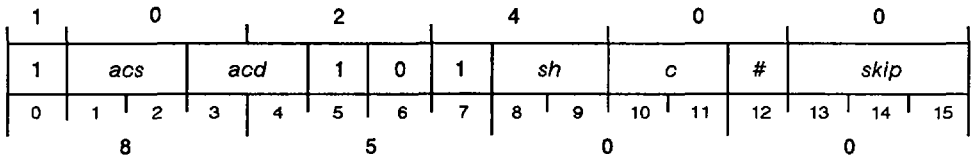
Example

```
XNLDA  1,DTYPE      ;AC1 contains the data type indicator.
XLEF   3,RESULT     ;Word pointer to integer field.
WADD   3,3          ;AC3 is a byte pointer to the integer.
STIX                ;Convert the contents of all four FPACs
                    ;into a commercial integer, with the type
                    ;specified by AC1, and at the location
                    ;specified by AC3.
```

Subtract SUB

ECLIPSE Instruction

SUB[*c*][*sh*][*#*] *acs*,*acd*[,*skip*]
(*skip* false return)
(*skip* true return)



Function: *acd* - *acs* -> *acd*

Parameters: None

SUB initializes CARRY to its specified value. The instruction subtracts the unsigned 16-bit integer in *acs* from the unsigned 16-bit integer in *acd* by taking the two's complement of the number in *acs* and adding it to the number in *acd*. **SUB** then places this result in the shifter, performs the specified shift operation, and places the final result in *acd* if the no-load bit is 0.

Arguments

[*c*] Processor determines effect of CARRY flag (*c*) on imitial value of CARRY before performing operation (opcode). Following table gives values of *c* and bits 10 and 11 and specifies operation.

Symbol [*c*]	Bits 10-11	Operation
omitted	0 0	Leave CARRY unchanged
Z	0 1	Initialize CARRY to 0
O	1 0	Initialize CARRY to 1
C	1 1	Complement CARRY

[*sh*] Processor shifts CARRY flag and 16 data bits after performing operation. Processor can shift bits left or right one bit position or can swap the two bytes. Following table gives values of *sh* and bits 8 and 9 and specifies shift operation.

Symbol [*sh*]	Bits 8-9	Shift Operation
omitted	0 0	Do not shift the result
L	0 1	Shift left
R	1 0	Shift right
S	1 1	Swap the two 8-bit bytes

[*#*] Except with no-load option (#), processor loads result of shift operation into destination accumulator. No-load option can test result of operation without destroying destination accumulator contents. Following table gives values of no-load option and bit 12 and specifies operation.

Symbol [#]	Bit 12	Operation
omitted	0	Load result into *acd*
#	1	Do not load result; restore initial CARRY flag

acs(16-31) Before execution, contains unsigned 16-bit integer.

After execution, contents unchanged unless *acs* and *acd* are same accumulator.

acd(16–31) Before execution, contains unsigned 16-bit integer.

After execution, contains result if no-load bit (#) is 0.

[*skip*] Processor skips next instruction if condition test true. Following table gives test conditions as specified by bits 13 to 15 and specifies operation.

Symbol [*skip*]	Bits 13-15	Operation
omitted	0 0 0	Never skip
SKP	0 0 1	Always skip
SZC	0 1 0	Skip if CARRY is 0
SNC	0 1 1	Skip if CARRY is not 0
SZR	1 0 0	Skip if result is 0
SNR	1 0 1	Skip if result is not 0
SEZ	1 1 0	Skip if either CARRY or result is 0
SBN	1 1 1	Skip if both CARRY and result are not 0

Skip omits next sequential 16-bit word. Make sure that skip does not transfer control to point within 32-bit or longer instruction.

Registers, Flags, and Stacks

AC0–AC3 Can be individually specified as *acs* and *acd*; otherwise unused.

CARRY If number in *acs* is less than or equal to number in *acd* (producing result greater than 65,535), initial CARRY is complemented. Then, if left or right shift occurs, final resulting CARRY is bit shifted into CARRY.

Overflow 0

PC PC + 1 (false exit)
PC + 2 (true exit)

PSR Unchanged

Stacks Unchanged

Related Instructions

NSUB Narrow Subtract

WSUB Wide Subtract

Exceptions

If the number in *acs* is less than or equal to the number in *acd* (producing a result greater than 65,535), **SUB** complements CARRY. (See also CARRY)

Do not specify **SUB** with no-load option (#) in combination with either never skip or always skip option. Instruction may not end in 1000_2 or 1001_2 (reserved for other instructions).

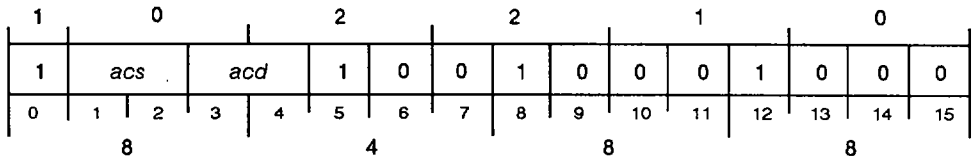
Example

```
SUB#   0,1,SZR      ;Does AC0[16-31] equal AC1[16-31]?
JMP    NOT_EQ       ;No. Not equal.
. . .               ;Yes. Equal.
```

Skip on Zero Bit SZB

ECLIPSE Instruction

SZB *acs,acd*
(bit is 1 return)
(bit is 0 return)



Function: If (E)bit = 0 then skip

Parameters: *acs* = base word pointer -> unchanged
acd = word offset & bit identifier -> unchanged

SZB tests the specified bit in memory and skips the next sequential word, if the addressed bit is zero.

The effective address generated is confined to the first 64 Kbytes of the current segment. If *acs* and *acd* are the same accumulator, then high-order word of address defaults to 0.

Arguments

acs(16-31)	Contains high-order 16 bits of 32-bit address.
acd(16-31)	Contains low-order 16 bits of 32-bit address (includes bit pointer).

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* or *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1 (test bit is 1) PC + 2 (test bit is 0)
Stack	Unchanged

Related Instructions

SZBO	Skip on Zero Bit and Set to One
SNB	Skip on Nonzero Bit

Exceptions

None

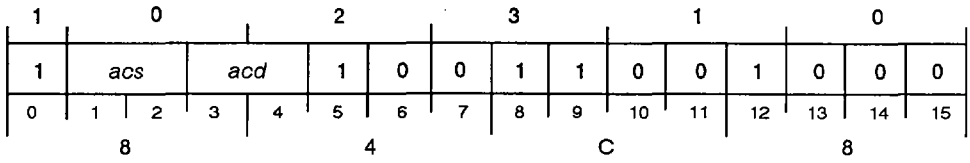
Example

```
    ELEF   0,FLAGS      ;Get word address of flags word.
    SUB    1,1          ;Get a zero in AC1.
    ADI    3,1          ;Get a 3 in AC1.
    SZB    0,1          ;Is bit 3 of the flags word set?
    JMP    SET          ;Yes.
    . . .               ;No.
    ...
FLAGS:    .WORD 0       ;Flags word.
```

Skip on Zero Bit and Set to One SZBO

ECLIPSE Instruction

SZBO *acs,acd*
(bit is 1 return)
(bit is 0 return)



Function: If (E)bit = 0 then skip
1 -> (E)bit

Parameters: *acs* = base word pointer -> unchanged
acd = word offset & bit identifier -> unchanged

NOTE: If *acs* is *acd*, base word pointer = 0 (of the current segment)

SZBO tests the specified bit in memory for 0.

If the bit is 0, it sets the bit to 1 and skips the next sequential word.

If the bit is 1, it remains unchanged and the next sequential word is executed.

SZBO is particularly useful for creating bit maps for such purposes as allocation of facilities (memory blocks, I/O devices, etc.) to several processes (or tasks) that may interrupt one another, or in a multiprocessor environment. Using **SZBO**, the bit is tested and set to 1 atomically.

The effective address generated is confined to the first 64 Kbytes of the current segment.

Arguments

acs(16–31) Contains high–order 16 bits of 32–bit address. If same accumulator as specified for *acd*, then high–order word of address defaults to 0.

acd(16–31) Contains low–order 16 bits of 32–bit address (includes bit pointer).

Registers, Flags, and Stacks

AC0–AC3 Can be individually specified as *acs* or *acd*; otherwise unused.

CARRY Unchanged

Overflow 0

PC PC + 1 (test bit is 1)
PC + 2 (test bit is 0)

Stack Unchanged

Related Instructions

SZB Skip on Zero Bit
SNB Skip on Nonzero Bit

Exceptions

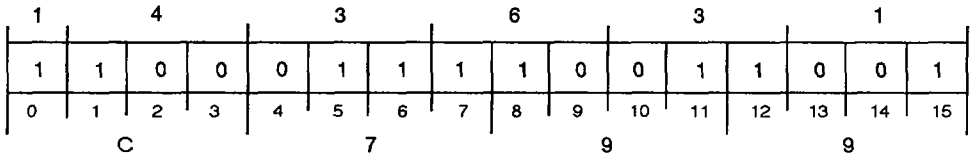
None

Example

```
    ELEF   0,FLAGS      ;Get word address of flags word.
    SUB    1,1          ;Get a zero in AC1.
    ADI    3,1          ;Get a 3 in AC1.
AGAIN:     SZBO 0,1     ;Is bit 3 of the flags word already set?
           JMP AGAIN    ;Yes. Try again to get the lock bit.
           . . .        ;No.        The bit was changed from 0 to 1, so
           .            ;we have the lock.
    ...
FLAGS:     .WORD 0      ;Flags word.
```

Skip on Valid Byte Pointer VBP

VBP
(Invalid pointer return)
(Normal return)



Function: byte pointer ?=? valid references
If (AC0 segment >= AC1 segment #) and (AC0 segment > = current segment) then skip

Parameters: AC0 = byte pointer -> unchanged
AC1 = segment #(1-3) -> unchanged

VBP checks a byte pointer contained in an accumulator for a valid ring-structured reference. The instruction, executing in a lower segment, compares the segment number in AC0 to the segment number in AC1 and to the current segment. If the byte pointer is valid, **VBP** skips the next word.

The byte pointer is valid if the segment number in AC0 is greater than or equal to the segment number in AC1 and is greater than or equal to the current segment. Otherwise, the byte pointer is invalid.

Arguments

None

Registers, Flags, and Stacks

AC0	Contains 32-bit byte pointer.
AC1(1-3)	Contains segment number; all other bits must contain zeros.
AC2-AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1 (Invalid pointer) PC + 2 (Normal return)
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load effective byte address
Use these instructions to load a byte address into AC0.

VWP Skip on Valid Word Pointer

Exceptions

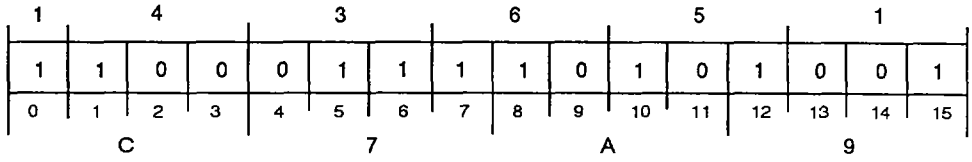
An invalid access (read, write, or execute) generates a protection violation.

Example

```
XWLDA 0,BYTE_POINTER          ;Get the byte pointer to check.
WLDAI 4S3,1                   ;Specify ring 4 in AC1.
VBP                           ;Validate the pointer.
WBR   BAD                     ;Byte pointer was bad.
. . .                         ;Byte pointer was OK.
```

Skip on Valid Word Pointer VWP

VWP
(Invalid pointer return)
(Normal return)



Function: word pointer ?=? valid references
If (AC0 segment >= AC1 segment #) and (AC0 segment > = current segment) then skip

Parameters: AC0 = word pointer -> unchanged
AC1 = segment #(1-3) -> unchanged

VWP checks a word pointer for a valid ring-structured reference. The instruction, executing in a lower segment, compares the segment number in AC0 to the segment number in AC1 and to the current segment. If the word pointer is valid, **VWP** skips the next word.

The word pointer is valid if all of the following conditions are true:

- The segment number in AC0 is greater than or equal to the segment number in AC1.
- The segment number in an indirect address is greater than or equal to the segment number in AC1 and the currently-referenced segment.
- If an indirection to a higher-numbered segment is followed by another indirection, the subsequent indirection(s) must be to the same segment or a highersegment.
- The segment number in the effective address (specified by AC0) is greater than or equal to the segment number in AC1 and the current segment.

Arguments

None

Registers, Flags, and Stacks

AC0	Contains 31-bit, indirectable word pointer.
AC1(1-3)	Contains segment number; all other bits must contain zeros.
AC2-AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1 (Invalid pointer) PC + 2 (Normal return)
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load effective address
: Use these instructions to load an effective word address into AC0.

VBP Skip on Valid Byte Pointer

Exceptions

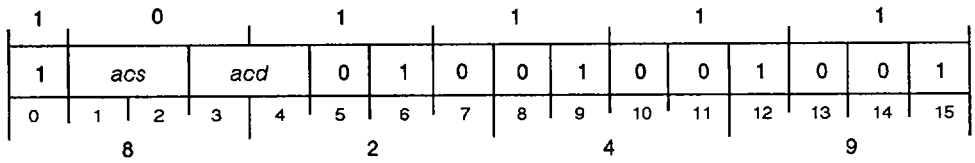
An invalid access (read, write, or execute) or more than 15 indirect addresses generate a protection violation.

Example

```
XWLDA 0,WORD_POINTER          ;Get the word pointer to check.
WLDAI 4S3,1                   ;Specify ring 4 in AC1.
VWP                           ;Validate the pointer.
WBR   BAD                     ;Byte pointer was bad.
. . .                         ;Byte pointer was OK.
```

Wide Add Complement WADC

WADC *acs,acd*



Function: $\overline{acs}$ + *acd* -> *acd*

Parameters: None

WADC forms the logical complement of the signed 32-bit integer contained in *acs* and adds it to the signed 32-bit integer contained in *acd*.

Arguments

acs	Before execution, contains signed 32-bit integer.
	After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd	Before execution, contains signed 32-bit integer.
	After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Set according to value of ALU CARRY.
Overflow	1 if ALU overflow
PC	PC + 1
PSR	OVR set to 1 if overflow occurs
Stack	Unchanged

Related Instructions

ADC	Add Complement

Exceptions

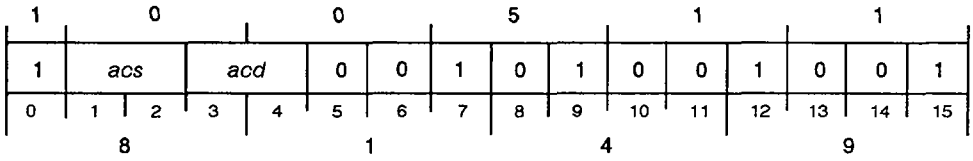
None

Example

```
WADC  1,1          ;Create a -1 in AC1.
```

Wide Add WADD

WADD *acs,acd*



Function: *acs + acd -> acd*
Parameters: None

WADD adds the signed 32-bit integer in *acs* to the signed 32-bit integer in *acd* and stores the result in *acd*.

Arguments

acs	Before execution, contains signed 32-bit integer.
	After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd	Before execution, contains signed 32-bit integer.
	After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Set according to value of ALU carry.
Overflow	1 if ALU overflow
PC	PC + 1
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

ADD	Add
NADD	Narrow Add

Exceptions

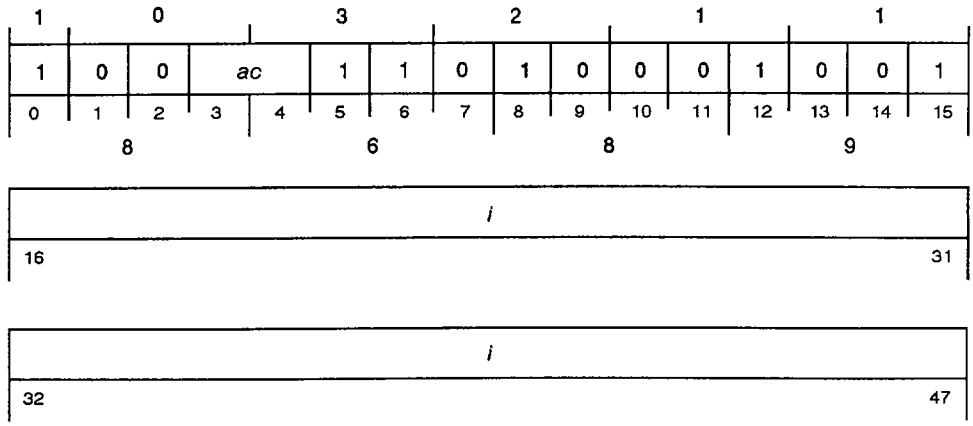
None

Example

```
        ;This subroutine appends one or two nulls to the end of a string,
        ;filling to a word boundary.
        ;
        ;Calling conventions:          XJSR NFILL
        ;                              <return>
        ;
        ;       AC1 = Byte pointer to start of string
        ;       AC2 = Length of stream
        ;       AC3 = Return address
NFILL:  WPSH    3,3             ;Save return address
        WSUB    3,3             ;Get a zero
        WADD    2,1             ;Get end of string
        WSTB    1,3             ;Append a null
        WINC    1,1             ;Bump pointer
        MOVR#   1,1,SZC         ;Check if odd (middle of word)
        WSTB    1,3             ;Yes, append another null
        LDAFP   3               ;AC3 contains frame pointer
        WPOPJ                   ;Return
```

Wide Add With Wide Immediate WADDI

WADDI *i,ac*



Function: *i + ac -> ac*

Parameters: None

WADDI adds the signed 32-bit integer in the immediate field to the signed 32-bit integer in the specified accumulator.

Arguments

i	Contains signed 32-bit integer.
ac	Before execution, contains signed 32-bit integer.
	After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Set with value of ALU CARRY
Overflow	1 if ALU overflow
PC	PC + 3
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

ADDI, NADDI, WNADI	Add a signed 16- or 32-bit immediate value to an accumulator
ADI, NADI, WADI	Add a 2-bit immediate value to an accumulator

Exceptions

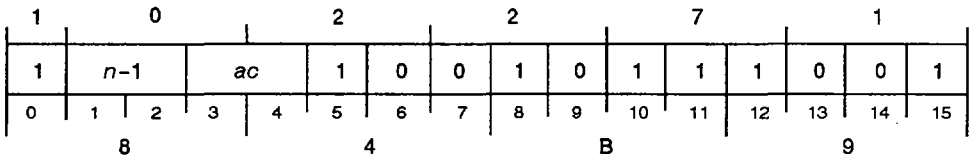
None

Example

```
XWLDA 3,FIRST      ;Get first value.
WADDI 40000000,3   ;Add a constant 40000000₈ to AC3.
XWSTA 3,RESULT     ;Store the result.
```

Wide Add Immediate WADI

WADI *n,ac*



Function: *n + ac -> ac*

Parameters: None

WADI adds an integer in the range 1–4 to the signed 32-bit integer contained in the specified accumulator.

Arguments

n	Integer in range 1–4. Assembler takes coded value of *n* and subtracts one from it before placing it in immediate field. Thus, programmer should code exact value to be added.
ac	Before execution, contains signed 32-bit integer. After execution, contains result.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Set with value of ALU CARRY.
Overflow	1 if ALU overflow
PC	PC + 1
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

ADI, NADI	Add a 2-bit immediate value to an accumulator.
ADDI, NADDI, WADDI, WNADI	Add a signed 16- or 32-bit immediate value to an accumulator.

Exceptions

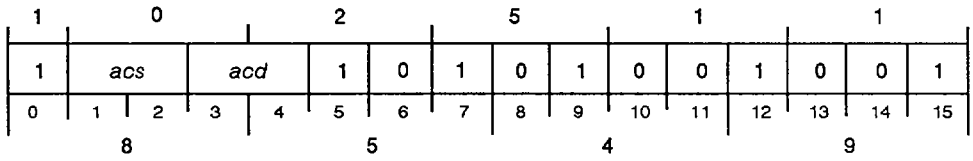
None

Example

```
XWLDA 3,FIRST      ;Get first value.
WADI  4,3          ;Add a constant 4 to AC3.
XWSTA 3,RESULT     ;Store the result.
```

Wide AND with Complemented Source WANC

WANC *acs,acd*



Function: $\overline{acs}$ AND *acd* -> *acd*

Parameters: None

WANC forms the one's complement of *acs* and logically ANDs it with the contents of *acd*.

Arguments

acs	Before execution, contains 32-bit value.
	After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd	Before execution, contains 32-bit value.
	After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

ANC	AND with Complemented Source

Exceptions

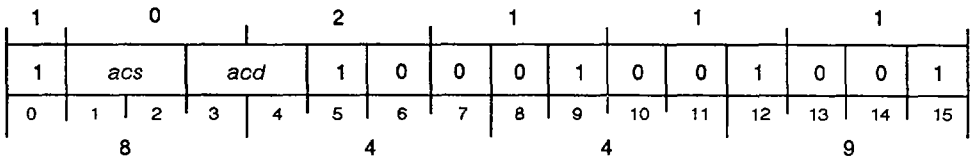
None

Example

```
XWLDA  0,FLAGS              ;Get the flags double word.
WLDAI  1B3+1B27+1B29,1      ;Get double word with bits 3, 27,
                            ;29 set.
WANC   0,1                  ;AND with complement of flags double
                            ;word.
WSEQ   1,1                  ;If the result is zero, bits 3,27 and
                            ;29 in the flags word were all set.
WBR    NOT_ALL_SET          ;
. . .                       ;All three were set.
```

Wide AND WAND

WAND *acs,acd*



Function: *acs* AND *acd* -> *acd*

Parameters: None

WAND forms the logical AND between corresponding bits of *acs* and *acd*, placing the result in *acd*.

Arguments

acs	Before execution, contains 32-bit value.
	After execution, contents unchanged.
acd	Before execution, contains 32-bit value.
	After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

AND	AND
ANC	AND with Complemented Source
WANC	Wide AND with Complemented Source

Exceptions

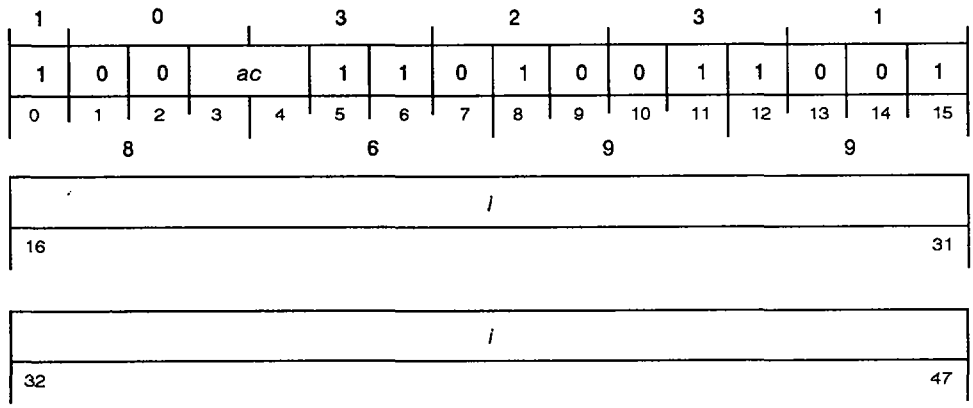
None

Example

```
LLEF  3,VARIABLE   ;Get the address of some variable.
NLDAI 1777,0       ;Get mask for offset within a page.
WAND  0,3          ;Mask the address to just offset.
```

Wide AND Immediate WANDI

WANDI *i,ac*



Function: *i* AND *ac* -> *ac*
Parameters: None

WANDI forms the logical AND between corresponding bits of the specified accumulator and the value contained in the immediate field, placing the result in the specified accumulator.

Arguments

i	32-bit immediate value
ac	Before execution, contains 32-bit value. After execution, contains result.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 3
PSR	Unchanged
Stack	Unchanged

Related Instructions

ANDI	AND Immediate

Exceptions

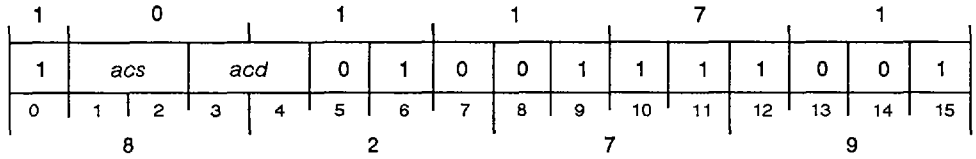
None

Example

```
    ;Convert lower case input to upper case.
    ;
    ;       AC0 = Byte pointer to string
CNVUC:  WLDB        0,2         ;Put a byte of source string into AC2
        WANDI       177,2       ;Mask to seven bits
        WCLM        2,2         ;See if lower case
        "A+40                   ;Lower limit for compare
        "Z+40                   ;Upper limit for compare
        WBR NOTLOW              ;Not lower case
        WNADI       -40,2       ;Yes, lower case, convert to upper
```

Wide Arithmetic Shift WASH

WASH *acs,acd*



Function: shift *acd*(*acs*(bits 24–31[+ = left,– = right])) -> *acd*

Parameters: None

WASH shifts the contents of *acd* left or right, according to the contents of *acs*.

Arguments

acs(24–31) Before execution, contains signed 8-bit integer specifying number of bits to shift and direction of shifting. Bits 0–23 are ignored.

If bit 24 is 0 (positive), instruction shifts contents of *acd* left and zero-fills vacated bit positions.

If bit 24 is 1 (negative), instruction shifts contents of *acd* right (rounding towards zero), and sign-bit fills vacated bit positions.

If number is zero, no shifting occurs.

After execution, contents unchanged unless *acs* and *acd* are same accumulator.

acd Before execution, contains value to be shifted.

After execution, contains result.

Negative values shifted right are rounded towards zero. For instance, –3 shifted one position right is –1.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0 (see Exceptions)
PC	PC + 1
PSR	OVR is set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

WMOVR	Wide Move Right
WHLV	wide Halve

Exceptions

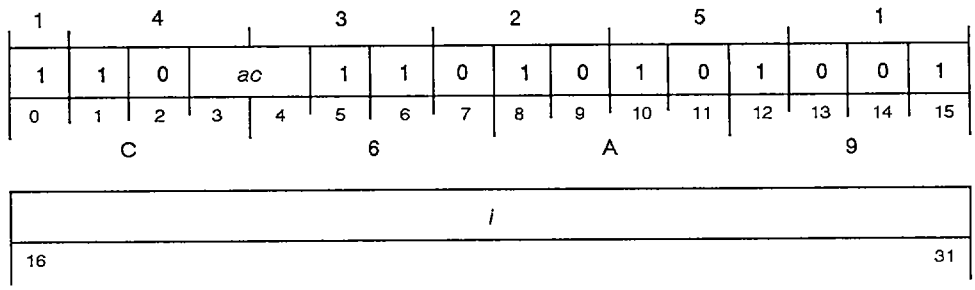
In a left shift, if the bit whose value is the complement of the sign bit of *acd* is shifted out, the result is correct, but *overflow* is 1.

Example

```
NLDAI  -2,3          ;Shift count is -2.
NLDAI  -5,0          ;Get a constant -5.
WASH   3,0           ;Shift AC0 two bit positions to the right.
```

Wide Arithmetic Shift With Narrow Immediate **WASHI**

WASHI *i,ac*



Function: shift *ac*(*i*[+ = left, − = right]) -> *ac*

Parameters: None

WASHI shifts the contents of the specified accumulator left or right according to the contents of the immediate field.

Arguments

i(24–31) Specifies number of bits to shift and direction of shifting. Bits 16–23 must be identical to bit 24 (sign bit); otherwise, results indeterminate. Processor sign–extends this value to 32 bits.

If bit 24 is 0 (positive) (1 to 32_{10}), **WASHI** shifts contents of *ac* left and zero–fills vacated bit positions.

If bit 24 is 1 (negative) (−1 to -32_{10}), **WASHI** shifts contents of *ac* right (rounding towards zero), and sign–bit fills vacated bit positions.

If *i* is zero, no shifting occurs.

ac Before execution, contains value to be shifted. Negative values shifted right are rounded towards zero. For instance, −3 shifted one position right is −1.

After execution, contains result.

Registers, Flags, and Stacks

AC0–AC3 Can be individually specified as *ac*; otherwise unused.

CARRY Unchanged

Overflow 0 (see Exceptions)

PC PC + 2

PSR OVR set to 1 if overflow occurs.

Stack Unchanged

Related Instructions

WMOVR Wide Move Right

WHLV Wide Halve

Exceptions

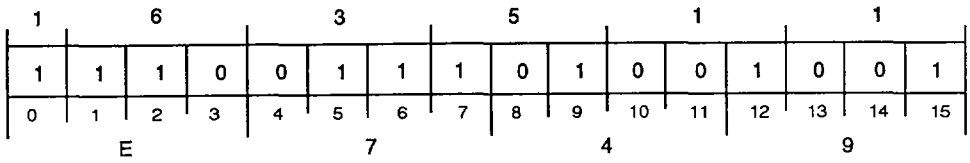
In a left shift, if the bit whose value is the complement of the sign bit of *ac* is shifted out, the result is correct, but an overflow occurs and PSR(OVR) is set to 1.

Example

```
NLDAI  -5,3          ;Get a constant -5.
WASHI  -2,3          ;Shift AC3 two bit positions to the right.
```

Wide Block Move WBLM

WBLM



Function: source @(AC2) -> destination @(AC3)

Parameters: AC1 = 2# number of words [+ = asc.; - = desc.] --> 0
AC2 = source E -> last E +/-1
AC3 = destination E -> last E +/-1

NOTE: If AC1 = 0, then no words are moved.

WBLM moves the specified number of memory words in consecutive (ascending or descending) order from the specified source location to the specified destination location. The words are treated as unsigned 16-bit integers.

Arguments

None

Registers, Flags, and Stacks

AC0	Unused
AC1	Before execution, contains signed 32-bit integer specifying number of words to be moved. If negative number, string moved in descending order; if positive number, string moved in ascending order. With each word moved, count increments (if negative number) or decrements (if positive number). If initial value 0, no words moved. After execution, contains 0.
AC2	Before execution, specifies source location in memory. With each word moved, value increments by 1 (if ascending) or decrements by 1 (if descending). After execution, contains pointer to next word after last word moved.
AC3	Before execution, specifies destination location in memory. With each word moved, value increments by 1 (if ascending) or decrements by 1 (if descending). After execution, contains pointer to next word after last word moved.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

WBLM	Wide Block Move

Exceptions

Because **WBLM** may have a long execution time, it is interruptible. If the instruction is interrupted, the program counter is decremented by one before being saved, thus it points to the interrupted instruction. Because addresses are updated after each word is stored, any interrupt service routine returning control via the saved program counter will correctly restart the instruction.

When updating the source and destination addresses, **WBLM** forces bit 0 of the result to 0. This ensures that on the return from an interrupt, the instruction will not try to resolve an indirect address in either AC2 or AC3.

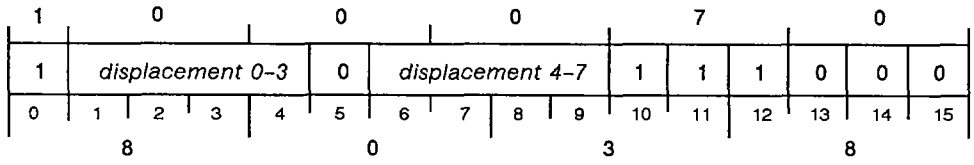
If a ring crossing is attempted in the descending mode, a protection trap is triggered, the instruction does not execute, and AC1 receives a value of 4.

Example

```
XLEF   2,DEST        ;Get the destination address.
XLEF   3,SOURCE      ;Get the source address.
NLDAI  7.,1          ;Move 7 words.
WBLM                 ;Do it.
```

Wide Branch WBR

WBR *displacement*



Function: PC + *displacement* -> PC

Parameters: None

WBR adds a specified value to the program counter.

Arguments

displacement Signed 8-bit integer

Registers, Flags, and Stacks

AC0-AC3 Unused

CARRY Unchanged

Overflow 0

PC PC + *displacement* (forced to reference location in current segment.)

PSR Unchanged

Stack Unchanged

Related Instructions

JMP Jump

Exceptions

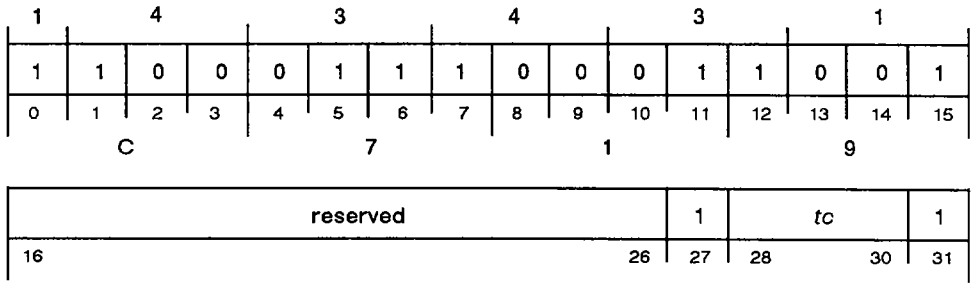
None

Example

```
    ;This subroutine deques 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
    ;                              <return>
    ;       AC1 = Queue descriptor address
    ;       AC2 = Element to be dequeued
PDEQ:   WSSVR       0           ;Save return block on stack
        WMOV        1,0         ;Move Queue address to AC0
        WMOV        2,1         ;Move dequeueing element 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!
```

Wide Backward Search Queue and Skip WBStc

WBS*tc*
(Unsuccessful return)
(Interrupt return)
(Successful return)



Function: Search from @(AC1) to Q tail for @(AC1 + AC3) = 32-bit test (*tc*)
=all 0 {AC}
=all 1 {AS}
=(wsp) {E}
<=(wsp) {GE}
>=(wsp) {LE}
≠(wsp) {NE}
=some 0s {SC}
=some 1s {SS}

Parameters: AC1 = E(first queue data element - E(Q element -- See Note)
AC3 = 2#(word offset) --> unch
(wsp) = mask word --> unch

NOTE: The call sequence for the Search Queue instruction is:

Search Queue instruction	
Unsuccessful Return	E(last element searched) --> AC1
Interrupt Return	E(next element to search) --> AC1
Successful Return	E(last element searched) --> AC1

WBS*tc* searches backward through a queue, examining a 32-bit data field. The processor locates the beginning queue element by calculating the effective address (AC1). The data field examined in this element is located by adding to AC1 the offset in AC3. The result is then compared to a 32-bit mask. The search continues until the processor reaches either the head of the queue or a data element that meets the test condition (*tc*).

Arguments

tc Bits 28-30 specify search condition.

tc Value	Bits 28-30 Encoding	Meaning
SS	0 0,0	Some of sampled test location bits 1.
SC	0 0,1	Some of sampled test location bits 0.
AS	0 1,0	All of sampled test location bits 1.
AC	0 1,1	All of sampled test location bits 0.
E	1 0,0	Mask and test location equal.
GE	1 0,1	Mask greater than or equal to test location.
LE	1 1,0	Mask less than or equal to test location.
NE	1 1,1	Mask and test location not equal.

NOTE: *For* **E, GE, LE,** *and* **NE** *test conditions, instruction treats values contained in mask and in test location as unsigned 32-bit integers.*

Registers, Flags, and Stacks

AC0	Unused
AC1	Before execution, contains effective address. With AC3, identifies location in data field as beginning data element in queue search. Processor increments AC1 for each data element it tests. If search succeeds, contains effective address of data element. New beginning pointer must be placed in AC1 if search is to continue through rest of queue. (If this is not done, and both the search condition and the examined data field in the element remain unchanged, continuing the search will result in this element being found again.) If search fails, contains effective address of last data element searched. If processor interrupts search (after unsuccessful search or another interrrupt only), contains effective address of next data element to be searched.
AC2	Unused
AC3	Before execution, contains signed 32-bit integer for word offset which is added to AC1 to identify location of data field in beginning data element in queue search. After execution, for all returns, contents unchanged.
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 2 (Unsuccessful return) Processor honors interrupts between completed search and execution of PC + 2 instruction. PC + 3 (Interrupt return) Processor honors interrupts between occurrence of interrupt and execution of PC + 3 instruction. PC + 4 (Successful return) Processor does not honors interrupts between completed search and execution of PC + 4 instruction.
PSR	Unchanged
Stack	Before execution, top wide stack double word contains mask, identifying test location bits to sample. After execution, for all returns, wide stack unchanged.

Related Instructions

Queue Management
Use these instructions to insert, delete, and test queue entries.

Exceptions

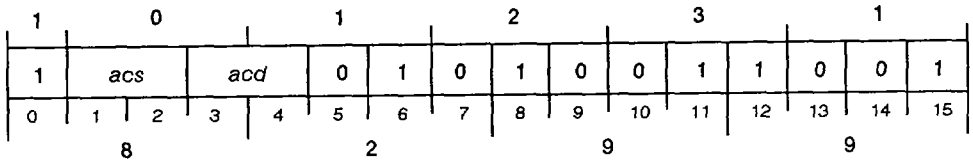
None

Example

```
   ;This example searches through a queue, finding all elements with a
   ;value of 6 at offset 4, and for all such elements changing the value
   ;to 0.
   WLDAI  6,0          ;Push the value to search
   WPSH   0,0          ;for onto the stack.
   LWLDA  1,TAIL       ;Put address of last queue element in AC1
                       ;to start search.
   WLDAI  4,3          ;Field to test is at offset 4 in each element.
REPEAT:   WBSE         ;Find an element whose data field equals 6.
          JMP DONE     ;If none found, all done.
          JMP REPEAT   ;If interrupted, just continue.
   WMOV   1,2          ;Copy address of found element to AC2.
   WSUB   0,0          ;Put a 0 in AC0.
   XWSTA  0,4,2        ;Store it in offset 4 in the element.
   JMP    REPEAT       ;Go look for next element.
DONE:     WPOP 0,0     ;Restore stack.
   .
   .
   .
TAIL:     .DWORD
```

Wide Set Bit to One — WBTO

WBTO *acs,acd*



Function: 1 -> @(*acs* + *acd*)bit

Parameters: *acs* = base word pointer -> unchanged
acd = word offset & bit pointer -> unchanged

NOTE: If *acs* is *acd*, base word identifier = 0 (of the current segment)

WBTO sets the specified bit to one. **WBTO** is an indivisible instruction.

Arguments

acs — Before execution, contains indirectable word address.

If *acs* and *acd* are specified to be the same accumulator, the processor assumes the word address is zero within the current segment. In this case, the specified accumulator contains the word offset and bit identifier.

After execution, contents unchanged.

acd — Before execution, contains word offset and bit identifier.

After execution, contents unchanged.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

BTO, BTZ, WBTO
Set bit to one or zero.

Exceptions

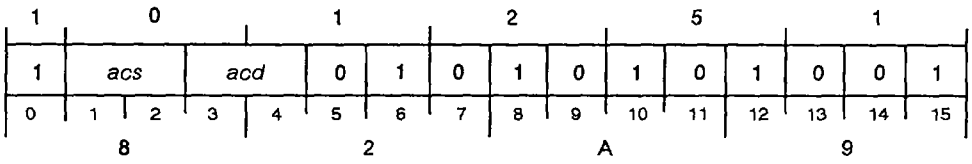
None

Example

```
XLEF   0,FLAGS      ;Get the flags word address
NLDAI  5,1          ;We want to set bit 5 of the flags word.
WBTO   0,1          ;Set the bit.
```

Wide Set Bit to Zero WBTZ

WBTZ *acs,acd*



Function: 0 -> @(*acs* & *acd*)bit

Parameters: *acs* = base word pointer -> unchanged
acd = word offset & bit pointer -> unchanged

NOTE: If *acs* is *acd*, base word identifier = 0 (of the current segment)

WBTZ sets the specified bit to zero. **WBTZ** is an indivisible instruction.

Arguments

acs Before execution, contains indirectable word address.

If *acs* and *acd* are specified to be the same accumulator, the processor assumes the word address is zero within the current segment. In this case, the specified accumulator contains a word offset and a bit identifier.

After execution, contents unchanged.

acd Before execution, contains word offset and bit identifier.

After execution, contents unchanged.

Registers, Flags, and Stacks

AC0–AC3 Can be individually specified as *acs* and *acd*; otherwise unused.

CARRY Unchanged

Overflow 0

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

WBTO, BTO, BTZ
Set bit to one or zero.

Exceptions

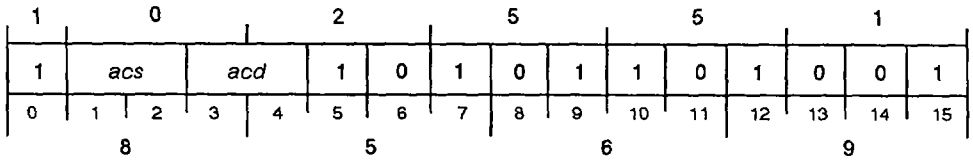
None

Example

```
    ;This subroutine deques 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
    ;                              <return>
    ;       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 dequeueing element 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!
```

Wide Compare to Limits WCLM

WCLM *acs,acd*
(if *acs* ≠ *acd* and integer not within limits return)
(if *acs* ≠ *acd* and integer within limits return)
(if *acs* = *acd* and integer not within limits return)
(if *acs* = *acd* and integer within limits return)



Function: L <= *acs* <= H then skip

Parameters: *acs* = 2# -> unch

NOTE: If *acs* is not *acd*:
@(*acs*) = L
@(*acd* + 2) = H
If *acs* is *acd*:
@(WCLM + 1) = L
@(WCLM + 3) = H

WCLM compares a signed 32-bit integer in *acs* with two other signed 32-bit integers (lower limit L and higher limit H).

If the integer in *acs* is equal to or between L and H, **WCLM** skips the next sequential word.

If the integer in *acs* is less than L or greater than H, the next sequential word is executed.

The specification of *acd* determines the location of L and H.

Arguments

acs — Contains signed 32-bit integer for comparison.

acd — If specification is different from *acs*, contains address of lower limit double-word value L; higher limit value H is contained in next double-word location following L.

If specified same as *acs*, limits L and H are in next two respective double-word locations following instruction.

Values of L and H must be expressed as signed 32-bit integers.

Registers, Flags, and Stacks

AC0–AC3 — Can be individually specified as *acs* and *acd*; otherwise unused.

CARRY — Unchanged

Overflow — 0

PC — If *acs* not ≠ *acd*:
PC + 1 (integer not within limits)
PC + 2 (integer within limits)

If *acs* = *acd*:
PC + 5 (integer not within limits)
PC + 6 (integer within limits)

PSR	Unchanged
Stack	Unchanged

Related Instructions

CLM	Compare to Limits

Exceptions

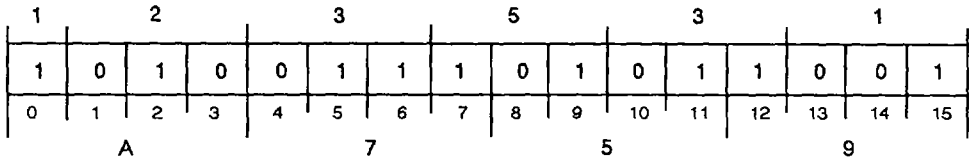
None

Example

```
    ;Convert lower case input to upper case.
    ;
    ;AC0 = Byte pointer to string
CNVUC:  WLDB        0,2         ;Put a byte of source string into AC2
        WANDI       177,2       ;Mask to seven bits
        WCLM        2,2         ;See if lower case
        .DWORD "A+40 ;          Lower limit for compare
        .DWORD "Z+40 ;          Upper limit for compare
        WBR NOTLOW              ;Not lower case
        WNADI       -40,2       ;Yes, lower case, convert to upper
        ...
NOTLOW:
```

Wide Character Compare WCMP

WCMP



Function: string 1 ?=? string 2
Result returned -> AC1

Parameters: AC0 = str2 #bytes[+ = asc, - = desc] -> 0 or uncompared bytes
AC1 = str1 #bytes[+ = asc, - = desc] -> result
-1 (str1 < str2)
0 (str1 = str2)
+1 (str1 > str2)
AC2 = str2 bp -> last bp +/-1 or failing byte
AC3 = str1 bp -> last bp +/-1 or failing byte

NOTE: Longer string compared against spaces when shorter string exhausted.

WCMP compares two strings of bytes, a byte at a time from each string, and halts when two bytes do not match or when the strings are completed. With each mismatch, the instruction returns a code reflecting the type of mismatch. Each byte is treated as an unsigned 8-bit integer in the range $0-255_{10}$.

If no mismatches occur, the instruction completes the comparison for the maximum number of bytes and returns a code indicating that both strings compare. At completion of the instruction, both strings remain unchanged.

The strings may overlap in any way; the overlap does not affect execution.

Arguments

None

Registers, Flags, and Stacks

AC0 Before execution, contains signed 32-bit integer indicating length and direction of comparison for string 2.

If string is compared from lowest memory location to highest, contains positive value of number of bytes in string 2.

If string is compared from highest memory location to lowest, contains negative value of number of bytes in string 2.

After execution, contains number of bytes (or two's complement of number of bytes) left to compare in string 2.

AC1 Before execution, contains signed 32-bit integer indicating length and direction of comparison for string 1.

If string is compared from lowest memory location to highest, contains positive value of number of bytes in string 1.

If string is compared from highest memory location to lowest, contains negative value of number of bytes in string 1.

After execution, contains a return code as follows:

Code	Meaning
-1	string 1 byte < string 2 byte
0	string 1 byte = string 2 byte
+1	string 1 byte > string 2 byte
4	invalid pointer (protection fault error)

AC2	Before execution, contains memory address for first byte to be compared in string 2. When string is to be compared in ascending order, points to lowest byte. When string is to be compared in descending order, points to highest byte. With each successful comparison, address is incremented or decremented, depending on direction of compare. After execution, contains byte address either of failing byte (if mismatch found) or of next byte following string 2 (if both strings compare).
AC3	Before execution, contains memory address for first byte to be compared in string 1. When string is compared in ascending order, points to lowest byte. When string is compared in descending order, points to highest byte. With each successful comparison, address is incremented or decremented, depending on direction of comparison. After execution, contains byte address either of failing byte (if mismatch found) or of next byte following string 1 (if both strings compare).
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

CMP	Character Compare

Exceptions

Because **WCMP** may have a long execution time, it is interruptible. If the instruction is interrupted, the program counter is decremented by one before being saved, thus it points to the interrupted instruction. Because addresses are updated after each byte is compared, any interrupt service routine returning control via the saved program counter will correctly restart the instruction.

If both strings are defined with zero length, no comparisons are made and the result returned is 0. If the two strings are unequal in length, on completion of the shorter string the comparisons continue, using space characters (040_8) for comparison with remaining bytes of the longer string.

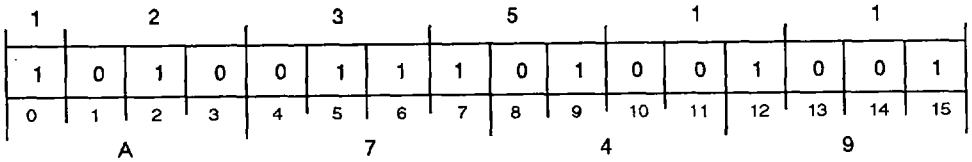
If the addresses are not valid byte–pointers within the user's address space, a protection fault may occur, even if no bytes are to be compared.

Example

```
    .TITLE    WCMP
    .ENT  START, LESS, EQUAL, GREATER, STR1, STR2
    .NREL
    ;The following program will compare the contents of string1 to
    ;string2, and
    ;will jump to one of three locations depending on the outcome.
START:    LWDA        0,LEN2        ;get the lengths of the strings
          LWDA        1,LEN1
          LLEFB       2,2*STR2      ;point to the strings
          LLEFB       3,2*STR1
          WCMP                      ;do the compare
          WMOV        1,2           ;move the -1/0/1 to ac2
          XJMP        TABLE,2       ;use the -1/0/1 to jump into the table
   WBR    LESS                      ;<
TABLE:    WBR EQUAL     ;=
          WBR GREATER   ;>
LESS:     WBR EXIT                  ;string1 is less than string2
EQUAL:    WBR EXIT                  ;strings are equal
GREATER:  WBR EXIT                  ;string1 is greater than string2
EXIT:     WSUB        2,2
          ?RETURN
    ;     DATA
STR1:     .TXT "ABCDEFG"            ;buffers for strings
STR2:     .TXT "ABCDEFG"
LEN1:     7                         ;length of string1
LEN2:     7                         ;length of string2
.END      START
```

Wide Character Move Until True WCMT

WCMT



Function: Source @(AC3) --> destination @(AC2)
If byte = delimiter; terminate instruction, byte not moved

Parameters: AC0 = delimiter table address --> E(delimiter table)
AC1 = # bytes [+ = ascending; - = descending] --> 0 or # unmoved bytes
AC2 = destination byte pointer --> last byte pointer +/-1
AC3 = source byte pointer --> last byte pointer +/-1

NOTE: If AC0 = 0, the instruction is a No-op.
If AC2 = AC3, no bytes are written, but string is scanned for delimiter.

WCMT moves a string of bytes, one at a time, from one area of memory to another, until either a table-specified delimiter character is encountered or the specified number of bytes has been transferred.

Before each byte is moved, its value (an unsigned 8-bit integer in the range $0–255_{10}$) is used as a bit index into a 256-bit delimiter table.

If the indexed bit in the delimiter table is 0, the byte is not a delimiter; it is copied from the source string into the destination string.

If the indexed bit in the delimiter table is 1, the byte is a delimiter; the byte does not get copied, and the instruction terminates with AC3 containing the address of the delimiter.

Both strings are processed in the same direction, either from the lowest specified memory locations upward (ascending order), or from the highest specified memory locations downward (descending order). The source and destination strings may overlap in any way; however, since one byte is moved at a time, certain types of overlap may produce undesired results.

Arguments

None

Registers, Flags, and Stacks

AC0 Before execution, contains word address, possibly indirect, of start of 256-bit (16-word) delimiter table.

After execution, contains resolved address of delimiter table.

AC1 Before execution, specifies total number of bytes in source string to be moved and direction in which strings are to be processed.

If ascending order used, contains positive value of number of bytes in source string.

If descending order used, contains negative value of number of bytes in source string.

After execution, contains number of bytes (or two's complement of number of bytes) not moved.

AC2	Before execution, contains byte address for first byte to be written in destination string. When string written in ascending order, points to lowest byte. When string written in descending order, points to highest byte. With each byte stored, address incremented or decremented, depending on direction of processing. After written execution, contains address of next byte following last byte in string. (If AC2 equals AC3 before execution, they are also equal after execution, even though no writes are performed.)
AC3	Before execution, contains byte address for first byte to be accessed in source string. When source string accessed in ascending order, points to lowest byte. When string accessed in descending order, points to highest byte. With each byte accessed, address incremented or decremented, depending on direction of processing. After execution, contains address of delimiter or, if none found, next byte following last byte in string.
CARRY	Indeterminate
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

CMT	Character Move Until True

Load effective address
: Use these instructions to place a word address in AC0 or byte addresses in AC2 and AC3.

Load with immediate
: Use these instructions to load AC1 with the appropriate value.

Exceptions

Because **WCMT** may have a long execution time, it is interruptible. If the instruction is interrupted, the program counter is decremented by one before being saved, thus it points to the interrupted instruction. Because addresses are updated after each byte is moved, any interrupt service routine returning control via the saved program counter will correctly restart the instruction.

The original contents of AC0, AC2, and AC3 must be valid pointers to some area in the user's address space. If the addresses are invalid, a protection fault may occur (even if no bytes are to be moved) with error code 4 stored in AC1.

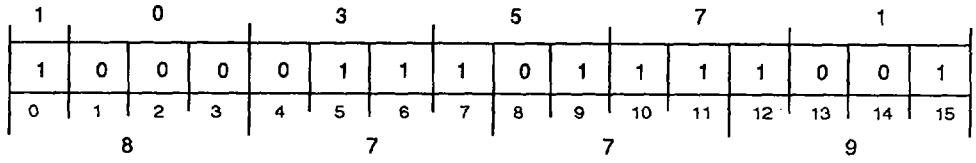
If AC2=AC3, no bytes are written, but the string is scanned for a delimiter.

Example

```
    .TITLE WCMT
    .ENT   START, RESULT, STR1
    .NREL
    .RDX   16.
    ;WCMT
    ;move characters from a source string to a destination string up to
    ;but not including the first comma.
START:    LLEF        0,DELIMS     ;address of comma delimiter table
          NLADI       46.,1        ;length of source string
          LLEFB       3,2*STR1     ;point to source and dest strings
          LLEFB       2,2*RESULT
          WCMT                     ;move until we find the comma
          WSUB        2,2
          ?RETURN
    ;result buffer now holds 'PROGRAM_NAME/L=FOO'
    ;ac3 points past end of text in result
    ;ac2 points to the first comma in str1 DATA
STR1:     .TXT        'PROGRAM_NAME/L=FOO,:UDD:FOMA,:MACROS:PRINT.CLI'
RESULT:   .BLK        100.
DELIMS:   .DWORD      0            ;the delimiter table
          .DWORD      80000        ;bit for the comma is set
          .DWORD      0
          .DWORD      0
          .DWORD      0
          .DWORD      0
          .DWORD      0
          .DWORD      0
          .END        START
```

Wide Character Move WCMV

WCMV



Function: Source @(AC3) --> destination @(AC2)
Relative length --> CARRY

Parameters: AC0 = destination #bytes[+ = ascending; - = descending] --> 0
AC1 = source #bytes[+ = ascending; - = descending] --> 0 or # unmoved bytes
AC2 = destination byte pointer --> last byte pointer +/- 1
AC3 = source byte pointer --> last byte pointer +/- 1
CARRY = x --> relative length:
0 = source =< destination
1 = source > destination

NOTE: If source < destination, remainder of destination is filled with spaces.

WCMV moves a string of bytes, one at a time, from one area of memory to another, and returns a value in CARRY reflecting the relative lengths of the source and destination strings.

The source and destination strings may be individually processed either from the lowest specified memory locations upward (ascending order), or from the highest specified memory locations downward (descending order). The strings may overlap in any way; overlap does not affect execution of the instruction.

Arguments

None

Registers, Flags, and Stacks

AC0 Before execution, contains length and direction of processing for destination string.

If ascending order used, contains positive value of number of bytes in string.

If descending order used, contains negative value of number of bytes in string.

After execution, contains 0.

AC1 Before execution, contains length and direction of processing for source string.

If ascending order used, contains positive value of number of bytes in string.

If descending order used, contains negative value of number of bytes in string.

After execution, contains number of bytes (or two's complement of number of bytes) left unmoved in source string.

AC2	Before execution, contains byte address for first byte to be written in destination string. When string is written in ascending order, points to lowest byte. When string is written in descending order, points to highest byte. With each byte moved, address incremented or decremented, depending on direction of processing. After execution, contains address for next byte following string.
AC3	Before execution, contains byte address for first byte to be accessed in source string. When string is accessed in ascending order, points to lowest byte. When string is accessed in descending order, points to highest byte. With each byte moved, address incremented or decremented, depending on direction of accessing. After execution, contains address for next byte following last byte fetched.
CARRY	If 1, source number of bytes is > destination number of bytes; otherwise 0.
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

CMV Character Move

Load with immediate
: Use these instructions to load AC0 and AC1 with the appropriate values.

Load effective byte address
: Use these instructions to place byte addresses in AC2 and AC3.

Exceptions

Because **WCMV** may have a long execution time, it is interruptible. If the instruction is interrupted, the program counter is decremented by one before being saved, thus it points to the interrupted instruction. Because addresses are updated after each byte is moved, any interrupt service routine returning control via the saved program counter will correctly restart the instruction.

If the destination string is longer than the source string, when the source string is completed the remaining locations of the destination string are filled with space characters.

If the initial value of AC0 is 0, no bytes are fetched and none are stored. If the initial value of AC1 is 0, no bytes are fetched and the destination field is filled with spaces.

If the contents of AC2 and AC3 are not valid byte pointers to some area in the user's address space, a protection fault may occur (even if no bytes are to be moved).

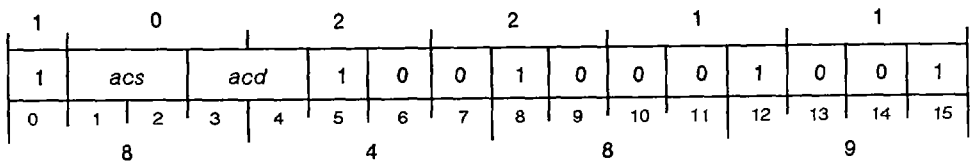
If a backward move would cause an inward ring crossing, a protection fault occurs before **WCMV** begins executing.

Example

```
        LLEFB  2,DEST*2      ;Get the destination byte address.
        LLEFB  3,SOURCE*2    ;Get the source byte address.
        NLDAI  32.,0         ;Set up to move 32 bytes to destination.
        WMOV   0,1           ;Also 32 bytes from the source.
        WCMV                 ;Move them all.
        .
        .
        .
DEST:   .BLK 16.             ;32 bytes.
SOURCE: .BLK 16.             ;32 bytes.
```

Wide Count Bits — WCOB

WCOB *acs,acd*



Function: *acs*(# of 1s) + *acd* -> *acd*

Parameters: None

WCOB counts the number of ones in *acs* and adds the number to the signed 32-bit integer in *acd*.

Arguments

acs — Before execution, contains source word for bit count.

After execution, contents unchanged unless *acs* and *acd* are same accumulator.

acd — Before execution, contains signed 32-bit integer.

After execution, contains initial *acd* value plus number of nonzero bits in *acs*.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

COB — Count Bits

Exceptions

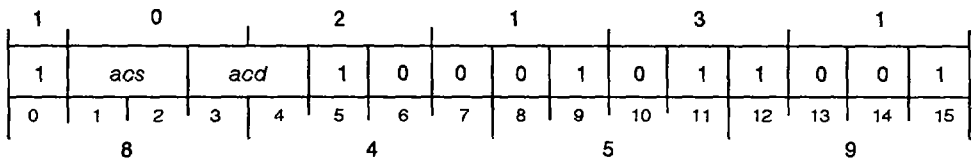
None

Example

```
WSUB  0,0          ;Start with 0 in AC0.
WADC  1,1          ;Set AC1 to all ones.
WCOB  1,0          ;Adds 32 to AC0. New value is 32.
```

Wide Complement WCOM

WCOM *acs,acd*



Function: $\overline{acs}$ -> *acd*

Parameters: None

WCOM forms the one's complement of a 32-bit integer in *acs*, placing the result into *acd*.

Arguments

acs	Before execution, contains 32-bit integer.
	After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd	After execution, contains result.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

COM	Complement

Exceptions

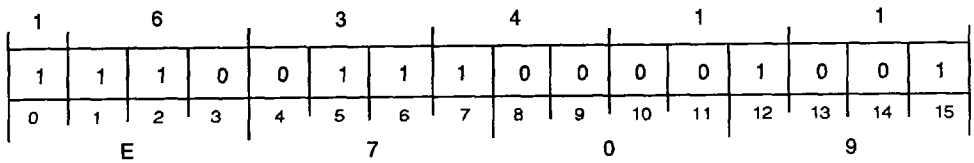
None

Example

```
WADC  1,1          ;Get all ones in AC1.
WCOM  1,0          ;Complement AC1, giving all zeros in AC0.
```

Wide Character Scan Until True WCST

WCST



Function: If @(AC3) = delimiter = halt instruction
? -> CRY

Parameters: AC0 = delimiter table address -> E(delimiter table)
AC1 = #bytes [+ = acs, - = desc] --> 0 or # unscanned bytes + 1
AC3 = bp -> lst bp +/-1 or bp of delimiter

WCST, under control of three accumulators, scans a string of bytes until either a table-specified delimiter character is found or the string is exhausted.

The instruction scans the string one byte at a time, testing each value to determine whether it is a delimiter. It treats the byte as an unsigned eight-bit integer (in the range of $0-255_{10}$) and uses it as a bit index into a 256-bit delimiter table.

If the indexed bit in the delimiter table is zero, the byte is not a delimiter, and the instruction processes the next byte.

If the indexed bit in the delimiter table is one, the byte is a delimiter, and the instruction terminates.

When the string is exhausted, the instruction terminates.

Arguments

None

Register, Flags, and Stacks

AC0 Before execution, contains word address, possibly indirect, of start of 256-bit (16-word) delimiter table.

After execution, contains resolved address of delimiter table.

AC1 Before execution, contains length of string and direction of processing.

If string is scanned in ascending order (lowest memory location to highest), AC1 contains positive value of number of bytes in string.

If string is scanned in descending order (highest memory location to lowest), AC1 contains negative value of number of bytes in string.

If no bytes are to be scanned, AC1 contains zero.

After execution, contains number of bytes (or two's complement of number of bytes) not scanned plus one. (See Exceptions.)

AC2 Unused

AC3	Before execution, contains byte pointer to first byte to be processed in string. When processing in ascending order, AC3 points to lowest byte in string. When processing in descending order, AC3 points to highest byte in string. After execution, contains byte pointer to either delimiter or first byte following string.
CARRY	Indeterminate
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load effective address
: Use the load word address instructions to load the address of the delimiter table into AC0. Use the load byte address instructions to enter the byte address into AC3.

Load with immediate
: Use these instructions to place the appropriate value into AC1.

Exceptions

Because **WCST** may have a long execution time, it is interruptible. If the instruction is interrupted, the program counter is decremented by one before being saved, thus it points to the interrupted instruction. Because addresses are updated after each byte is scanned, any interrupt service routine returning control via the saved program counter will correctly restart the instruction.

The original contents of AC0 and AC3 must be valid pointers to an area in the user's address space. If they are invalid, a protection fault may occur, even if no bytes are to be scanned, with error code 4 returned to AC1.

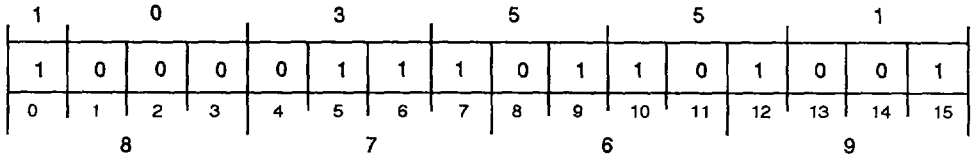
Example

```
    TITLE WCST
    ENT   START, STR1
    NREL
    RDX   16.
    ;WCST
    ;
    ; SCAN characters in a string until the first comma (inclusive).
START:  LLEF        0,DELIMS     ;address of comma delimiter table
        NLDAI       46.,1        ;length of string
        LLEFB       3,2*STR1     ;point to first byte of string
        WCST                     ;SCAN until we find the comma
                                 ;upon completion of WCST instruction:
                                 ;ac3 points to the comma of
                                 ;'...FOO,:UDD...'
```

```
          WINC        3,3          ;Point to ':' of ':UDD...'
    . . .
          WSUB        2,2
          ?RETURN
    ;     DATA
STR1:     .TXT        'PROGRAM_NAME/L=FOO,:UDD:BARON,:MACROS:PRINT.CLI'
DELIMS: .DWORD        0            ;the delimiter table
          .DWORD      80000        ;bit for the comma is set
          .DWORD      0
          .DWORD      0
          .DWORD      0
          .DWORD      0
          .DWORD      0
          .DWORD      0
          .END        START
```

Wide Character Translate WCTR

WCTR



Function: source @(AC3) (translates) --> destination @(AC2)

Parameters: AC0 = address of translation table byte pointer --> unchanged
AC1 = # bytes [-2#] --> 0
AC2 = destination byte pointer --> last byte pointer + 1
AC3 = source byte pointer --> last byte pointer + 1

-OR-

Function: source @(AC3) (translates) =? destination @(AC2) result code --> AC1

Parameters: AC0 = address of translation table byte pointer --> unch
AC1 = # bytes [+#] --> result:
-1 (S < D)
0 (S = D)
+1 (S > D)
AC2 = destination byte pointer --> last byte pointer + 1 or to failing byte
AC3 = source byte pointer --> last byte pointer + 1 or to failing byte

WCTR has two different operating modes: *translate-and-move*, or *translate-and-compare*.

In the *translate-and-move* mode, **WCTR** translates a string of bytes, one at a time, from one data representation to another, and moves the translated results into a corresponding string in another area of memory.

In the *translate-and-compare* mode, **WCTR** translates bytes from two strings, a byte at a time from each string, and compares the translated results. Each translated byte is treated as an unsigned 8-bit integer in the range $0\text{-}255_{10}$. If the translated bytes from both strings are equal, the process continues until either the specified number of bytes is processed or a pair of bytes are not equal. If two translated bytes are not equal, the instruction ends and returns a result code in AC1. The string for which the byte has the smaller numerical value is defined as the lower-valued string.

Translation uses each byte as an 8-bit index into a 256-byte translation table. The byte addressed by the index then becomes the translated value. Regardless of operating mode, source strings for translations remain unchanged after execution.

In both modes, the strings are processed from the specified starting addresses upward in single-byte increments. The source and destination strings may overlap in any way; however, certain overlaps may produce undesired results.

Arguments

None

Registers, Flags, and Stacks

AC0 Before execution, contains address, direct or indirect, of double memory word containing byte pointer to first byte in 256-byte translation table.

After execution, contains address of double word containing byte pointer to translation table.

AC1 Before execution, contains total number of bytes in each string and defines operating mode.

In *translate-and-move* mode, number of bytes in strings expressed as negative value.

In *translate-and-compare* mode, number of bytes in strings expressed as positive value.

With each byte of source string processed, value is either incremented or decremented, depending on operating mode.

After execution, value dependent upon operating mode.

In *translate-and-move* mode, contains 0.

In *translate-and-compare* mode, contains code defined as follows:

Code	Meaning (translated byte values)
-1	String 1 byte < string 2 byte
0	String 1 byte = string 2 byte
+1	String 1 byte > string 2 byte

AC2 Before execution, contains 32-bit byte pointer to first byte in destination string (string 2). With each byte accessed, address is incremented by 1.

After execution, contains either byte pointer to byte following string 2, or, if inequality found in *translate-and-compare* mode, byte pointer to failing byte in string 2.

AC3 Before execution, contains 32-bit byte pointer to first byte in source string (string 1). With each byte accessed, address is incremented by 1.

After execution, contains either byte pointer to byte following string 1, or, if inequality found in *translate-and-compare* mode, byte pointer to failing byte in string 1.

CARRY Unchanged

Overflow 0

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

CTR Character Translate

Load effective address
Use these instructions to place a word address in AC0 or byte addresses in AC2 and AC3.

Load with immediate
Use these instructions to load AC1 with the appropriate value.

WNEG Use the Wide Negate instruction to form the two's complement of the number in AC1.

Exceptions

Because **WCTR** may have a long execution time, it is interruptible. If the instruction is interrupted, the program counter is decremented by one before being saved, thus it points to the interrupted instruction. Because addresses are updated after each byte operation, any interrupt service routine returning control via the saved program counter will correctly restart the instruction.

If the length of both string 1 and string 2 is zero, the *translate-and-compare* mode returns a zero in AC1.

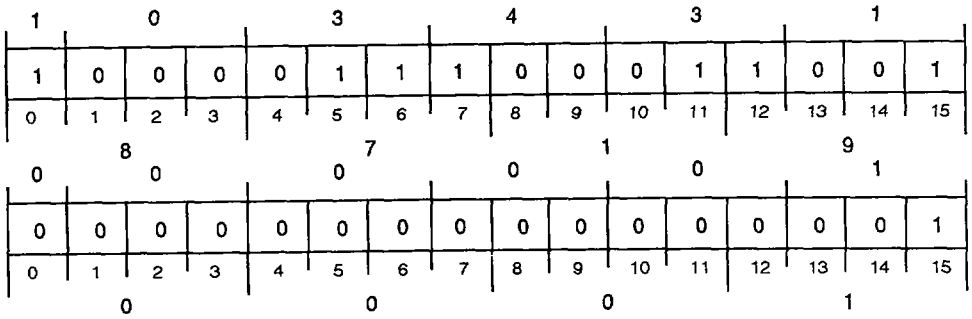
If the contents of AC0, AC2, and AC3 are not valid addresses to some area in the user's address space, a protection fault may occur (even if no bytes are to be moved or compared) with error code 4 stored in AC1.

Example

```
    .TITLE    WCTR
    .NREL
    .ENT   START, SOURCE, DEST
    ;WCTR
    ;translate all non-alphanumeric characters in a string to blanks
START;    LLEF         0,TBLPTR     ;address of translate table
          NLADI        -100.,1      ;2's complement of length of
                                    ;source string -- translate and move
          LLEFB        2,2*DEST     ;point to source and dest
          LLEFB        3,2*SOURCE
          WCTR                      ;translate
          WSUB         2,2
          ?RETURN
    ;result buffer now holds 'This is a test beep end'
    ;ac3 points past end of text in result
    ;ac2 points to the first comma in str1
    ;DATA
SOURCE:   .TXT         'This,is,a test,$&{}:beep"?!@,.+;\:end'
          .BLK         100.
   DEST:  .BLK         100.
TBLPTR: TBL*2
    .TXTN 1
TBL:      .TXT         '            '
          .TXT         '            '
          .TXT         '            '
          .TXT         '0123456789  '
          .TXT         ' ABCDEFGHIJKLMNO'
          .TXT         'PQRSTUVWXYZ '
          .TXT         ' abcdefghijklmno'
          .TXT         'pqrstuvwxyz '
          .TXT         '            '
          .TXT         '            '
          .TXT         '            '
          .TXT         '0123456789  '
          .TXT         ' ABCDEFGHIJKLMNO'
          .TXT         'PQRSTUVWXYZ '
          .TXT         ' abcdefghijklmno'
          .TXT         'pqrstuvwxyz '
          .END         START
```

Wide Decimal Compare WDCMP

WDCMP



Function: ARG1 @(AC2)[dec#] ?=? ARG2 @(AC3)[dec#]

Parameters:
AC0 = ARG1 data type indicator -> unch
AC1 = ARG2 data type indicator -> result
-1 (ARG1 < ARG2)
0 (ARG1 = ARG2)
+1 (ARG1 > ARG2)
AC2 = ARG1 byte pointer -> unch
AC3 = ARG2 byte pointer -> unch
CRY = ? -> 0

NOTE: Only data types 0, 1, 2, 3, 4, and 5 are valid.

WDCMP compares two decimal strings of types 0 through 5 to determine which string is larger.

Arguments

None

Registers, Flags, and Stacks

AC0 Contains data-type indicator describing decimal string Arg1.

WDCMP does not use the scale factor in the data type indicator.

After execution, contents unchanged.

AC1 Contains data-type indicator describing decimal string Arg2.

WDCMP does not use the scale factor in the data type indicator.

After execution, AC1 holds return code as follows:

AC1 = -1 (Arg1 less than Arg2)
AC1 = 0 (Arg1 equal to Arg2)
AC1 = 1 (Arg1 greater than Arg2)

AC2 Contains byte pointer to high-order byte of decimal string Arg1 in memory.

After execution, contents unchanged.

AC3 Contains byte pointer to high-order byte of decimal string Arg2 in memory.

After execution, contents unchanged.

CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load with immediate
: Use these instructions to place values into AC0 and AC1.

Load effective byte address
: Use these instructions to place byte addresses into AC2 and AC3.

Exceptions

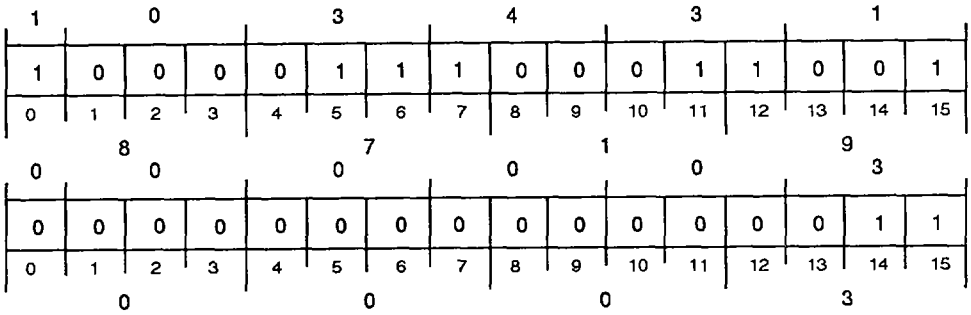
Any digit position not actually present in either argument but needing to be considered in the comparison is treated as zero.

Example

```
XWLDA 0,DESC1      ;AC0 contains the descriptor for ARG1.
XWLDA 1,DESC2      ;AC1 contains the descriptor for ARG2.
XLEF  2,ARG1       ;Word pointer to ARG1 integer field.
XLEF  3,ARG2       ;Word pointer to ARG2 integer field.
WADD  2,2          ;AC2 is a byte pointer to the ARG1 field.
WADD  3,3          ;AC3 is a byte pointer to the ARG2 field.
WDCMP              ;Compare ARG1 to ARG2, and put a code
                   ;into AC1 to reflect the comparison.
```

Wide Decimal Decrement WDDEC

WDDEC



Function: @(AC3)[dec#] - 1 --> @(AC3)[dec#]

Parameters: AC1 = data type indicator --> unch
AC3 = byte pointer to dec# --> unch

NOTE: Only data types 0, 1, 2, 3, 4, and 5 are valid.
If dec# size is not large enough to hold result of decrement, 1 -> CRY, else 0 -> CRY.
If the result is - and data type indicator = 4 (unsigned), then negative sign is ignored.

WDDEC subtracts one from a decimal string of type 0 through 5.

Arguments

None

Registers, Flags, and Stacks

AC0	Unused
AC1	Contains data-type indicator describing integer. After execution, contents unchanged.
AC2	Unused
AC3	Contains byte pointer to high-order byte of decimal string in memory. After execution, contents unchanged.
CARRY	Set to 1 if decrement overflows decimal string; otherwise 0.
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load effective byte address
Use these instructions to place a byte address into AC3.

Load with immediate
Use these instructions to place a value into AC1.

WDINC Wide Decimal Increment

Exceptions

If the decrement overflows the decimal string, CARRY is set to 1, the lower-order result is stored, and the high-order result is ignored.

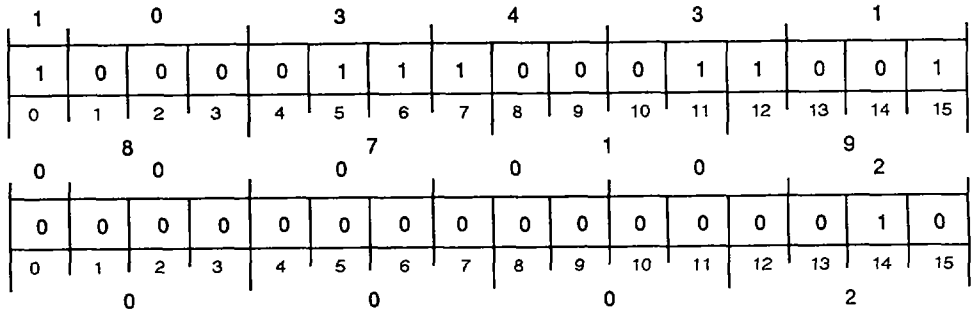
If the result is negative and the data-type is 4 (unsigned), any negative sign of the result is ignored, and the absolute value of the result is stored.

Example

```
XNLDA  1,DESC      ;AC1 contains the data descriptor.
XLEF   3,DATA      ;Word pointer to the integer field.
WADD   3,3         ;AC3 is a byte pointer to the integer.
WDDEC              ;Decrement the commercial integer.
```

Wide Decimal Increment WDINC

WDINC



Function: @(AC3)[dec#] - 1 -> @(AC3)[dec#]

Parameters: AC1 = data type indicator -> unch
AC3 = byte pointer to dec# -> unch

NOTE: Only data types 0, 1, 2, 3, 4, and 5 are valid.
If dec# size is not large enough to hold result of increment, 1 -> CRY, else 0 -> CRY.

WDINC adds one to a decimal string of type 0 through 5.

Arguments

None

Registers, Flags, and Stacks

AC0	Unused
AC1	Before execution, contains data-type indicator describing integer. **WDINC** does not use the scale factor in the data type indicator. After execution, contents unchanged.
AC2	Unused
AC3	Before execution, contains byte pointer to high-order byte of decimal string in memory. After execution, contents unchanged.
CARRY	Set to 1 if increment overflows decimal string; otherwise 0.
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load effective byte address
Use these instructions to place a byte address into AC3.

Load with immediate
Use these instructions to place a value into AC1.

WDDEC Wide Decimal Decrement

Exceptions

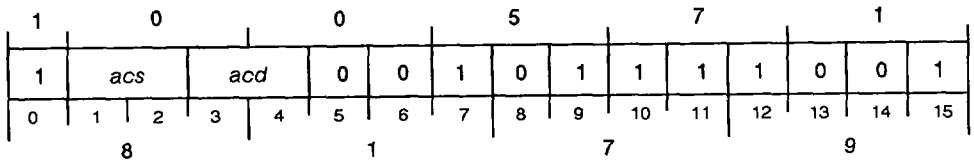
If the increment overflows the decimal string, CARRY is set to 1, the lower-order result is stored, and the high order result is ignored.

Example

```
XNLDA  1,DESC       ;AC1 contains the data descriptor.
XLEF   3,DATA       ;Word pointer to the integer field.
WADD   3,3          ;AC3 is a byte pointer to the integer.
WDINC               ;Increment the commercial integer.
```

Wide Divide WDIV

WDIV *acs,acd*



Function: *acd* / *acs* -> *acd*(quotient)

Parameters: None

NOTE: If *acs* = 0, or result overflows; overflow = 1 and *acd* = unch

WDIV sign-extends the signed 32-bit integer contained in *acd* to 64 bits and divides this value by the signed 32-bit integer contained in *acs*.

Arguments

acs	Before execution, contains signed 32-bit dividend.
	After execution, contents unchanged.
acd	Before execution, contains signed 32-bit divisor. **WDIV** sign-extends this to 64 bits.
	After execution, contains signed 32-bit result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	1 if result is not within specified range or if *acs* is 0; otherwise 0.
PC	PC + 1
PSR	OVR is set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

DIV	Unsigned Divide
DIVS	Signed Divide
DIVX	Sign Extend and Divide
NDIV	Narrow Divide
WDIVS	Wide Signed Divide

Exceptions

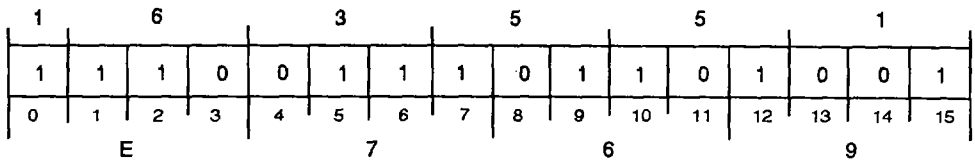
If quotient is outside the range, -2,147,483,648 to +2,147,483,647, or if *acs* contains zero, an *overflow* occurs. OVR is set to 1, and *acd* is unchanged.

Example

```
XWLDA 2,DIVIDEND   ;Get the dividend.
XWLDA 3,DIVISOR    ;Get the divisor.
WDIV  3,2          ;Divide.
XWSTA 2,RESULT     ;Store the result.
```

Wide Signed Divide WDIVS

WDIVS



Function: AC0&AC1 / AC2 -> AC1(quotient)&AC0(remainder)

Parameters: AC0 = high-order dividend -> remainder
AC1 = low-order dividend -> quotient
AC2 = divisor -> unchanged

NOTE: If AC2 = 0, or result overflows; overflow = 1 and AC0 & AC1 = unchanged

WDIVS divides a signed 64-bit integer contained in AC0 and AC1 by a signed 32-bit integer contained in AC2.

Zero remainders are always positive. All other remainders have the same sign as the dividend.

Arguments

None

Registers, Flags, and Stacks

AC0	Before execution, contains high-order 32 bits of signed 64-bit dividend.
	After execution, contains signed 32-bit remainder.
AC1	Before execution, contains low-order 32 bits of signed 64-bit divident.
	After execution, contains signed 32-bit quotient.
AC2	Before execution, contains signed 32-bit divisor.
	After execution, contents unchanged.
AC3	Unused
CARRY	Unchanged
Overflow	1 if result is not within specified range or AC2 is 0; otherwise 0.
PC	PC + 1
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

DIV	Unsigned Divide
DIVS	Signed Divide
DIVX	Sign Extend and Divide
NDIV	Narrow Divide
WDIV	Wide Divide

Exceptions

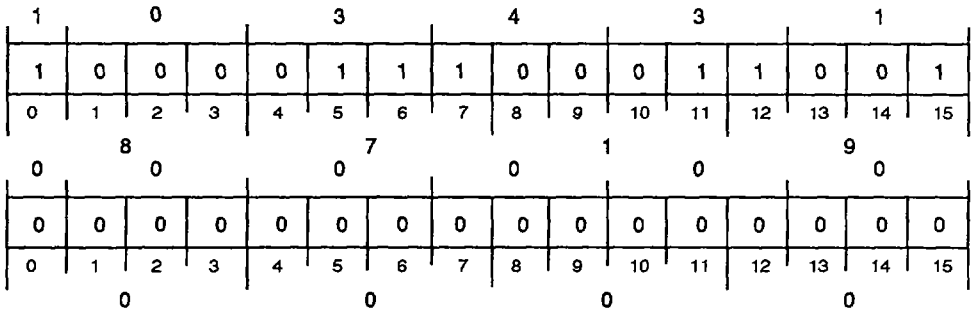
If quotient is outside the range, -2,147,483,648 to +2,147,483,647, or if AC2 contains 0, an overflow occurs. PSR(OVR) is set to 1, and AC0 and AC1 are unchanged.

Example

```
XWLDA  0,DIVIDEND_HIGH      ;Get the dividend high order 32 bits.
XWLDA  1,DIVIDEND_LOW       ;Get the dividend low order 32 bits.
XWLDA  2,DIVISOR            ;Get the divisor.
WDIVS                       ;Divide.
XWSTA  1,QUOTIENT           ;Store the quotient.
XWSTA  0,REMAINDER          ;Store the remainder.
```

Wide Decimal Move WDMOV

WDMOV



Function: @(AC2)[scaled decimal #] -> @(AC3)[scaled decimal #]

Parameters: AC0 = Source data type indicator -> unchanged
AC1 = Destination data type indicator -> unchanged
AC2 = Source byte pointer -> unchanged
AC3 = Destination byte pointer -> unchanged

NOTE: Only data types 0, 1, 2, 3, 4, and 5 are valid.
If Source is too large for Destination, 1 -> CRY.
If Source is - and Destination = data type 4 (unsigned), then Source sign is ignored.

WDMOV moves and scales the source integer decimal string into the destination integer decimal string. Source and destination strings must be of data types 0 through 5. Data types for the source and destination strings may differ, allowing conversion from one commercial format to another.

Any digits of the source string in positions of lesser significance than the destination string can represent are ignored.

The integer placed in destination is zero-extended if necessary to fill the decimal string.

Arguments

None

Registers, Flags, and Stacks

AC0 Before execution, contains data-type indicator describing source.

After execution, contents unchanged.

AC1 Before execution, contains data-type indicator describing destination.

After execution, contents unchanged.

AC2 Before execution, contains byte pointer to high-order byte of source.

After execution, contents unchanged.

AC3 Before execution, contains byte pointer to high-order byte of destination.

After execution, contents unchanged.

CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load with immediate
: Use these instructions to place values into AC0 and AC1.

Load effective byte address
: Use these instructions to place byte addresses into AC2 and AC3.

Exceptions

If the source string has one or more nonzero digits in positions of greater significance then the destination string can represent, the destination string will receive whatever digits of the source string it is capable of representing. CARRY is then set to 1.

If source is negative zero, destination is not changed to positive zero. (The sign of destination is given the sign of source.)

If source is negative and destination is data type 4 (unsigned), the sign of source is ignored, and the absolute value of source is moved to destination.

If the source and destination strings overlap in memory in any way, the results are undefined.

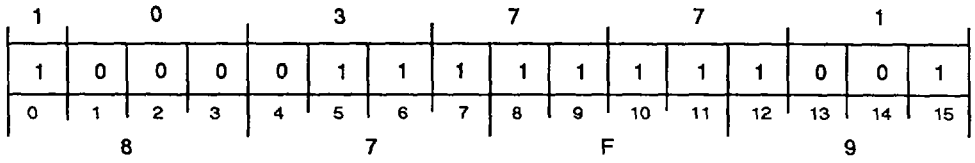
Example

```
XWLDA 0,DESC1      ;AC0 is the data descriptor for ARG1.
XWLDA 1,DESC2      ;AC1 is the data descriptor for ARG2.
XLEF  2,ARG1       ;Word pointer to the ARG1 integer field.
XLEF  3,ARG2       ;Word pointer to the ARG2 integer field.
WADD  2,2          ;AC2 is a byte pointer to ARG1.
WADD  3,3          ;AC3 is a byte pointer to ARG2.
WDMOV              ;Scale and move ARG1 to ARG2.
```

Pop Context Block WDPOP

Privileged Instruction

WDPOP



Function: Return from Page Fault Restores CPU state
(32-33)page zero -> context block

Parameters: None

NOTE: **WDPOP** is implementation-specific.

WDPOP uses the information pointed to by the context block pointer (locations 32_8–33_8 in page zero of segment 0) to restore the CPU to its state at the time of the page fault. Execution of the interrupted program resumes before, during, or after the instruction that caused the fault, depending on the instruction type and how far it had proceeded before the fault.

Arguments

None

Registers, Flags, and Stacks

AC0–AC3	Loaded from context block.
CARRY	Loaded from context block.
Overflow	0
PC	Loaded from context block.
PSR	Loaded from context block.
Stack	Unchanged

Related Instructions

None

Exceptions

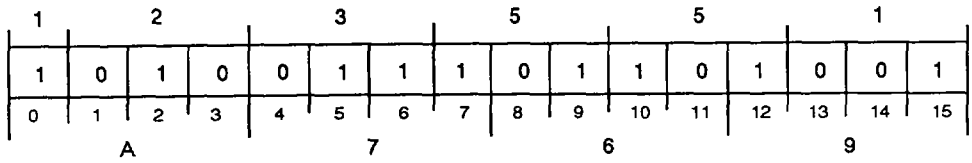
None

Example

```
WDPOP    ;Restore the state of the processor,
         ;including PC and ACs to what it was at the
         ;time of the last restartable fault.
```

Wide Edit WEDIT

WEDIT



Function: Enter wide edit subprogram

Parameters:
AC0 = byte pointer (1st subopcode) --> P
AC1 = data-type indicator --> ?
AC2 = byte pointer (destination) --> DI
AC3 = byte pointer (source) --> SI
T = 0 --> ?
S = SI sign (0 = +, 1 = -) --> ?
SI = AC3 --> last byte pointer + 1
DI = AC2 --> last byte pointer + 1
P = AC0 --> last byte pointer + 1
CRY = x --> T

NOTE:
For subopcodes: *j* = # characters
If *j*(high-order bit) = 1, word at (WSP + 2 + 2**j*)
During execution, a subprogram modifies DI, P, S, SI, and T; can test S and T flags.

WEDIT provides entry and control for an edit subprogram. The subprogram converts a decimal number from either packed or unpacked form to a string of bytes. This subprogram can perform many different operations on the number and its destination field, including leading zero suppression, leading or trailing signs, floating fill characters, punctuation control, and insertion of text into the destination field. Subprogram instructions also perform operations on alphanumeric data.

WEDIT uses the contents of the four accumulators to provide initial parameters for the subprogram, and maintains two flags and three indicators or pointers, which can be tested and modified by the subprogram. The flags are the significance Trigger (T) and the Sign flags (S); the three pointers are the Source Indicator (SI), the Destination Indicator (DI), and the opcode Pointer (P).

The significance Trigger flag is set to 1 when the first nonzero digit is processed; the Sign flag is set to reflect the sign of the source integer being processed. Some subprogram instructions may explicitly set or clear the T and S flags.

The three pointers are 32-bit byte pointers to the current byte in each respective area. These fields may overlap in any way. The instruction, however, processes characters one at a time, so unusual side effects may be produced by certain types of overlap.

The subprogram is made up of eight-bit opcodes (subopcodes) followed by one or more eight-bit operands. P, a byte pointer, serves as the program counter for the Wedit subprogram. The subprogram proceeds sequentially until a branching operation occurs -- much the same way programs are processed. Unless instructed to do otherwise, the Wedit instruction updates P after each operation to point to the next sequential subopcode. The instruction continues to process eight-bit opcodes until directed to stop by the **DEND** subopcode.

The effective addresses generated by **WEDIT**, and the subprogram itself, are confined to the current segment.

In the description of some of the Edit subopcodes, the symbol *j* denotes how many characters a certain operation should process. When the high order bit of *j* is 1, *j* has a different meaning: *j* is interpreted as an eight-bit, two's complement number, and the number of characters to process is equal to the value of the word at the address *wide stack pointer + 2 + 2*j*.

The Edit operations which process numeric data (**DMVF**, **DMVN**, **DMVO**, and **DMVS**) use the following algorithm to access each source digit:

1. If SI has ever moved outside the source area, a zero will be used for the source digit, and SI will not be affected. Note that zeros will be supplied for all future source digits, even if SI is moved back inside the source area.

2. If the source integer is data type 3, and SI currently points to the sign of the integer, SI will be incremented to skip over the sign.

3. The digit to which SI currently points is checked for validity, and the binary coded decimal (BCD) value of the digit is used. SI is incremented to point to the next digit in the source integer.

4. If the source integer is data type 2, and the last digit has been read, SI is incremented beyond the trailing sign byte.

Arguments

None

Registers, Flags, and Stacks

AC0	Initially contains byte pointer to first subopcode of the Wedit subprogram. After successful execution of subprogram, contains P, which points to byte following **DEND** subopcode.
AC1	Initially contains data-type indicator describing source integer being processed. The scale factor portion is not used. For further information, refer to "Decimal and Byte Operations" section of "Fixed-Point Computing" chapter. After successful execution of subprogram, contents undefined.
AC2	Initially contains byte pointer to first byte of destination byte field (DI). After successful execution of subprogram, contains byte pointer (DI) to next byte in destination field.
AC3	Initially contains byte pointer to first byte of source integer field (SI). After successful execution of subprogram, contains byte pointer (SI) to next source byte.
CARRY	After execution, contains T.
DI	Initially, contains AC2. After execution of subopcode, may be modified.
Overflow	0

P	Initially, contains AC0. Unless instructed otherwise, updated after each subopcode to point to next sequential subopcode.
PSR	If IRES contains 1, instruction assumes restart from interrupt. Do not set IRES under any other circumstances.
PC	PC + 1 (After successful completion of subprogram)
S	Initially, reflects sign of source integer being processed. 0 = positive; 1 = negative. May be explicitly set or cleared by some subopcodes.
SI	Initially, contains AC3. After execution of subopcode, may be modified.
Stack	Initially, wide stack should have at least nine double words available for use by **WEDIT**, plus six additional double words for interrupt use.
T	Initially, set to 0. Set to 1 when first non-zero digit processed. May be explicitly set or cleared by some subopcodes.

Related Instructions

EDIT	Edit
Edit subopcodes	Use these instructions to manipulate and process data as a subprogram.
LLEFB, XLEFB	Load Effective Byte address instructions

Exceptions

If the sign of the source integer is invalid, a commercial fault is inititated, and **WEDIT** terminates.

If the data type indicator in AC1 specifies that the source integer is data type 6 or 7, a commercial fault is initiated, and the instruction terminates.

WEDIT considers the subprogram as data and does not check for execute protection.

See also the exceptions for the individual subopcodes.

If **WEDIT** is interrupted, restart information is placed on the wide stack and IRES is set to 1.

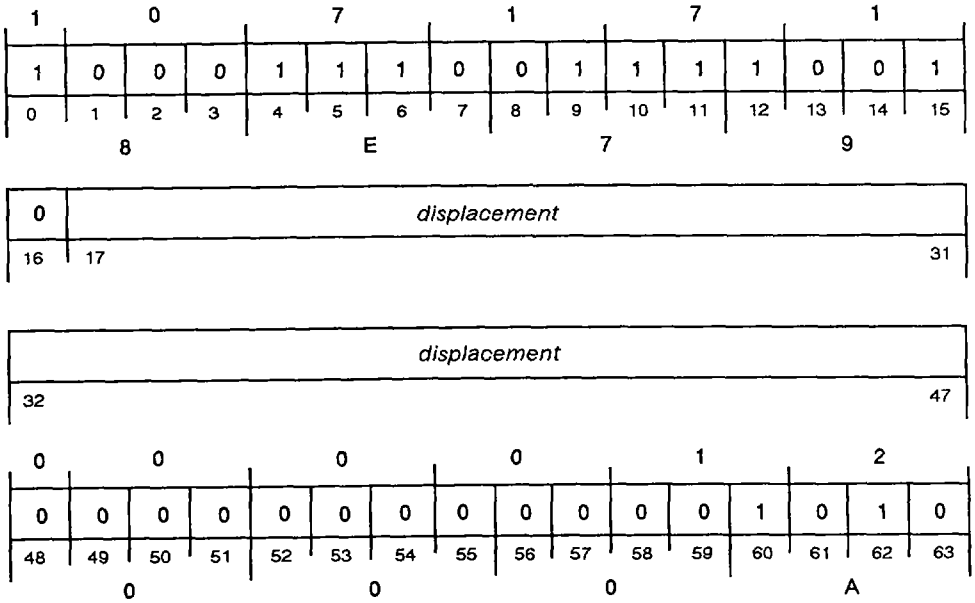
Example

```
    .TITLEW
    .ENT START,SRCSTRNG,DESTRING,EDITSUB
    ;User symbols defined in this source module may be referenced by other
    ;modules or the debugger.
    ;
    .NREL
START:    XWLDA       0,PROGPTR    ;Load the 32-bit contents of the
                                   ;edit subprogram into AC0.
          XWLDA       1,TYPE       ;Load the data type into AC1.
          XWLDA       2,DESTPTR    ;Load the 32-bit byte pointer to the
                                   ;destination string.
          XWLDA       3,SRCPTR     ;Byte pointer to the destination
                                   ;string.
                                   ;Byte pointer to the source string.
          WEDIT                    ;Call EDITSUB, the edit subprogram,
                                   ;which begins with the string ABCDEFGH
                                   ;(in SRCSTRNG), moves 2 characters
                                   ;(AB), inserts 1 (x), moves CD,
                                   ;inserts xxx, moves EFG, inserts x,
                                   ;and moves H, so that the string
                                   ;ABCDEFGH becomes ABxCDxxxEFGxH.
   WSUB   2,2          ;Place zeros in AC2 to flag a normal return.
EXIT:     ?RETURN      ;Terminate the calling process and return
                       ;control to the CLI.
          WBR EXIT     ;<If return fails, retry>
                       ;
                       ;
EDITSUB:               ;The edit subprogram starts here.
          .TXT         "
              <DMVC><2>
          <DICI><1>x
          <DMVC><2>
          <DICI><3>xxx
          <DMVC><3>
          <DICI><1>x
          <DMVC><1>
          <DEND>"
PROGPTR:  .DWORD       EDITSUB*2    ;A byte pointer to the first opcode of
                                   ;the WEDIT subprogram, in 2 words.
SRCPTR:   .DWORD       SRCSTRNG*2   ;A byte pointer to the data or text
                                   ;string, in 2 words.
DESTPTR:  .DWORD       DESTRING*2   ;A byte pointer to the first byte of
                                   ;the destination field, in 2 words.
TYPE:     .DWORD       4            ;Store the value 4 in a single word.
SRCSTRNG: .TXT "ABCDEFGH"
DESTRING: .BLK 7.                  ;Reserve 14 bytes for result.
          .END         START
```

Floating-Point Arccosine Double WFACOSD

Intrinsic Instruction

WFACOSD *displacement*



Function: arccosine FPAC0 -> FPAC0

Parameters: FPAC0 = floating-point # -> arccosine[radians]
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: Result is in radians.
If absolute value (FPAC0) > 1, then 1 -> FPSR(3), code 3 -> FPSR(28-31)

WFACOSD computes the arccosine of the 64-bit floating-point value in FPAC0 and places the result (in radians) in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFACOSD** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0 Before execution, contains 64-bit floating-point value.
After execution, contains result.

FPAC1-FPAC3 Unused.
After execution, contents undefined.

FPSR Updated Z and N flags.

PC PC + 4

Stack Unchanged

Related Instructions

WFACOSS Floating-Point Arccosine Single

Exceptions

If the absolute value of the input number in FPAC0 is greater than one, the processor sets the INV bit in the FPSR to one and returns error code 3 to the INP bits of the FPSR.

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the arccosine calculation. The WPOPJ instruction exits this software emulator.

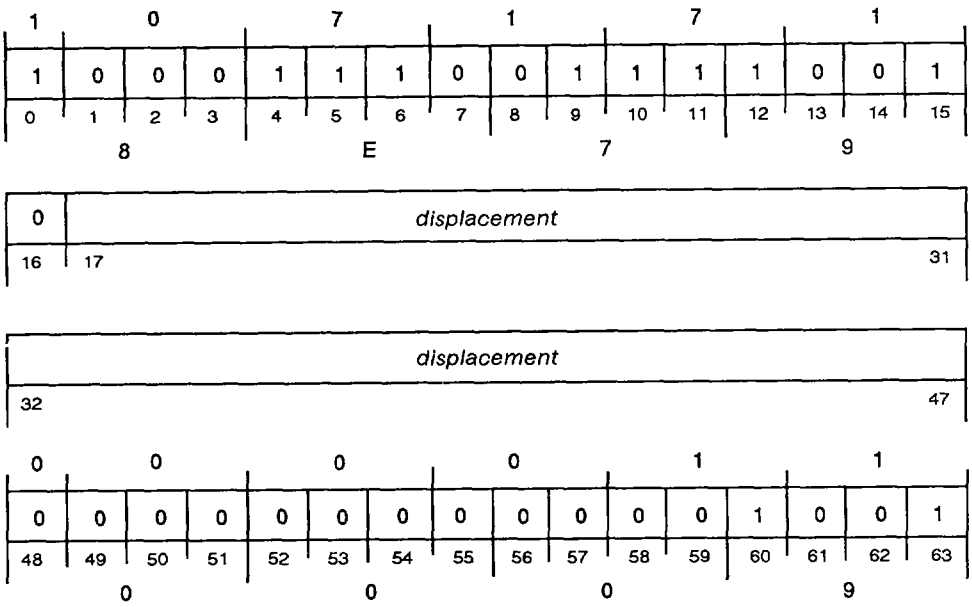
Example

```
LFLDD 0,FLOATX      ;Calculate the double precision arccosine
WFACOSD ACOSD       ;of the floating point number at location
LFSTD 0,ACOSDX      ;FLOATX, and store the result at location
                    ;ACOSDX. ACOSD is a routine that is called
                    ;if IIS is not available.
```

Floating-Point Arccosine Single — WFACOSS

Intrinsic Instruction

WFACOSS *displacement*



Function: arccosine FPAC0 -> FPAC0

Parameters: FPAC0 = floating-point # -> arccosine[radians]
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: Result is in radians.
If absolute value (FPAC0) > 1, then 1 -> FPSR(3), code 3 -> FPSR(28-31).

WFACOSS computes the arccosine of the 32-bit floating-point value in FPAC0, and places the result (in radians) in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFACOSS** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0(0-31) Before execution, contains 32-bit floating-point value.
After execution, contains 32-bit result with bits 32-63 set to 0.

FPAC1-FPAC3 Unused.
After execution, contents undefined.

FPSR Updated Z and N flags.

PC PC + 4

Stack Unchanged

Related Instructions

WFACOSD Floating-Point Arccosine Double

Exceptions

If the absolute value of the input number in FPAC0 is greater than one, the processor sets the INV bit in the FPSR to one and returns error code 3 to the INP bits of the FPSR.

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the arccosine calculation. The **WPOPJ** instruction exits this software emulator.

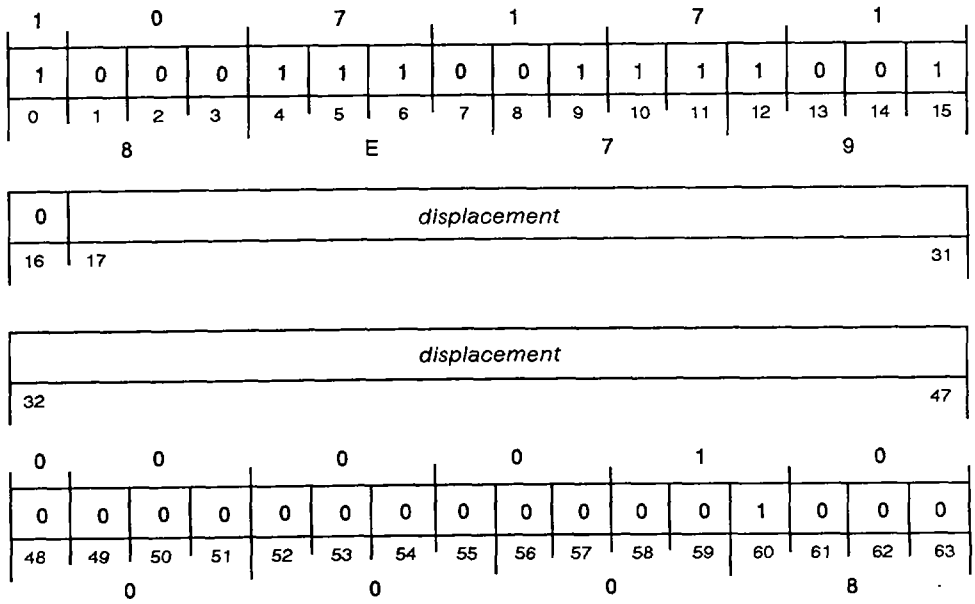
Example

```
FMOV  1,0          ;Calculate the single precision arccosine
WFACOSS ACOSS      ;of the value in FPAC1, and store the result
LFSTS 0,ACOSSX     ;at memory location ACOSSX. ACOSS is a routine
                   ;that is called if IIS is not available.
```

Floating-Point Arcsine Double — WFASIND

Intrinsic Instruction

WFASIND *displacement*



Function: arcsine FPAC0 -> FPAC0

Parameters: FPAC0 = floating-point # -> arcsine[radians]
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: Result is in radians.
If absolute value (FPAC0) > 1, then 1 -> FPSR(3), code 3 -> FPSR(28-31).

WFASIND computes the arcsine of the 64-bit floating-point value in FPAC0 and places the result (in radians) in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFASIND** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0	Before execution, contains 64-bit floating-point value.
	After execution, contains result.
FPAC1-FPAC3	Unused.
	After execution, contents undefined.
FPSR	Updated Z and N flags.
PC	PC + 4
Stack	Unchanged

Related Instructions

WFASINS Floating-Point Arcsine Single

Exceptions

If the absolute value of the input number in FPAC0 is greater than one, the processor sets the INV bit in the FPSR to one and returns error code 3 to the INP bits of the FPSR.

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the arcsine calculation. The **WPOPJ** instruction exits this software emulator.

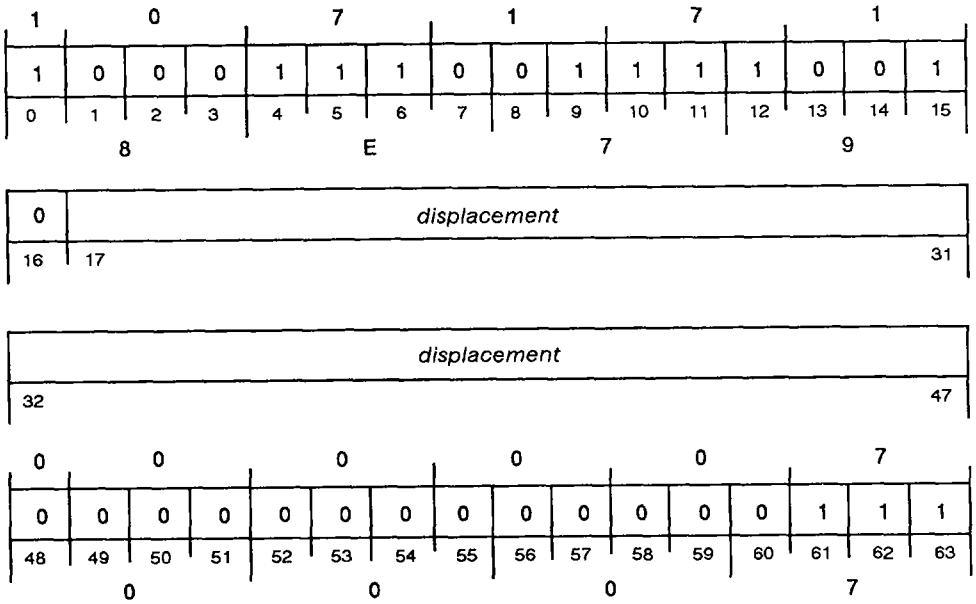
Example

```
LFLDD 0,FLOATX      ;Calculate the double precision arcsine
WFASIND ASIND       ;of the floating point number at memory
LFSTD 0,ASINDX      ;location FLOATX, and store the result at
                    ;location ASINDX. ASIND is a routine that
                    ;is called if IIS is not available.
```

Floating-Point Arcsine Single — WFASINS

Intrinsic Instruction

WFASINS *displacement*



Function: arcsine FPAC0 -> FPAC0

Parameters: FPAC0 = floating-point # -> arcsine[radians]
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: Result is in radians.
If absolute value (FPAC0) > 1, then 1 -> FPSR(3), code 3 -> FPSR(28-31).

WFASINS computes the arcsine of the 32-bit floating-point value in FPAC0 and places the result (in radians) in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFASINS** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0(0-31)	Before execution, contains 32-bit floating-point value.
	After execution, contains 32-bit result with bits 32-63 set to 0.
FPAC1-FPAC3	Unused.
	After execution, contents undefined.
FPSR	Updated Z and N flags.
PC	PC + 4
Stack	Unchanged

Related Instructions

WFASIND Floating-Point Arcsine Double

Exceptions

If the absolute value of the input number in FPAC0 is greater than one, the processor sets the INV bit in the FPSR to one and returns error code 3 to the INP bits of the FPSR.

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the arcsine calculation. The **WPOPJ** instruction exits this software emulator.

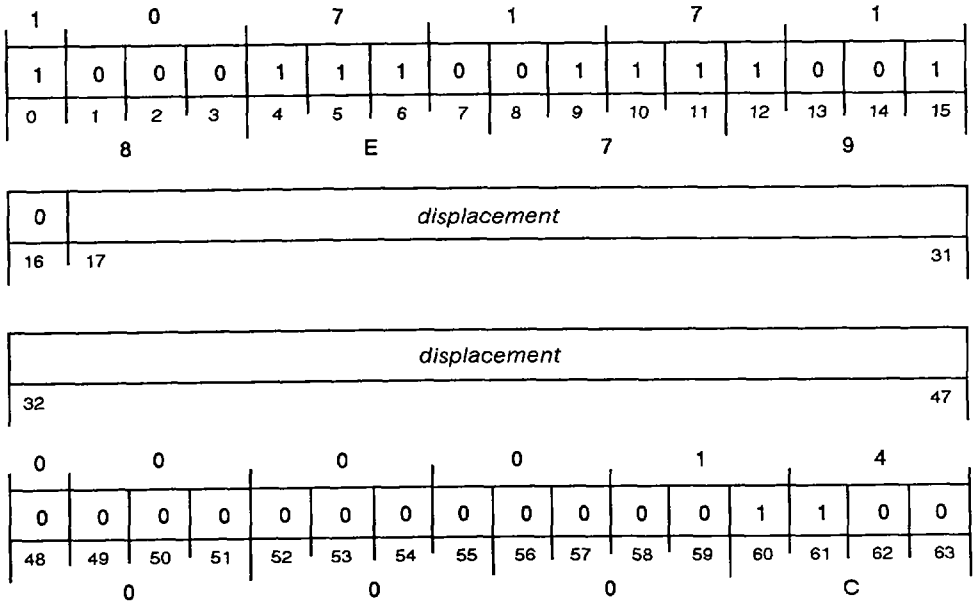
Example

```
FMOV  1,0          ;Calculate the single precision arcsine
WFASINS ASINS      ;of the value in FPAC1, and store the result
LFSTS 0,ASINSX     ;at memory location ASINSX. ASINS is a
                   ;routine that is called if IIS is not
                   ;available.
```

Floating-Point Arctangent Double WFATAND

Intrinsic Instruction

WFATAND *displacement*



Function: arctangent FPAC0 -> FPAC0

Parameters: FPAC0 = floating-point # -> arctangent [radians]
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: Result is in radians.

WFATAND computes the arctangent of the 64-bit floating-point value in FPAC0 and places the result (in radians) in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFATAND** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0	Before execution, contains 64-bit floating-point value.
	After execution, contains result.
FPAC1-FPAC3	Unused.
	After execution, contents undefined.
FPSR	Updated Z and N flags.
PC	PC + 4
Stack	Unchanged

Related Instructions

WFATANS Floating-Point Arctangent Single

Exceptions

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the arctangent calculation. The **WPOPJ** instruction exits this software emulator.

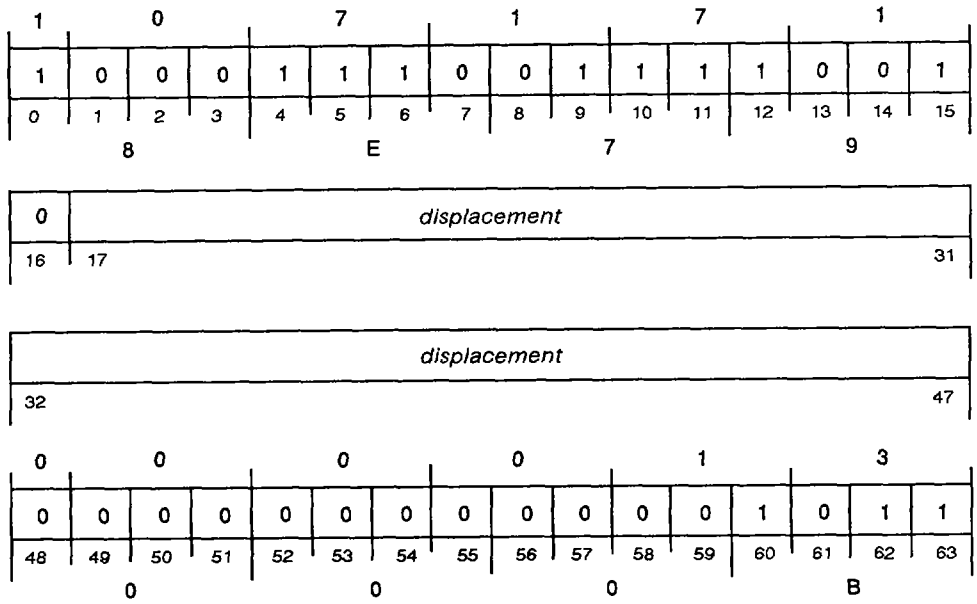
Example

```
LFLDD 0,FLOATX      ;Calculate the double precision arctangent
WFATAND ATAND       ;of the floating point number at memory
LFSTD 0,ATANDX      ;location FLOATX, and store the result at
                    ;location ATANDX. ATAND is a routine that
                    ;is called if IIS is not available.
```

Floating-Point Arctangent Single WFATANS

Intrinsic Instruction

WFATANS *displacement*



Function: arctangent FPAC0 -> FPAC0

Parameters: FPAC0 = floating-point # -> arctangent[radians]
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: Result is in radians.

WFATANS computes the arctangent of the 32-bit floating-point value in FPAC0 and places the result (in radians) in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFATANS** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0(0-31) Before execution, contains 32-bit floating-point value.

After execution, contains 32-bit result with bits 32-63 set to 0.

FPAC1-FPAC3 Unused. After execution, contents undefined.

FPSR Updated Z and N flags.

PC PC + 4

Stack Unchanged

Related Instructions

WFATAND Floating-Point Arctangent Double

Exceptions

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the arctangent calculation. The **WPOPJ** instruction exits this software emulator.

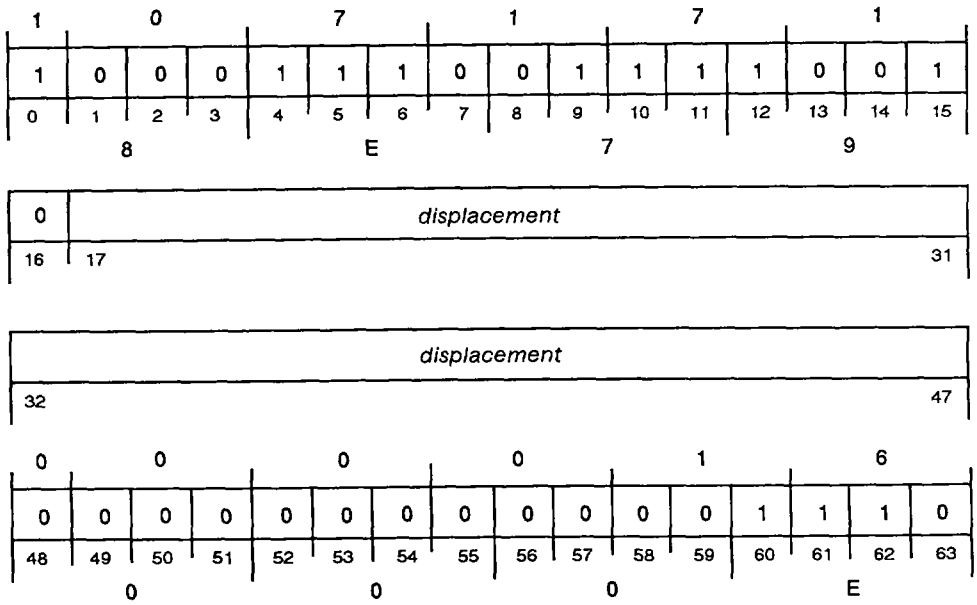
Example

```
FMOV   2,0          ;Calculate the single precision arctangent
WFATANS ATANS       ;of the value in FPAC2, and store the result
LFSTS  0,ATANSX     ;at memory location ATANSX. ATANS is a routine
                    ;that is called if IIS is not available.
```

Floating-Point Arctangent Double(two-accumulator) WFATN2D

Intrinsic Instruction

WFATN2D *displacement*



Function: arctangent FPAC0/FPAC1 -> FPAC0

Parameters: FPAC0 = X(floating-point #) -> arctangent[radians]
FPAC1 = X(floating-point #) -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: Result is in radians.
If FPAC1 = 0, then 1 -> FPSR(3), code 7 -> FPSR(28-31)

WFATN2D computes the arctangent of the quotient of two 64-bit floating-point values (FPAC0 divided by FPAC1), and places the result with correct quadrature (in radians) in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFATN2D** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0	Before execution, contains 64-bit floating-point value.
	After execution, contains result.
FPAC1	Before execution, contains 64-bit floating-point value.
	After execution, undefined.
FPAC2-FPAC3	Unused. After execution, contents undefined.
FPSR	Updated Z and N flags.
PC	PC + 4
Stack	Unchanged

Related Instructions

WFATN2S Floating-Point Arctangent Single (two-accumulator)

Exceptions

If the value in FPAC1 is 0, the processor sets the INV bit in the FPSR to one and returns error code 7 to the INP bits of the FPSR.

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction function and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the arctangent function. The **WPOPJ** instruction exits this software emulator.

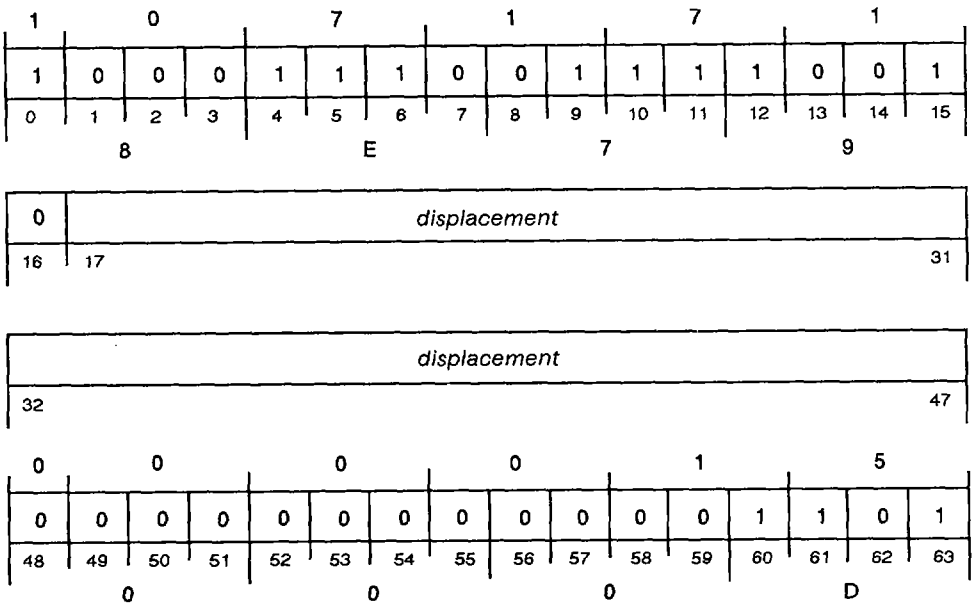
Example

```
LFLDD 0,FLOATY      ;Calculate the radian measure of the
LFLDD 1,FLOATX      ;angle at the point (FLOATX, FLOATY),
WFATN2D ATN2D       ;and store the result at location ATN2DX.
LFSTD 0,ATN2DX      ;ATN2D is a routine that is called if
                    ;IIS is not available.
```

Floating-Point Arctangent Single (two-accumulator) WFATN2S

Intrinsic Instruction

WFATN2S *displacement*



Function: arctangent FPAC0/FPAC1 -> FPAC0

Parameters: FPAC0 = X(floating-point #) -> arctangent[radians]
FPAC1 = X(floating-point #) -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: Result is in radians.
If FPAC1 = 0, then 1 -> FPSR(3), code 7 -> FPSR(28-31).

WFATN2S computes the arctangent of the quotient of two 32-bit floating-point values (FPAC0 divided by FPAC1) and places the result with correct quadrature (in radians) in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFATN2S** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0(0-31)	Before execution, contains 32-bit floating-point value.
	After execution, contains 32-bit result with bits 32-63 set to 0.
FPAC1(0-31)	Before execution, contains 32-bit floating-point value.
	After execution, undefined.
FPAC2-FPAC3	Unused. After execution, contents undefined.
FPSR	Updated Z and N flags.
PC	PC + 4
Stack	Unchanged

Related Instructions

WFATN2D Floating-Point Arctangent Double (two-accumulator)

Exceptions

If the value in FPAC1 is 0, the processor sets the INV bit in the FPSR to one and returns error code 7 to the INP bits of the FPSR.

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the arctangent calculation. The **WPOPJ** instruction exits this software emulator.

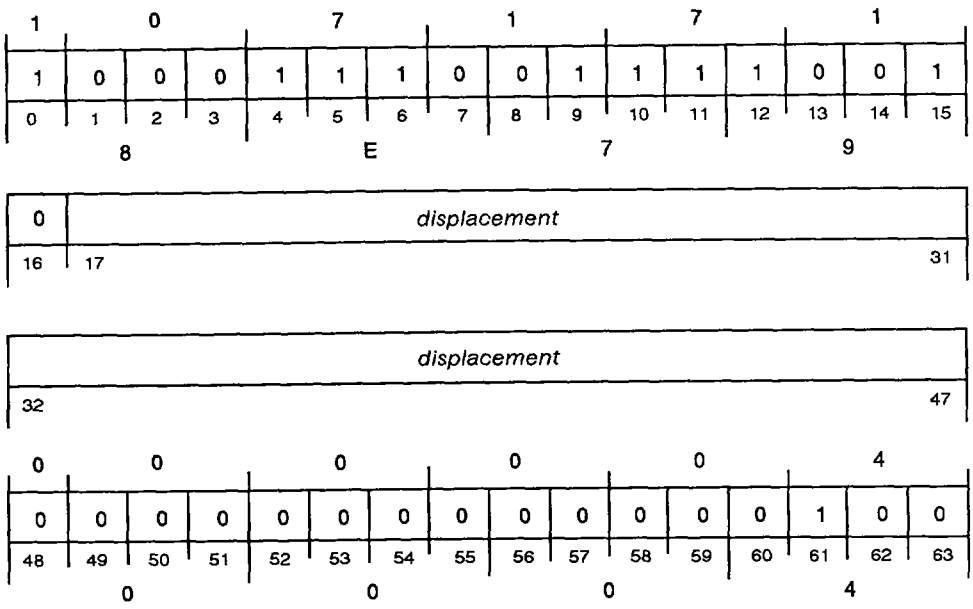
Example

```
LFLDS 0,Y          ;Calculate the radian measure of the
LFLDS 1,X          ;angle at the point (X, Y), and store
WFATN2S ATN2S      ;the result at location ANGLE. ATN2S is a
LFSTS 0,ANGLE      ;routine that is called if IIS is
                   ;not available.
```

Floating-Point Cosine Double — WFCOSD

Intrinsic Instruction

WFCOSD *displacement*



Function: cosine FPAC0 -> FPAC0

Parameters: FPAC0 = floating-point #[radians] -> cosine
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: Input is in radians.

WFCOSD computes the cosine of the 64-bit floating-point value (in radians) in FPAC0, and places the result in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFCOSD** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0 Before execution, contains 64-bit floating-point value (in radians).

After execution, contains 64-bit result.

FPAC1-FPAC3 Unused.

After execution, contents undefined.

FPSR Updated Z and N flags.

PC PC + 4

Stack Unchanged

Related Instructions

WFCOSS Floating-Point Cosine Single

Exceptions

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the cosine calculation. The **WPOPJ** instruction exits this software emulator.

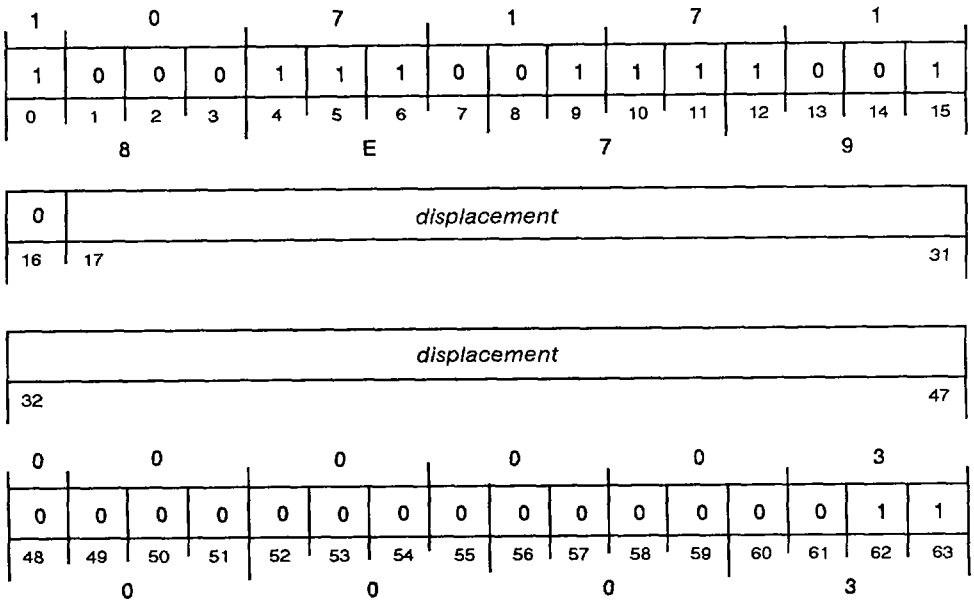
Example

```
LFLDD 0,FLOATX      ;Calculate the double precision cosine
WFCOSD COSD         ;of the floating point number at memory
LFSTD 0,COSDX       ;location FLOATX, and store the result at
                    ;location COSDX. COSD is a routine that
                    ;is called if IIS is not available.
```

Floating-Point Cosine Single — WFCOSS

Intrinsic Instruction

WFCOSS *displacement*



Function: Cosine FPAC0 -> FPAC0

Parameters: FPAC0 = floating-point #[radians] -> cosine
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: Input is in radians.

WFCOSS computes the cosine of the 32-bit floating-point value (in radians) in FPAC0 and places the result in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFCOSS** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0(0-31) Before execution, contains 32-bit floating-point value (in radians).

After execution, contains 32-bit result with bits 32-63 set to 0.

FPAC1-FPAC3 Unused.

After execution, contents undefined.

FPSR Updated Z and N flags.

PC PC + 4

Stack Unchanged

Related Instructions

WFCOSD Floating-Point Cosine Double

Exceptions

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the cosine calculation. The **WPOPJ** instruction exits this software emulator.

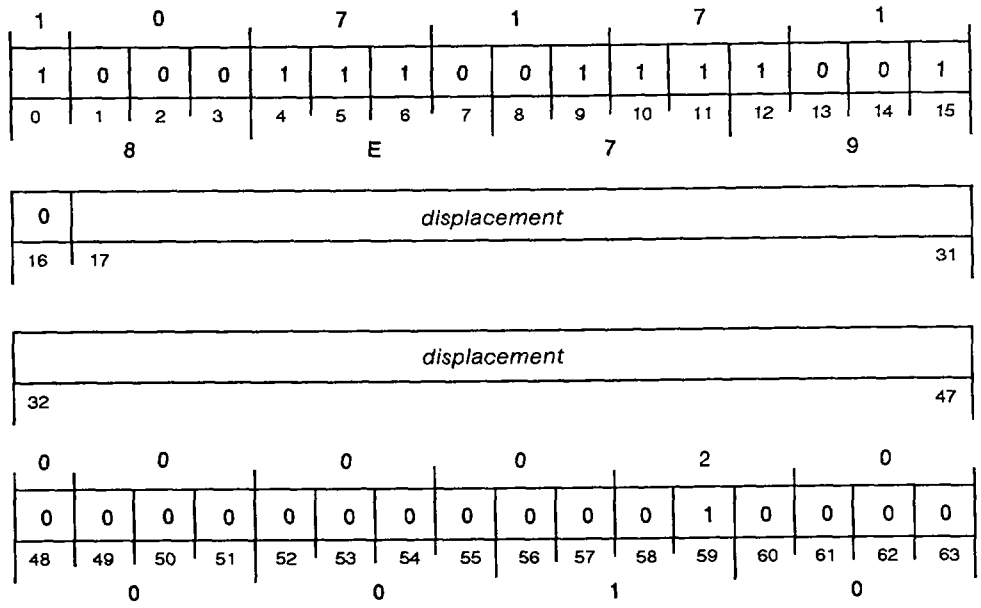
Example

```
FMOV  3,0          ;Calculate the single precision cosine
WFCOSS COSS        ;of the value in FPAC3, and store the result
LFSTS 0,COSSX      ;at memory location COSSX. COSS is a routine
                   ;that is called if IIS is not available.
```

Floating-Point Exponential Double WFEXPD

Intrinsic Instruction

WFEXPD *displacement*



Function: e ** FPAC0 -> FPAC0

Parameters: FPAC0 = floating-point # -> result
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: If result will overflow (approximately log(e) 16**63), then 1 -> FPSR(3), code 5 -> FPSR(28-31).

WFEXPD raises the value e to the 64-bit floating-point power contained in FPAC0 and places the result in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFEXPD** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0 Before execution, contains 64-bit floating-point value.
After execution, contains result.

FPAC1-FPAC3 Unused.
After execution, contents undefined.

FPSR Updated Z and N flags.

PC PC + 4

Stack Unchanged

Related Instructions

WFEXPS Floating-Point Exponential Single

Exceptions

If the input value in FPAC0 produces a result that overflows (approximately the natural log of 16 raised to the 63rd power), the processor sets the INV bit in the FPSR to one and returns error code 5 to the INP bits of the FPSR.

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the exponential calculation. The **WPOPJ** instruction exits this software emulator.

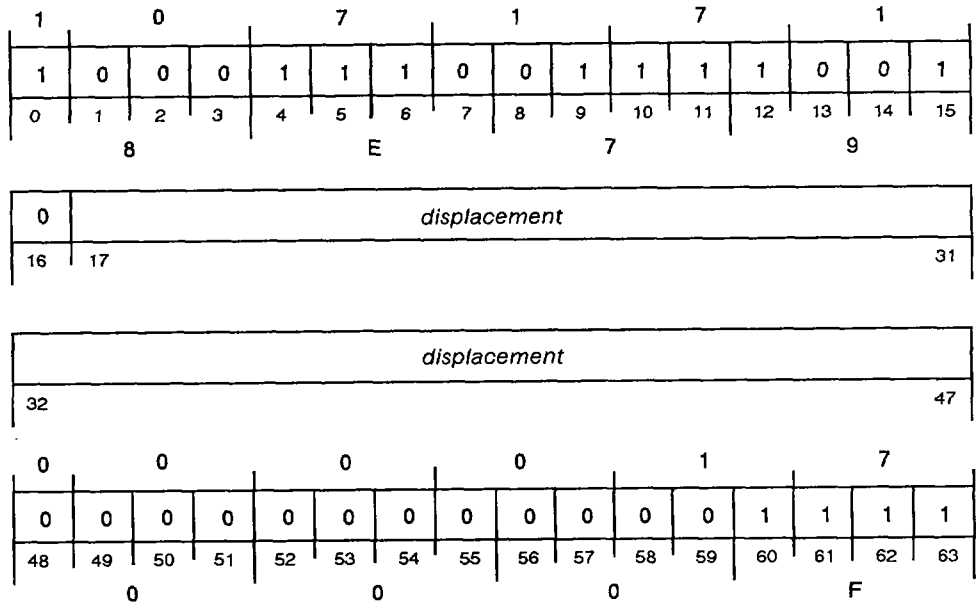
Example

```
LFLDD 0,POWER      ;Calculate the double precision exponential
WFEXPD EXPD        ;of the floating point number at memory
LFSTD 0,EXPX       ;location POWER, and store the result at
                   ;location EXPX. EXPD is a routine that
                   ;is called if IIS is not available.
```

Floating-Point Exponential Single — *WFEXPS

Intrinsic Instruction

WFEXPS *displacement*



Function: e ** FPAC0 -> FPAC0

Parameters: FPAC0 = floating-point # -> result
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: If result will overflow (approximately log(e) 16 ** 63), then 1 -> FPSR(3), code 5 FPSR(28-31).

WFEXPS raises the value e to the 32-bit floating-point power contained in FPAC0 and places the result in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFEXPS** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0(0-31) Before execution, contains 32-bit floating-point value.
After execution, contains 32-bit result with bits 32-63 set to 0.

FPAC1-FPAC3 Unused.
After execution, contents undefined.

FPSR Updated Z and N flags.

PC PC + 4

Stack Unchanged

Related Instructions

WFEXPD Floating-Point Exponential Double

Exceptions

If the input value in FPAC0 produces a result that overflows (approximately the natural log of 16 raised to the 63rd power), the processor sets the INV bit in the FPSR to one and returns error code 5 to the INP bits of the FPSR.

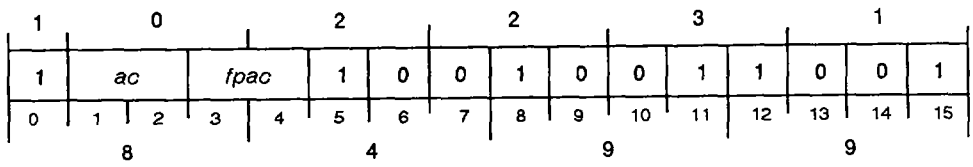
If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the exponential calculation. The **WPOPJ** instruction exits this software emulator.

Example

```
FMOV   2,0          ;Calculate the single precision exponential
WFEXPSEXPS          ;of the value in FPAC2, and store the result
FMOV   0,2          ;back into FPAC2. EXPS is a routine that
                    ;is called if IIS is not available.
```

Wide Fix from Floating-Point Accumulator WFFAD

WFFAD *ac,fpac*



Function: integer(*fpac*) -> *ac*

Parameters: None

NOTE: If *fpac* = +, *ac* = #; if *fpac* = -, *ac* = 2#
If *fpac* < -2,147,483,648 or > 2,147,483,647; then 1 -> FPSR(MOF)

WFFAD converts the integer portion of the floating-point number contained in the specified *fpac* to a signed 32-bit integer. It does this by taking the absolute value of the integer portion of the number contained in the *fpac* and appending a 0 onto the leftmost bit of the 31 least significant bits to produce a 32-bit number. If the sign of the number is negative, **WFFAD** forms the two's complement of the 32-bit result. The instruction then places the 32-bit result in the specified *ac*.

Arguments

ac	After execution, contains result.
fpac	Before execution, contains floating-point value.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
FPAC0-FPAC3	Can be specified as *fpac*; otherwise unused.
FPSR	Z and N flags unchanged. MOV set to 1, if integer portion of number in *fpac* outside range.
Overflow	Unaffected
PC	PC + 1
Stack	Unchanged

Related Instructions

WFLAD	Wide Float from Fixed-Point Accumulator

Exceptions

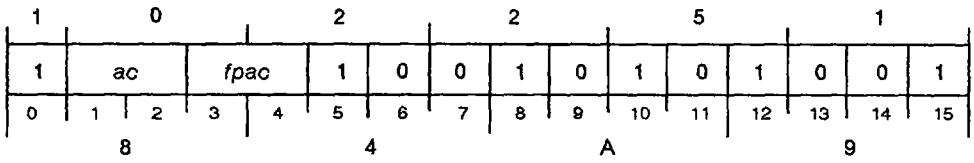
If the integer portion of the number contained in *fpac* is less than -2,147,483,648 or greater than +2,147,483,647, FPSR(MOV) is set to 1.

Example

```
LFLDD 0,FLPTX      ;Convert the floating point number at memory
WFFAD 2,0          ;location FLPTX into a 32-bit integer, and
                   ;store the result in AC2.
```

Wide Float from Fixed-Point Accumulator WFLAD

WFLAD *ac,fpac*



Function: *ac*[2#] -> *fpac*[fp#d]

Parameters: None

WFLAD converts the signed 32-bit integer in the specified *ac* into a 64-bit floating-point number and places the result into the specified *fpac*.

Arguments

ac	Before execution, contains signed 32-bit integer to be converted.
	After execution, contents unchanged.
fpac	After execution, contains 64-bit floating-point number.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
FPSR	Updated Z and N flags.
Overflow	Unaffected
PC	PC + 1
Stack	Unchanged

Related Instructions

WFFAD	Wide Fix from Floating-Point Accumulator

Exceptions

None

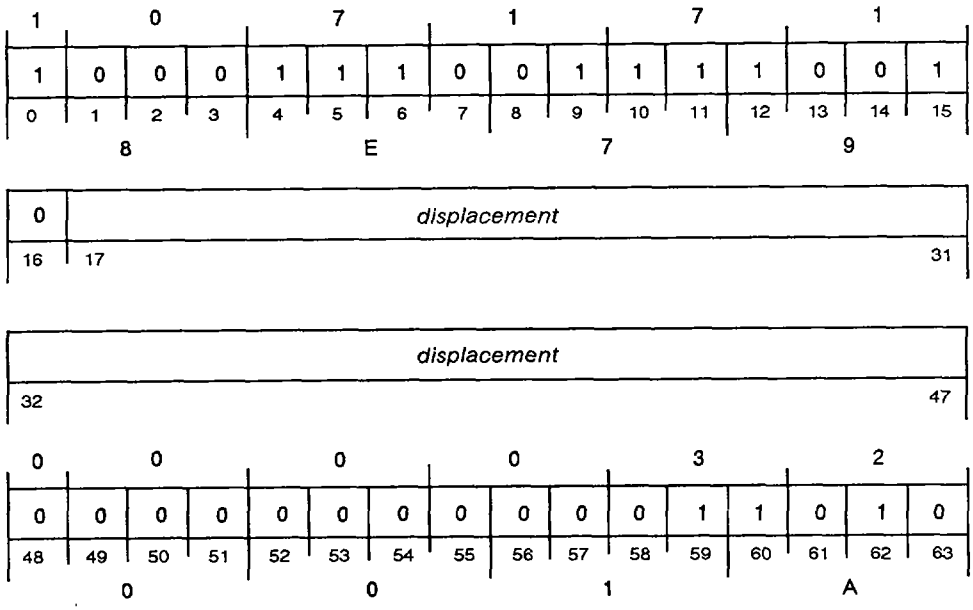
Example

```
WFLAD 3,0          ;Convert the 32-bit contents of AC3 into
LFSTD 0,FLOATF     ;a floating-point number, and store the
                   ;result at memory location FLOATF.
```

Floating-Point Binary Logarithm Double WFLG2D

Intrinsic Instruction

WFLG2D *displacement*



Function: $\log_2$ FPAC0 -> FPAC0

Parameters: FPAC0 = floating-point # -> result
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: If FPAC0 < = 0, then 1 -> FPSR(3), code 1 -> FPSR(28-31).

WFLG2D computes the binary logarithm (log base 2) of the 64-bit floating-point number contained in FPAC0 and places the result in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFLG2D** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0 Before execution, contains 64-bit floating-point value.

After execution, contains result.

FPAC1-FPAC3 Unused. After execution, contents undefined.

FPSR Updated Z and N flags.

PC PC + 4

Stack Unchanged

Related Instructions

WFLG2S Floating-Point Binary Logarithm Single

Exceptions

If the input value in FPAC0 is less than or equal to 0, the processor sets the INV bit in the FPSR to 1 and returns error code 1 to the INP bits of the FPSR.

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the binary logarithm calculation. The **WPOPJ** instruction exits this software emulator.

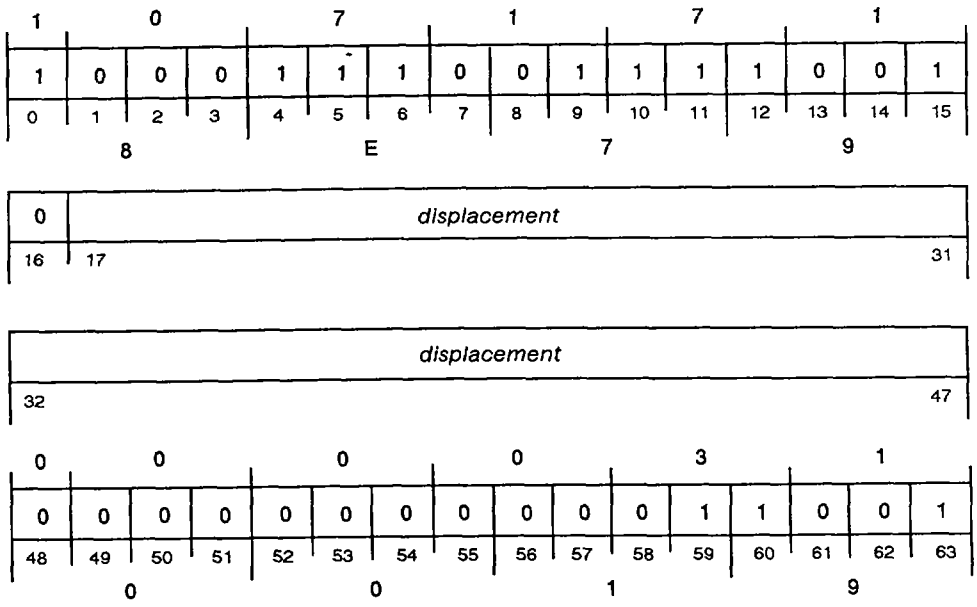
Example

```
LFLDD 0,X          ;Calculate the double precision Base 2 Log
WFLG2D LOG2D       ;of the floating-point number at location
LFSTD 0,LOG2X      ;X, and store the result at location LOG2X.
                   ;LOG2D is a routine that is called if IIS
                   ;is not available.
```

Floating-Point Binary Logarithm Single — WFLG2S

Intrinsic Instruction

WFLG2S *displacement*



Function: $\log_2$ FPAC0 -> FPAC0

Parameters: FPAC0 = floating-point # --> result
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: If FPAC0 < = 0, the 1 -> FPSR(3), code 1 -> FPSR(28-31).

WFLG2S computes the binary logarithm (log base 2) of the 32-bit floating-point number contained in FPAC0 and places the result in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFLG2S** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0(0-31) Before execution, contains 32-bit floating-point value.

After execution, contains 32-bit result with bits 32-63 set to 0.

FPAC1-FPAC3 Unused. Contents undefined after execution

FPSR Updated Z and N flags.

PC PC + 4

Stack Unchanged

Related Instructions

WFLG2D Floating-Point Binary Logarithm Double

Exceptions

If the input value in FPAC0 is less than or equal to 0, then the processor sets the INV bit in the FPSR to one and returns error 1 one to the INP bits of the FPSR.

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the binary logarithm calculation. The **WPOPJ** instruction exits this software emulator.

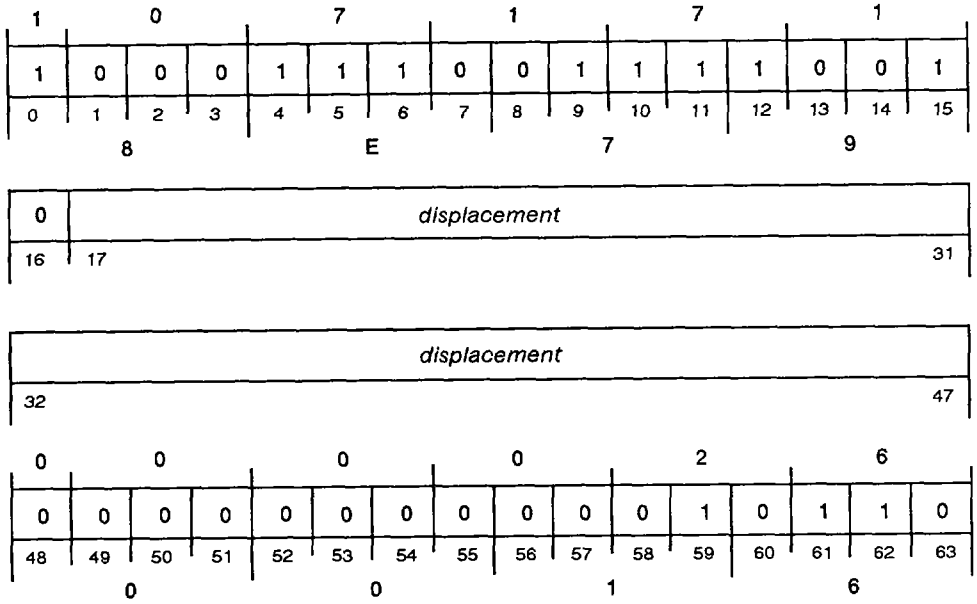
Example

```
FMOV  1,0          ;Calculate the single precision Base 2 Log
WFLG2S LOG2S       ;of the value in FPAC1, and store the result
LFSTS 0,RESULT     ;at location RESULT. LOG2S is a routine
                   ;that is called if IIS is not available.
```

Floating-Point Natural Logarithm Double WFLNGD

Intrinsic Instruction

WFLNGD *displacement*



Function: log e FPAC0 -> FPAC0

Parameters: FPAC0 = floating-point # -> result
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: If FPAC0 < = 0, then 1 -> FPSR(3), code 1 -> FPSR(28-31).

WFLNGD computes the natural logarithm (log base e) of the 64-bit floating-point number contained in FPAC0 and places the result in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFLNGD** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0 Before execution, contains 64-bit floating-point value.

After execution, contains result.

FPAC1-FPAC3 Unused. Contents undefined after execution

FPSR Updated Z and N flags.

PC PC + 4

Stack Unchanged

Related Instructions

WFLNGS Floating-Point Natural Logarithm Single

Exceptions

If the input value in FPAC0 is less than or equal to 0, the processor sets the INV bit in the FPSR to one and returns error code 1 to the INP bits of the FPSR.

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the natural logarithm calculation. The **WPOPJ** instruction exits this software emulator.

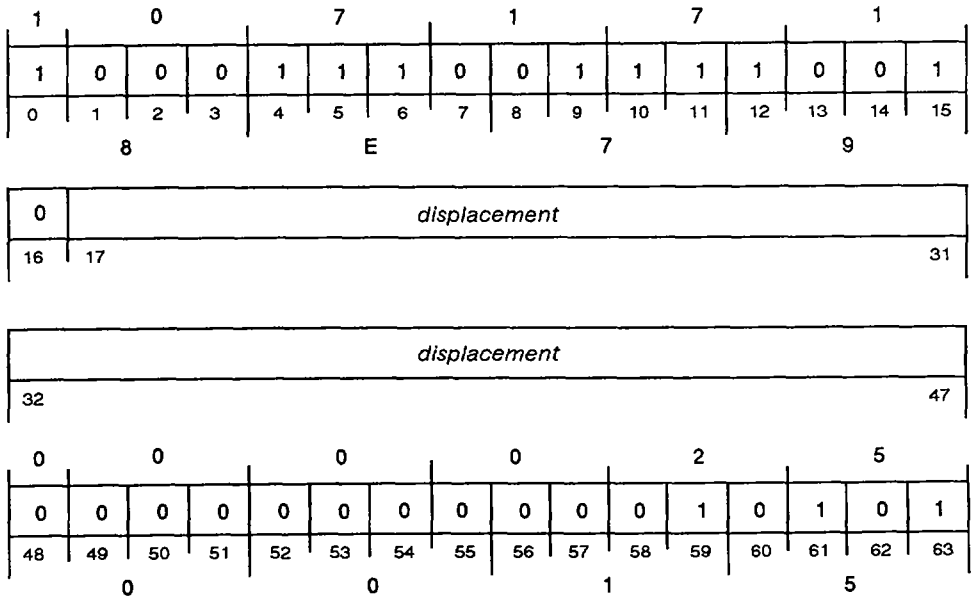
Example

```
LFLDD  0,FLPTX      ;Calculate the double precision natural log
WFLNGD NLOG         ;of the floating-point number at location
LFSTD  0,NLOGX      ;FLPTX, and store the result at location
                    ;NLOGX. NLOG is a routine that is called if
                    ;IIS is not available.
```

Floating-Point Natural Logarithm Single WFLNGS

Intrinsic Instruction

WFLNGS *displacement*



Function: log e FPAC0 -> FPAC0

Parameters: FPAC0 = floating-point # -> result
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: If FPAC0 < = 0, then 1 -> FPSR(3), code 1 -> FPSR(28-31).

WFLNGS computes the natural logarithm (log base e) of the 32-bit floating-point number contained in FPAC0 and places the result in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFLNGS** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

EPAC0(0-31) Before execution, contains 32-bit floating-point value.

After execution, contains 32-bit result with bits 32-63 set to 0.

FPAC1-FPAC3 Unused. Contents undefined after execution

FPSR Updated Z and N flags.

PC PC + 4

Stack Unchanged

Related Instructions

WFLNGD Floating-Point Natural Logarithm Double

Exceptions

If the input value in FPAC0 is less than or equal to 0, the processor sets the INV bit in the FPSR to one and returns error code 1 to the INP bits of the FPSR.

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the natural logarithm calculation. The **WPOPJ** instruction exits this software emulator.

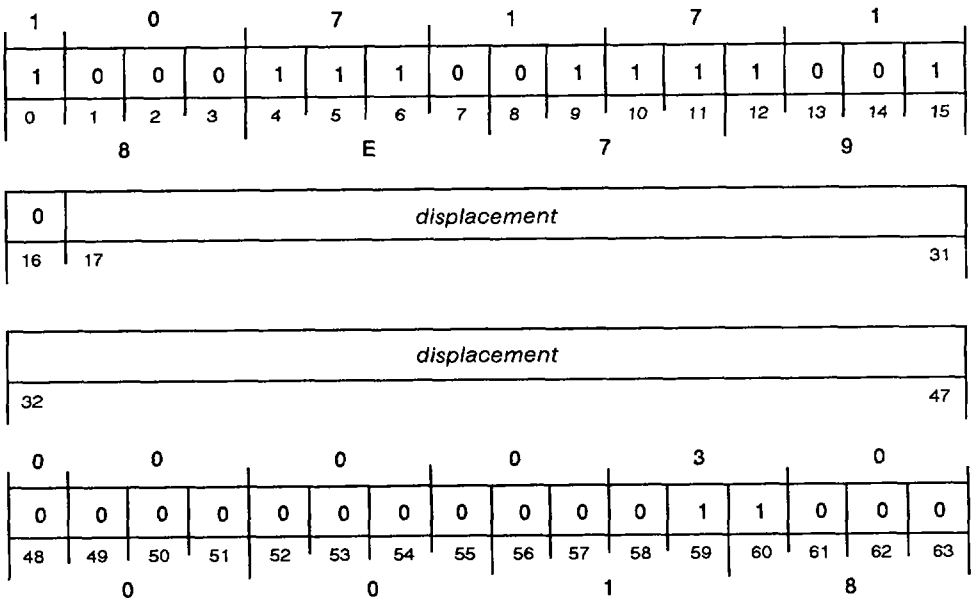
Example

```
FMS    2,0          ;Multiply FPAC0 by FPAC2, and store the
WFLNGS NLOG         ;single precision natural log of the
LFSTS  0,RESULT     ;product at location RESULT. NLOG is a
                    ;routine that is called if IIS is not
                    ;available.
```

Floating-Point Common Logarithm Double WFLOGD

Intrinsic Instruction

WFLOGD *displacement*



Function: $\log_{10}$ FPAC0 -> FPAC0

Parameters: FPAC0 = floating-point # -> result
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: If FPAC0 < = 0, then 1 -> FPSR(3), code 1 -> FPSR(28-31).

WFLOGD computes the common logarithm (log base 10) of the 64-bit floating-point number contained in FPAC0 and places the result in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFLOGD** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0 Before execution, contains 64-bit floating-point value.

After execution, contains result.

FPAC1-FPAC3 Unused. After execution, contents undefined.

FPSR Updated Z and N flags.

PC PC + 4

Stack Unchanged

Related Instructions

WFLOGS Floating-Point Common Logarithm Single

Exceptions

If the input value in FPAC0 is less than or equal to 0, the processor sets the INV bit in the FPSR to one and returns error code 1 to the INP bits of the FPSR.

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the common logarithm calculation. The **WPOPJ** instruction exits this software emulator.

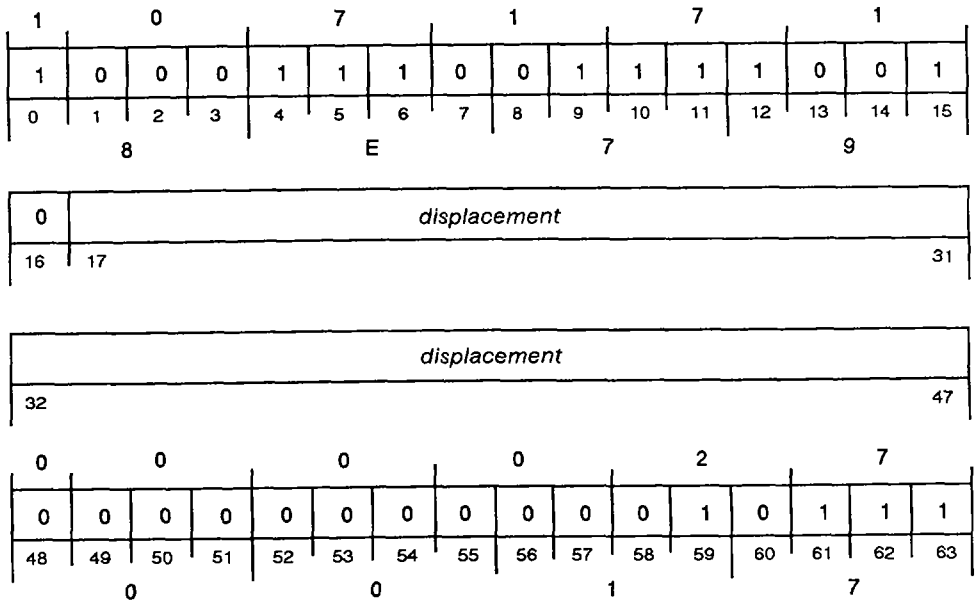
Example

```
LFLDD 0,FLPT1      ;Calculate the double precision common log
WFLOGD LOGD        ;of the floating-point number at location
LFSTD 0,LOGX       ;FLPT1, and store the result at location
                   ;LOGX. LOGD is a routine that is called if
                   ;IIS is not available.
```

Floating-Point Common Logarithm Single WFLOGS

Intrinsic Instruction

WFLOGS *displacement*



Function: $\log_{10}$ FPAC0 = fp# -> result

Parameters: FPAC0 = floating-point # -> result
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: If FPAC0 < = 0, then 1 -> FPSR(3), code 1 -> FPSR(28-31).

WFLOGS computes the common logarithm (log base 10) of the 32-bit floating-point number contained in FPAC0 and places the result in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFLOGS** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0(0-31) Before execution, contains 32-bit floating-point value.

After execution, contains 32-bit result with bits 32-63 set to 0.

FPAC1-FPAC3 Unused. After execution, contents undefined.

FPSR Updated Z and N flags.

PC PC + 4

Stack Unchanged

Related Instructions

WFLOGD Floating-Point Common Logarithm Double

Exceptions

If the input value in FPAC0 is less than or equal to 0, the processor sets the INV bit in the FPSR to one and returns error code 1 to the INP bits of the FPSR.

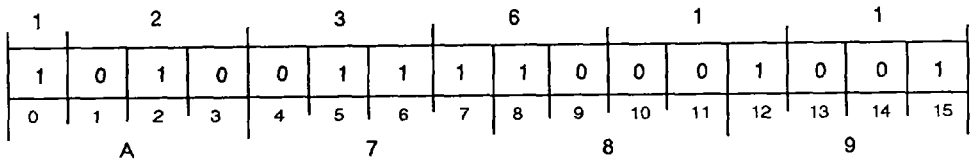
If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the common logarithm calculation. The **WPOPJ** instruction exits this software emulator.

Example

```
FMOV   2,0          ;Calculate the single precision common log
WFLOGS LOGS         ;of the value in FPAC2, and store the result
LFSTS  0,LOGX       ;at memory location LOGX. LOGS is a routine
                    ;that is called if IIS is not available.
```

Wide Floating-Point Pop WFPOP

WFPOP



Function: 10 stack double words -> registers

Parameters: 1st two double words -> FPAC3
2nd two double words -> FPAC2
3rd two double words -> FPAC1
4th two double words -> FPAC0
last two double words -> FPSR

NOTE: FPSR(bits 12-15) are not set by this instruction

WFPOP pops the state of the floating-point unit, a 10-double-word block, off the wide stack and loads the contents into the four floating-point accumulators and the floating-point status register (FPSR). The format of the 10-double-word block is shown in Figure 11-4.

The first 8 double words popped are loaded into the four fpacs; the last two double words popped are loaded as a 64-bit operand into the floating-point status register as follows:

FPSR(0) is not set from memory. If any of memory(1-4) are 1, ANY is set to 1; otherwise, ANY is 0.

FPSR(1-11) contains memory(1-11).

FPSR(12-15) is not set from memory. These bits contain floating-point identification code and cannot be changed.

FPSR(16-27) set to 0.

FPSR(28-31) set according to FPSR(0):

If ANY is 0, FPSR(28-31) set to 0.
If ANY is 1, FPSR(28-31) contain memory(28-31).

FPSR(32) set to 0.

FPSR(33-63) set according to FPSR(0):

If ANY is 0, FPSR(33-63) is undefined.
If ANY is 1, FPSR(33-63) contain memory(33-63).

Unnormalized data is moved without change.

Arguments

None

Registers, Flags, and Stacks

FPAC0–FPAC3	After execution, contains data from stack as shown in Figure 11–4.
PC	PC + 1
FPSR	After execution, contains data from stack.
Stack	Wide stack pointer decremented by 10 double words.

Related Instructions

FPOP Floating–Point Pop

FPSH, WFPSH Push the floating–point register contents onto the wide stack.

Exceptions

WFPOP initiates a floating–point trap if FPSR(ANY) and FPSR(TE) are both 1 after FPSR(FPPC) is loaded.

Example

```
WFPSH              ;Save the floating-point state before
LCALL  ROUTINE     ;calling this routine, since it may destroy
WFPOP              ;some of the FPACs. Restore the floating
                   ;point state after returning.
```

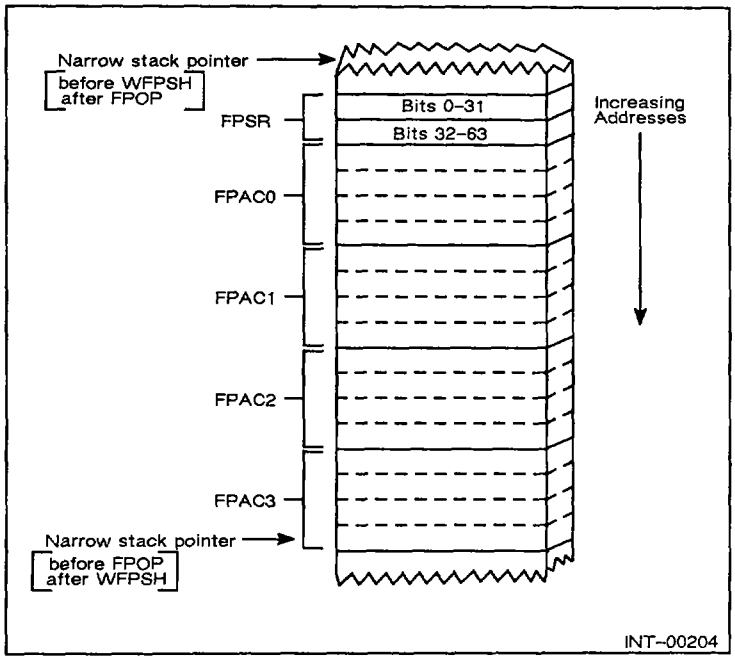
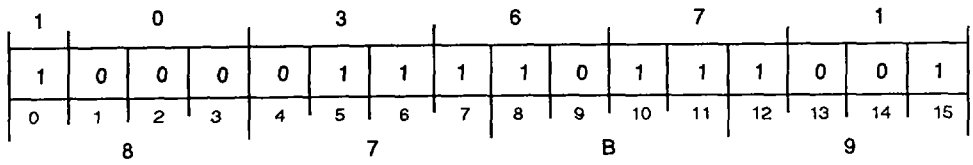


Figure 11–4 *Wide stack, 20–word floating–point return block*

Wide Floating-Point Push WFPSH

WFPSH



Function: registers -> 10 stack double words

Parameters: FPSR -> 1st two double words
FPAC0 -> 2nd two double words
FPAC1 -> 3rd two double words
FPAC2 -> 4th two double words
FPAC3 -> 5th two double words

WFPSH pushes the state of the floating-point unit, a 10-double-word block, onto the wide stack, leaving the contents of the floating-point accumulators and the floating-point status register unchanged. The format of the 10 double words pushed is shown in Figure 11-4.

The first two double words pushed onto the stack are taken as a 64-bit operand from the floating-point status register as follows:

Memory(0-15) contains FPSR(0-15).

Memory(16-27) set to 0.

Memory(28-31) dependent on FPSR(0):

If ANY is 0, memory(28-31) set to 0.
If ANY is 1, memory(28-31) contain FPSR(28-31).

Memory(32) set to 0.

Memory(33-63) dependent on FPSR(0):

If ANY is 0, memory(33-63) undefined.
If ANY is 1, memory(33-63) contain FPSR(33-63).

The next 8 double words pushed are taken from the four fpacs.

Unnormalized data is moved without change.

Arguments

None

Registers, Flags, and Stacks

FPAC0-FPAC3	After execution, contents unchanged.
PC	PC + 1
FPSR	After execution, contents unchanged.
Stack	Wide stack pointer incremented by 10 double words.

Related Instructions

FPSH Floating-Point Push

FPOP, WFPOP Load floating-point accumulators and floating-point status register with contents of wide stack.

Exceptions

WFPSH will not store a value with any combination of bit 5 (TE) and bits 1-4 concurrently set.

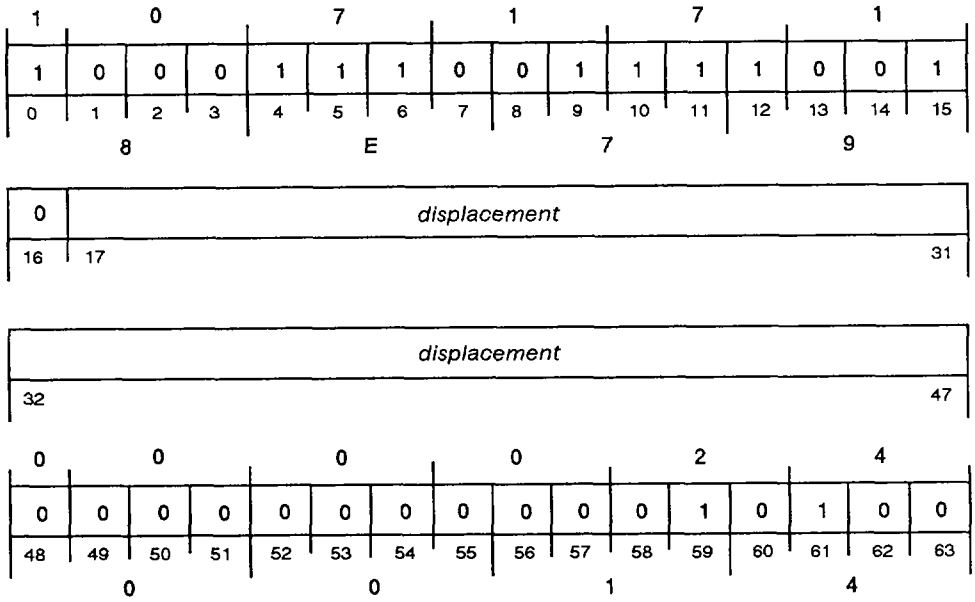
Example

```
WFPSH              ;Take the double precision sine of the
WFSINS SINS        ;value in FPAC0, and save the result at
LFSTS 0,RESULT     ;location RESULT. The WFPSH and WFPOP
WFPOP              ;protect the FPACs from being destroyed by
                   ;WFSINS.
```

Floating-Point Power Double — WFPWRD

Intrinsic Instruction

WFPWRD *displacement*



Function: FPAC0 ** FPAC1 -> FPAC0

Parameters: FPAC0 = floating-point # -> result
FPAC1 = floating-point #[power] -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: If FPAC0 < 0 and FPAC1 <> 0 or
if FPAC0 = 0 and FPAC1 < + 0, then 1 -> FPSR(3), code 4 -> FPSR(28-31).

WFPWRD raises the 64-bit floating-point number contained in FPAC0 to the 64-bit floating-point power contained in FPAC1 and places the result in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFPWRD** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0	Before execution, contains 64-bit floating-point value.
	After execution, contains 64-bit result.
FPAC1	Before execution, contains 64-bit floating-point value.
	After execution, undefined.
FPAC2-FPAC3	Unused. After execution, contents undefined.
FPSR	Updated Z and N flags.
PC	PC + 4
Stack	Unchanged

Related Instructions

WFPWRS Floating-Point Power Single

Exceptions

If the input value in FPAC0 is less than 0 AND the value in FPAC1 is not equal to 0, OR the input value in FPAC0 equals 0 AND the value in FPAC1 is less than or equal to 0, the processor sets the INV bit in the FPSR to one and returns error code 4 to the INP bits of the FPSR.

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the power calculation. The **WPOPJ** instruction exits this software emulator.

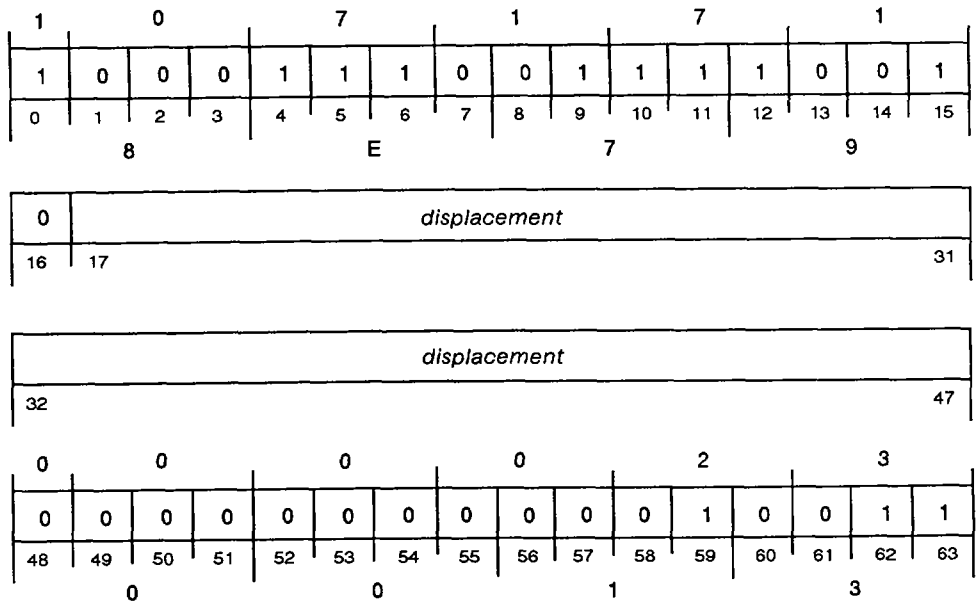
Example

```
LFLDD  0,BASE      ;Raise the double precision number at
LFLDD  1,POWER     ;location BASE to the power specified by
WFPWRD PWRD        ;the double precision number at location
LFSTD  0,RESULT    ;POWER, and store the result at location
                   ;RESULT. PWRD is a routine that is called
                   ;if IIS is not available.
```

Floating-Point Power Single — WFPWRS

Intrinsic Instruction

WFPWRS *displacement*



Function: FPAC0 ** FPAC1 -> FPAC0

Parameters: FPAC0 = floating-point # -> result
FPAC1 = floating-point #[power] -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: If FPAC0 < 0 and FPAC1 < > 0 or
if FPAC0 = 0 and FPAC1 < = 0, then 1 -> FPSR(3), code 4 -> FPSR(28-31).

WFPWRS raises the 32-bit floating-point number contained in FPAC0 to the 32-bit floating-point power contained in FPAC1 and places the 32-bit result in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFPWRS** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0(0-31)	Before execution, contains 32-bit floating-point value.
	After execution, contains 32-bit result with bits 32-63 set to 0.
FPAC1(0-31)	Before execution, contains 32-bit floating-point value.
	After execution, contents undefined.
FPAC2-FPAC3	Unused. After execution, contents undefined.
FPSR	Updated Z and N flags.
PC	PC + 4
Stack	Unchanged

Related Instructions

WFPWRD Floating-Point Power Double

Exceptions

If the input value in FPAC0 is less than 0 AND the value in FPAC1 is not equal to 0, OR the input value in FPAC0 equals 0 AND the value in FPAC1 is less than or equal to 0, the processor sets the INV bit in the FPSR to one and returns error code 4 to the INP bits of the FPSR.

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the power calculation. The **WPOPJ** instruction exits this software emulator.

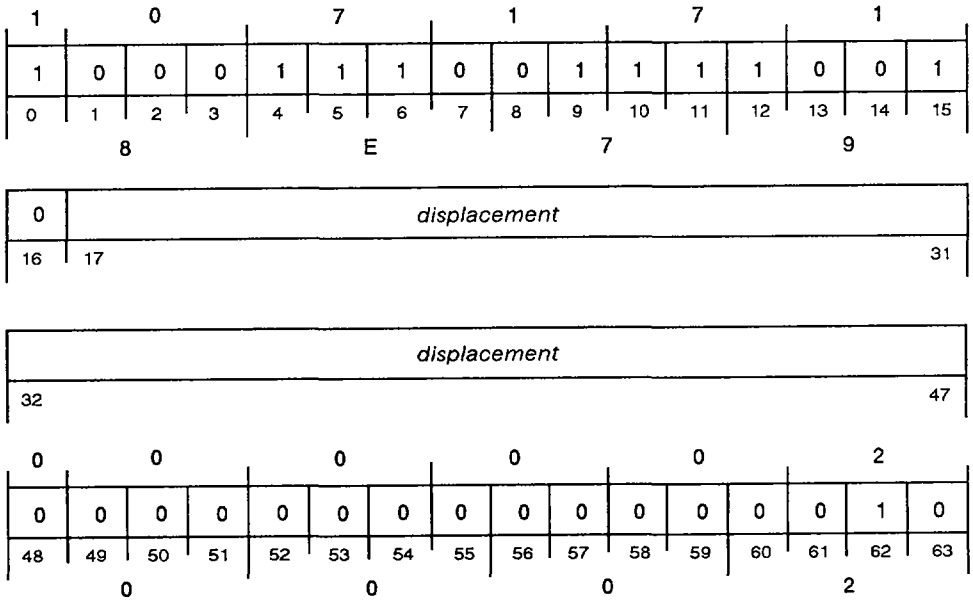
Example

```
LFLDS  0,BASE      ;Raise the single precision number at
LFLDS  1,POWER     ;location BASE to the power specified by
WFPWRS PWRS        ;the single precision number at location
LFSTS  0,RESULT    ;POWER, and store the result at location
                   ;RESULT. PWRS is a routine that is called
                   ;if IIS is not available.
```

Floating-Point Sine Double — WFSIND

Intrinsic Instruction

WFSIND *displacement*



Function: sine FPAC0 -> FPAC0

Parameters: FPAC0 = floating-point #[radians] -> sine
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: Input is in radians.

WFSIND computes the sine of the 64-bit floating-point value (in radians) in FPAC0 and places the result in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFSIND** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0 Before execution, contains 64-bit floating-point value (in radians).

After execution, contains 64-bit result.

FPAC1-FPAC3 Unused. After execution, contents undefined.

FPSR Updated Z and N flags.

PC PC + 4

Stack Unchanged

Related Instructions

WFSINS Floating-Point Sine Single

Exceptions

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the sine calculation. The **WPOPJ** instruction exits this software emulator.

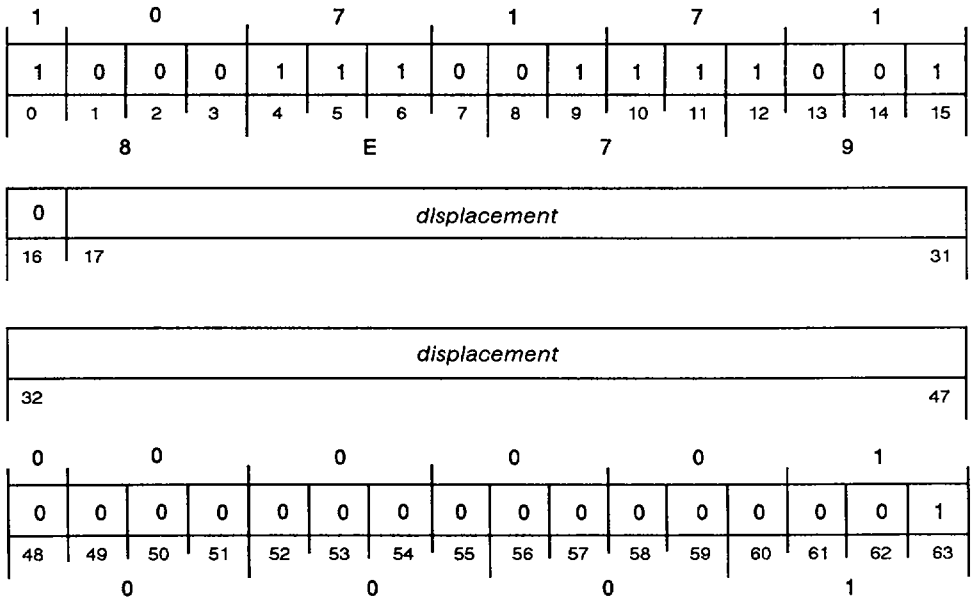
Example

```
LFLDD 0,FLOATX      ;Calculate the double precison sine of the
WFSIND SIND         ;floating point number at location FLOATX,
LFSTD 0,SINDX       ;and store the result at location SINDX.
                    ;SIND is a routine that is called if IIS
                    ;is not available.
```

Floating–Point Sine Single WFSINS

Intrinsic Instruction

WFSINS *displacement*



Function: sine FPAC0 -> FPAC0

Parameters: FPAC0 = floating–point #[radians] -> sine
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: Input is in radians.

WFSINS computes the sine of the 32–bit floating–point value (in radians) in FPAC0, and places the result in FPAC0.

Arguments

displacement Non–indirectable PC–relative offset to run–time routine that performs the **WFSINS** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0(0–31) Before execution, contains 32–bit floating–point value (in radians).

After execution, contains 32–bit result with bits 32–63 set to 0.

FPAC1–FPAC3 Unused. After execution, contents undefined.

FPSR Updated Z and N flags.

PC PC + 4

Stack Unchanged

Related Instructions

WFSIND Floating-Point Sine Double

Exceptions

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the sine calculation. The **WPOPJ** instruction exits this software emulator.

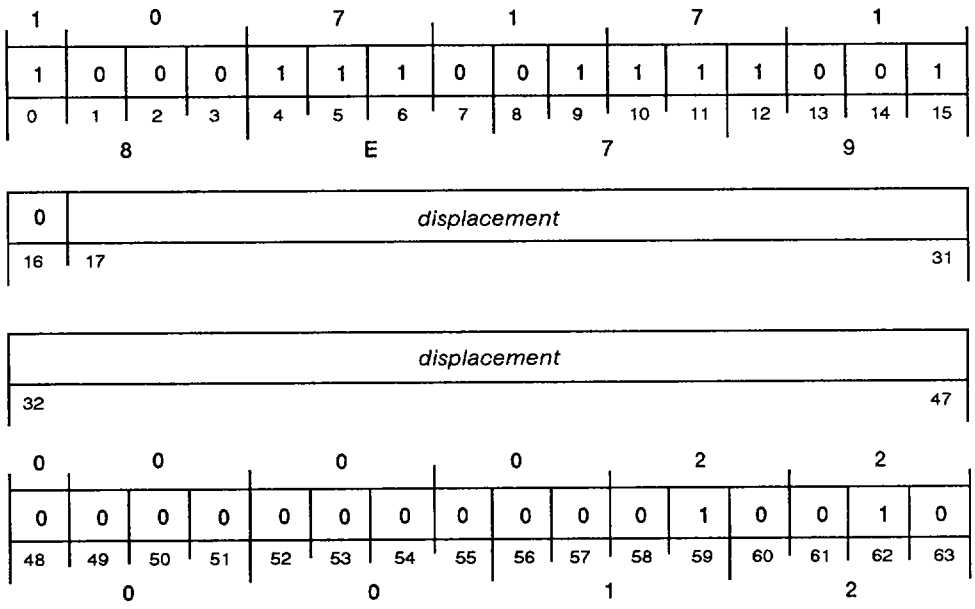
Example

```
FMOV   2,0          ;Calculate the single precision sine of the
WFSINS SINS         ;value in FPAC2, and store the result at
LFSTS  0,SINSX      ;location SINSX. SINS is a routine that is
                    ;called if IIS is not available.
```

Floating-Point Square Root Double WFSQRD

Intrinsic Instruction

WFSQRD *displacement*



Function: $\sqrt{\overline{FPAC0}}$ -> FPAC0

Parameters: FPAC0 = floating-point # -> result
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: If FPAC0 < 0, then 1 -> FPSR(3), code 2 -> FPSR(28-31).

WFSQRD computes the square root of the 64-bit floating-point number contained in FPAC0 and places the result in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFSQRD** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0 Before execution, contains 64-bit floating-point value.

After execution, contains 64-bit result.

FPAC1-FPAC3 Unused. After execution, contents undefined.

FPSR Updated Z and N flags.

PC PC + 4

Stack Unchanged

Related Instructions

WFSQRS Floating-Point Square Root Single

Exceptions

If the input value in FPAC0 is less than 0, the processor sets the INV bit in the FPSR to one and returns error code 2 to the INP bits of the FPSR.

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the square root calculation. The **WPOPJ** instruction exits this software emulator.

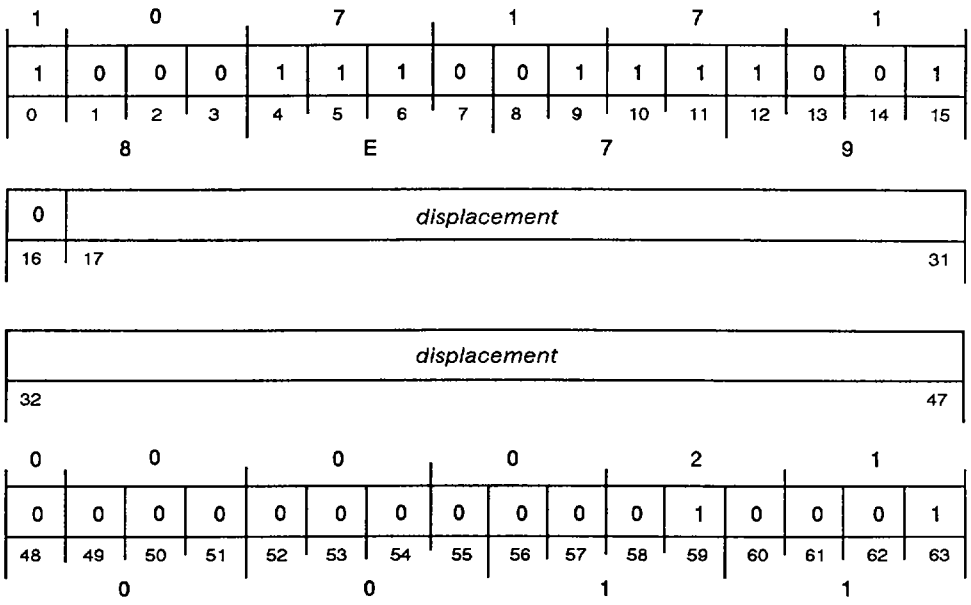
Example

```
LFLDD 0,FLPT1      ;Calculate the double precision square root
WFSQRD SQRD        ;of the floating-point number at location
LFSTD 0,FLPT2      ;FLPT1, and store the result at location
                   ;FLPT2. SQRD is a routine that is called if
                   ;IIS is not available.
```

Floating-Point Square Root Single WFSQRS

Intrinsic Instruction

WFSQRS *displacement*



Function: $\backslash/\overline{FPAC0}$ -> FPAC0

Parameters: FPAC0 = floating-point # -> result
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: If FPAC0 < 0, then 1 -> FPSR(3), code 2 -> FPSR(28-31).

WFSQRS computes the square root of the 32-bit floating-point number contained in FPAC0 and places the result in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFSQRS** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0(0-31) Before execution, contains 32-bit floating-point value.

After execution, contains 32-bit result with bits 32-63 set to 0.

FPAC1-FPAC3 Unused. After execution, contents undefined.

FPSR Updated Z and N flags.

PC PC + 4

Stack Unchanged

Related Instructions

WFSQRD Floating-Point Square Root Double

Exceptions

If the input value in FPAC0 is less than zero, the processor sets the INV bit in the FPSR to one and returns error code 2 to the INP bits of the FPSR.

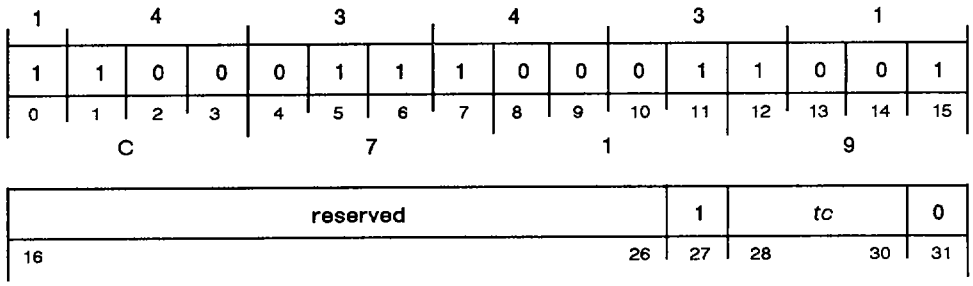
If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the square root calculation. The **WPOPJ** instruction exits this software emulator.

Example

```
FMOV   2,0          ;Calculate the single precision square root
WFSQRS SQRS         ;of the value in FPAC2, and move the result
FMOV   0,2          ;back into FPAC2. SQRS is a routine that
                    ;is called if IIS is not available.
```

Wide Forward Search Queue and Skip — WFStc

WFS*tc*
(Unsuccessful return)
(Interrupt return)
(Successful return)



Function: Search from @(AC1) to Q head for @(AC1 + AC3) = 32-bit test (*tc*)

=all 0	{AC}
=all 1	{AS}
=(wsp)	{E}
<=(wsp)	{GE}
>=(wsp)	{LE}
≠(wsp)	{NE}
=some 0s	{SC}
=some 1s	{SS}

Parameters:
AC1 = E(first queue data element - E(Q element -- See Note)
AC3 = 2#(word offset) --> unch
(wsp) = mask word --> unch

NOTE: The call sequence for the Search Queue instruction is:

Search Queue instruction	
Unsuccessful Return	E(last element searched) --> AC1
Interrupt Return	E(next element to search) --> AC1
Successful Return	E(last element searched) --> AC1

WFS*tc* searches forward through a queue, examining a 32-bit data field. The processor locates the beginning queue element by calculating the effective address (AC1). The data field examined in this element is located by adding to AC1 the offset in AC3. The result is then compared to a 32-bit mask. The search continues until the processor reaches either the tail of the queue or a data element that meets the test condition (*tc*).

Arguments

tc Bits 28-30 specify search condition.

tc Value	Bits 28-30 Encoding	Meaning
SS	0 0,0	Some of sampled test location bits 1.
SC	0 0,1	Some of sampled test location bits 0.
AS	0 1,0	All of sampled test location bits 1.
AC	0 1,1	All of sampled test location bits 0.
E	1 0,0	Mask and test location equal.
GE	1 0,1	Mask greater than or equal to test location.
LE	1 1,0	Mask less than or equal to test location.
NE	1 1,1	Mask and test location not equal.

NOTE: *For* **E, GE, LE,** *and* **NE** *test conditions, instruction treats values contained in mask and in test location as unsigned 32-bit integers.*

Registers, Flags, and Stacks

AC0	Unused
AC1	Before execution, contains effective address. With AC3, identifies location in data field as beginning data element in queue search. Processor increments AC1 for each data element it tests. If search succeeds, contains effective address of data element. New beginning pointer must be placed in AC1 if search is to continue through rest of queue. (If this is not done, and both the search condition and the examined data field in the element remain unchanged, continuing the search will result in this element being found again.) If search fails, contains effective address of last data element searched. If processor interrupts search (after unsuccessful search or another interrrupt only), contains effective address of next data element to be searched.
AC2	Unused
AC3	Before execution, contains signed 32-bit integer for word offset which is added to AC1 to identify location of data field in beginning data element in queue search. After execution, for all returns, contents unchanged.
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 2 (unsuccessful return) Processor honors interrupts between completed search and execution of PC + 2 instruction. PC + 3 (interrupted return) Processor honors interrupts between occurrence of interrupt and execution of PC + 3 instruction. PC + 4 (successful return) Processor does not honor interrupts between completed search and execution of PC + 4 instruction.
PSR	Unchanged
Stack	Before execution, top wide stack double word contains mask, identifying test location bits to sample. After execution, for all returns, wide stack unchanged.

Related Instructions

Queue Management
: Use these instructions to insert, delete, and test queue entries.

Exceptions

None

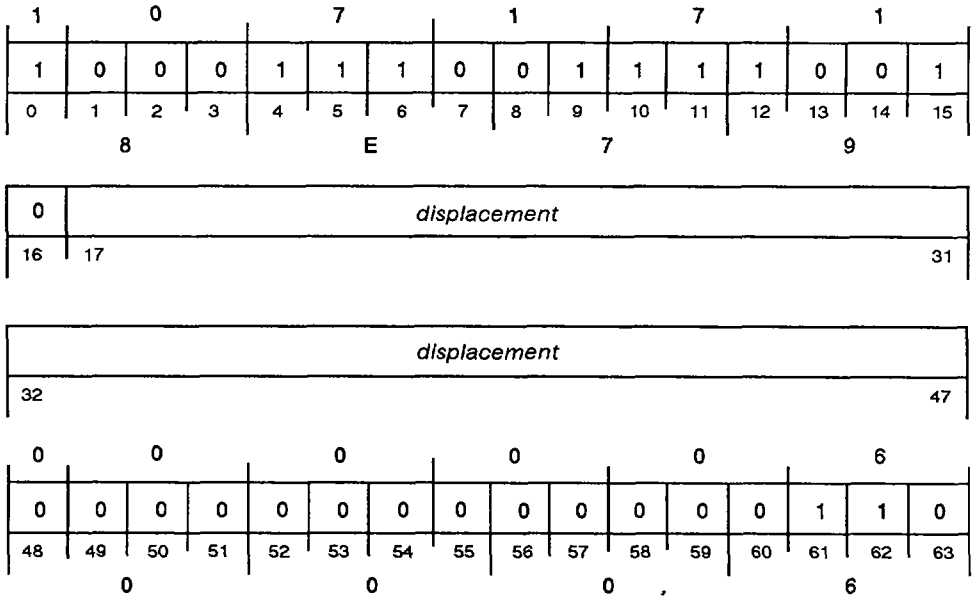
Example

```
    ;This example searches through a queue, finding all elements with a
    ;value of 6 at offset 4, and for all such elements changing the value
    ;to 0.
    WLDAI  6,0          ;Push the value to search
    WPSH   0,0          ;for onto the stack.
    LWLDA  1,HEAD       ;Put address of first queue element in AC1
                        ;to start search.
    WLDAI  4,3 ;Field to test is at offset 4 in each element.
REPEAT: WFSE            ;Find an element whose data field equals 6.
        JMP DONE        ;If none found, all done.
        JMP REPEAT      ;If interrupted, just continue.
        WMOV 1,2        ;Copy address of found element to AC2.
        WSUB 0,0        ;Put a 0 in AC0.
        XWSTA 0,4,2     ;Store it in offset 4 in the element.
        JMP REPEAT      ;Go look for next element.
DONE:   WPOP 0,0        ;Restore stack.
        .
        .
        .
HEAD:   .DWORD
```

Floating-Point Tangent Double WFTAND

Intrinsic Instruction

WFTAND *displacement*



Function: tangent FPAC0 -> FPAC0

Parameters: FPAC0 = floating-point #[radians] -> tangent
FPAC1 = x -> ?
FPAC2 = x -> ?FPAC3 = x -> ?

NOTE: Input is in radians.
If result will overflow, then 1 -> FPSR(3), code 6 -> FPSR(28-31).

WFTAND computes the tangent of the 64-bit floating-point value (in radians) in FPAC0, and places the result in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFTAND** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0 Before execution, contains 64-bit floating-point value (in radians).

After execution, contains result.

FPAC1-FPAC3 Unused. After execution, contents undefined.

FPSR Updated Z and N flags.

PC PC + 4

Stack Unchanged

Related Instructions

WFTANS Floating-Point Tangent Single

Exceptions

If the input value in FPAC0 produces a result that overflows (odd integer multiples of values near PI/2), the processor sets the FPSR(INV) flag to 1 and returns error code 6 to the FPSR(INP) bits. (If the integer multiples of values is even, the FPSR(INP) bits contain 0.)

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a non-indirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the tangent calculation. The **WPOPJ** instruction exits this software emulator.

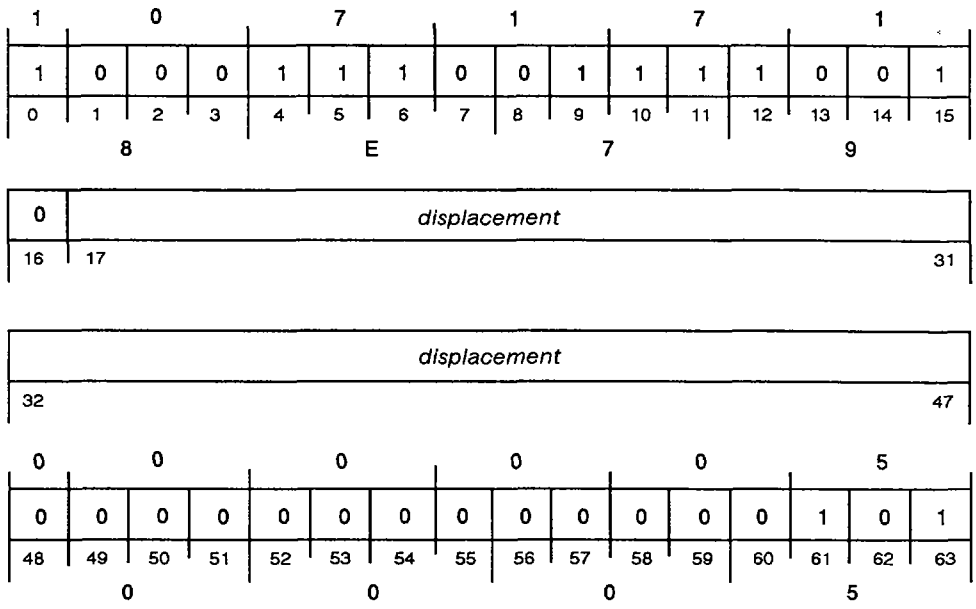
Example

```
LFLDD 0,FLOATX      ;Calculate the double precision tangent
WFTAND FTAND        ;of the floating-point number at location
LFSTD 0,TANDX       ;FLOATX, and store the result at location
                    ;TANDX. FTAND is a routine that is called
                    ;if IIS is not available.
```

Floating-Point Tangent Single WFTANS

Intrinsic Instruction

WFTANS *displacement*



Function: tangent FPAC0 -> FPAC0

Parameters: FPAC0 = floating-point #[radians] -> tangent
FPAC1 = x -> ?
FPAC2 = x -> ?
FPAC3 = x -> ?

NOTE: Input is in radians.
If result will overflow, then 1 -> FPSR(3), code 6 -> FPSR(28-31).

WFTANS computes the tangent of the 32-bit floating-point value (in radians) in FPAC0 and places the result in FPAC0.

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WFTANS** function (if hardware support is currently unavailable). Effective address generated by instruction is confined to current segment.

Registers, Flags, and Stacks

FPAC0(0-31) Before execution, contains 32-bit floating-point value (in radians).

After execution, contains 32-bit result with bits 32-63 set to 0.

FPAC1-FPAC3 Unused. After execution, contents undefined.

FPSR Updated Z and N flags.

PC PC + 4

Stack Unchanged

Related Instructions

WFTAND Floating-Point Tangent Double

Exceptions

If the input value in FPAC0 produces a result that overflows (odd integer multiples of values near PI/2), the processor sets the FPSR(INV) flag to 1 and returns error code 6 to the FPSR(INP) bits. (If the integer multiples of values is even, the FPSR(INP) bits contain 0.)

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a nonindirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a run-time routine that performs the tangent calculation. The **WPOPJ** instruction exits from this software emulator.

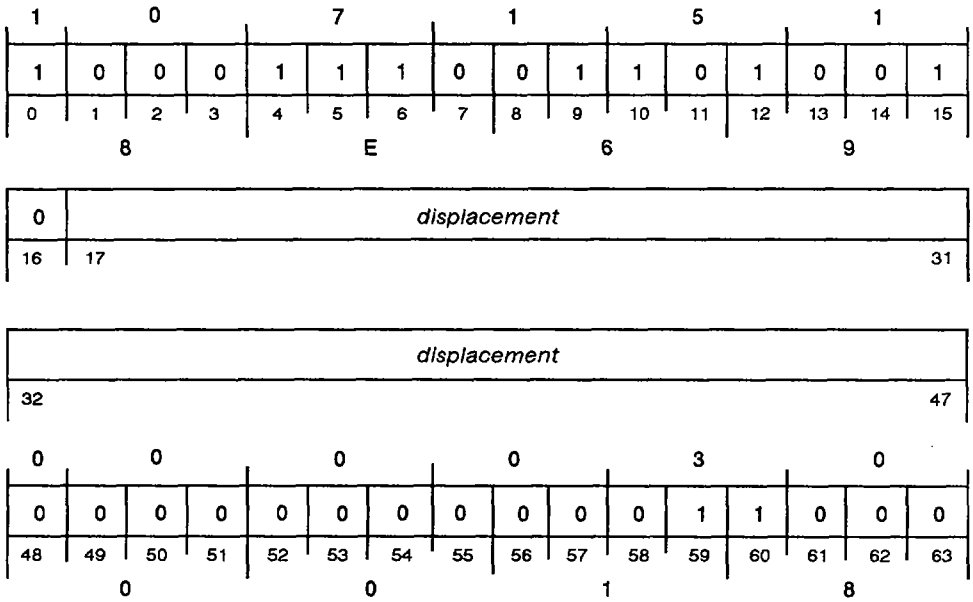
Example

```
FMOV  1,0          ;Calculate the single precision tangent of the
WFTANS FTANS       ;value in FPAC1, and store the result at location
LFSTS 0,TANGNT     ;TANGNT. FTANS is a routine that is called if IIS
                   ;is not available.
```

Bit Block Transfer WGBITBLT

Graphics Instruction

WGBITBLT *displacement*



Function: source form (combination rule & operation mask & form mask) --> destination form

Parameters: AC0 = Source form ID --> unchanged
AC1 = Destination form ID --> unchanged
AC2 = Address of **WGBITBLT** packet --> unchanged

WGBITBLT copies a rectangular set of pixels from one form to another, or from one location to another in a single form. Pixels in the destination form are modified according to its combination rule, operation mask, and form mask.

To be copied, a pixel in the source rectangle must be within the source form bounds, and the destination location in the destination rectangle must be within the destination form.

Arguments

displacement Nonindirectable PC-relative offset to runtime routine that performs the **WGBITBLT** function (if hardware support is currently unavailable). Effective address generated by instruction confined to the current segment.

Registers, Flags, and Stacks

AC0 Before execution, contains 32-bit source form ID.

After execution, contents unchanged.

AC1 Before execution, contains 32-bit destination form ID.

After execution, contents unchanged.

AC2 Before execution, contains address of **WGBITBLT** packet.

After execution, contents unchanged.

Packet consists of six 32-bit integers as follows:

Double Word #	Contents
1	X coordinate of destination rectangle's ULC (signed).
2	Y coordinate of destination rectangle's ULC (signed).
3	Width of rectangle in pixels (unsigned).
4	Height of rectangle in pixels (unsigned).
5	X coordinate of source rectangle's ULC (signed).
6	Y coordinate of source rectangle's ULC (signed).

AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 4
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load with immediate
: Use these instructions to place the appropriate values into AC0 and AC1.

Load effective address
: Use these instructions to place the packet address into AC2.

LPSHJ, WPOPJ If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a nonindirectable PC–relative offset). The processor uses E as the address of a runtime routine that performs the **WGBITBLT** function. The **WPOPJ** instruction exits from this software emulator.

Exceptions

If the source and destination rectangles overlap, **WGBITBLT** transfers pixels in such a way that all pixels are read before they are modified.

If the specified rectangle does not lie entirely within one of the forms, **WGBITBLT** causes the rectangle to be clipped so that it fits into the smaller of the two forms.

No pixel will ever be taken from outside the source form, or drawn outside the destination form.

WGBITBLT will not write to pixels on the form that are write inhibited.

If either the width or the height of the rectangle (in the packet) is set to zero, **WGBITBLT** has no effect.

If an overdraw condition occurs, the PSR(OVR) bit is set to one.

WGBITBLT microcode is designed to optimize its speed if certain conditions are met. For the fastest possible execution, you should ensure that the following condition is true: for virtual to physical bitmap transfers, if the pixels are left justified to a double–word boundary, **WGBITBLT** transfers an integral number of whole double words for each line.

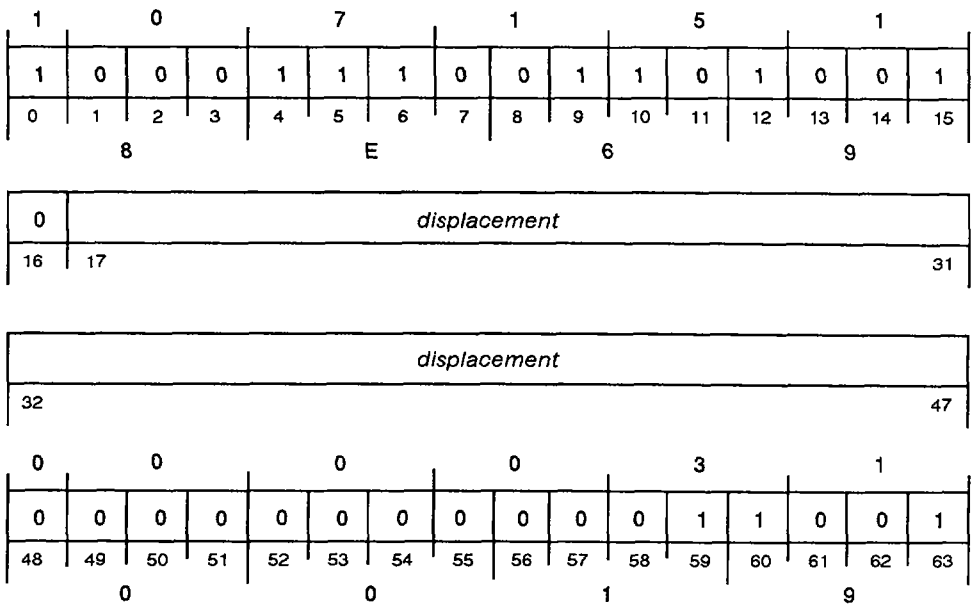
Example

```
WGBITBLT
```

Character Block Transfer WGCHRBLT

Graphics Instruction

WGCHRBLT *displacement*



Function: character (combination rule & operation mask & form mask) --> destination form

Parameters: AC0 = Form ID of character --> unchanged
AC1 = Destination form ID --> unchanged
AC2 = Address of **WGCHRBLT** packet --> unchanged

WGCHRBLT writes a character into the specified form. The character is any rectangle from a form with one bit per pixel. The instruction converts the bits in the character to the foreground and background colors specified by the destination form's attribute block. A character bit set to one indicates foreground color; a character bit set to zero indicates background color. Either color may be suppressed by bits in the character control word. Pixels in the destination form are modified according to its combination rule, operation mask, and form mask.

Arguments

displacement Nonindirectable PC-relative offset to runtime routine that performs the **WGCHRBLT** function (if hardware support is currently unavailable). Effective address generated by instruction confined to the current segment.

Registers, Flags, and Stacks

AC0 Before execution, contains 32-bit source (character) form ID.

After execution, contents unchanged.

AC1 Before execution, contains 32-bit destination form ID.

After execution, contents unchanged.

AC2 Before execution, contains address of **WGCHRBLT** packet.

After execution, contents unchanged.

Packet consists of six 32-bit integers as follows:

Double Word #	Contents
1	X coordinate of destination rectangle's ULC (signed).
2	Y coordinate of destination rectangle's ULC (signed).
3	Width of character in pixels (unsigned).
4	Height of character in pixels (unsigned).
5	X coordinate of character's ULC (signed).
6	Y coordinate of character's ULC (signed).

AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 4
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load with immediate
: Use these instructions to place the appropriate values into AC0 and AC1.

Load effective address
: Use these instructions to place the packet address into AC2.

LPSHJ, WPOPJ
: If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a nonindirectable PC-relative offset). The processor uses E as the address of a runtime routine that performs the **WGCHRBLT** function. The **WPOPJ** instruction exits from this software emulator.

Exceptions

If either the character width or the height value in the packet is zero, **WGCHRBLT** has no effect.

If the source and destination rectangles overlap, the results may be undefined, since **WGCHRBLT** does not read pixels before they are modified.

If the source form (character) is not one-bit per pixel or not on a virtual bitmap, an invalid **WGCHRBLT** source fault is generated.

If the specified character rectangle does not lie entirely within one of the forms, **WGCHRBLT** causes the character rectangle to be clipped so that it fits into the smaller of the two forms.

WGCHRBLT will not write to pixels on the form that are write inhibited.

If an overdraw condition occurs, the PSR(OVR) bit is set to one.

WGCHRBLT microcode is designed to optimize its speed if certain conditions are met. For the fastest possible execution, you should ensure that the following condition is true: if the pixels are left justified to a word boundary, **WGCHRBLT** transfers an integral number of single words for each line and for lines that are less than or equal to 16 pixels.

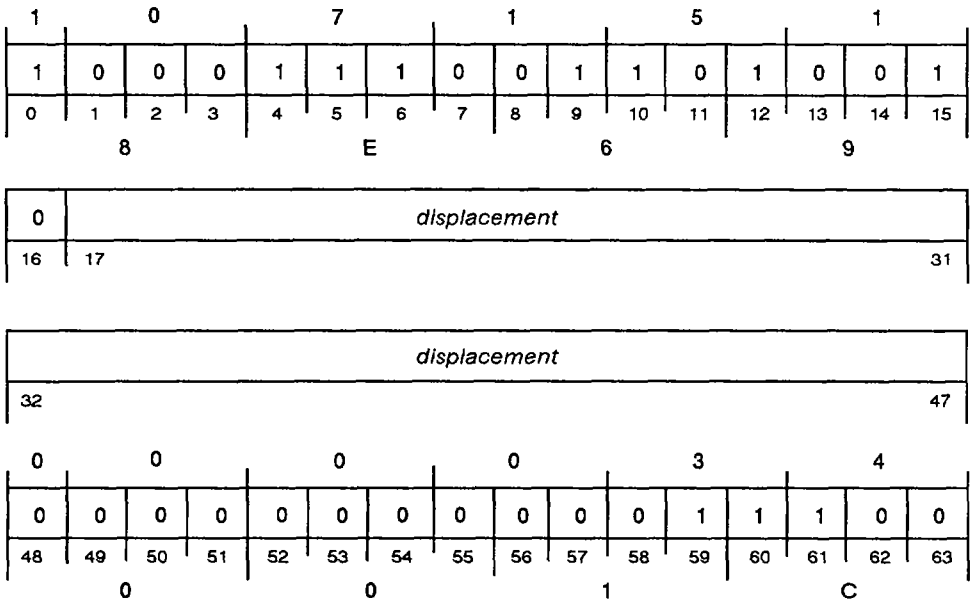
Example

```
WGCHRBLT
```

Load Cursor Descriptor WGLDCURS

Privileged Graphics Instruction

WGLDCURS *displacement*



Function: Write updated cursor block to cursor descriptor memory
Update the cursor shape (machine-dependent)

Parameters: AC0 = Physical address of cursor database --> unchanged
AC1 = Physical address of cursor descriptor --> unchanged
AC2 = Logical address of **WGLDCURS** packet --> unchanged

WGLDCURS loads the updated cursor block, specified by AC2, into the cursor descriptor in memory, specified by the physical address in AC1. The number of double words moved is determined by the new cursor type. If the cursor is invisible, **WGLDCURS** will move only one double-word. The cursor descriptor defines the size and position of the cursor. Note that the original cursor block is loaded into memory with the Load Form instruction (**WGLFORM**).

Arguments

displacement Non-indirectable PC-relative offset to run-time routine that performs the **WGLDCURS** function (if hardware support is currently unavailable). Effective address generated by instruction confined to the current segment.

Registers, Flags, and Stacks

AC0 Before execution, contains either physical address of cursor-drawing database or -1.

If the CPU does not support hardware cursor drawing, then AC0 should contain -1. If AC0 is not -1, then **WGLDCURS** will take an unimplemented instruction trap after moving the cursor descriptor packet.

If the CPU supports hardware cursor drawing, then AC0 should contain a physical address which points to cursor-drawing database. This database describes the shape and color of the cursor and its format. If AC0 contains -1, then the cursor-drawing database is assumed to have not changed.

	After execution, contents unchanged.
AC1	Before execution, contains physical address of cursor descriptor.
	After execution, contents unchanged.
AC2	Before execution, contains logical address of **WGLDCURS** packet. Refer to the "Cursor Descriptor" section of the "Graphics Management" chapter for the contents of the packets.
	After execution, contents unchanged.
AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load effective address
: Use these instructions to place the logical address into AC2.

LPHY
: Use this instruction to place physical addresses into AC0 and AC1.

LPSHJ, WPOPJ
: If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the displacement as a non-indirectable PC-relative offset). The processor uses E as the address of a run-time routine that performs the **WGLDCURS** function. The **WPOPJ** instruction exits this software emulator.

Exceptions

If cursor drawing is not implemented, this instruction takes an unimplemented instruction trap when AC0 is not -1.

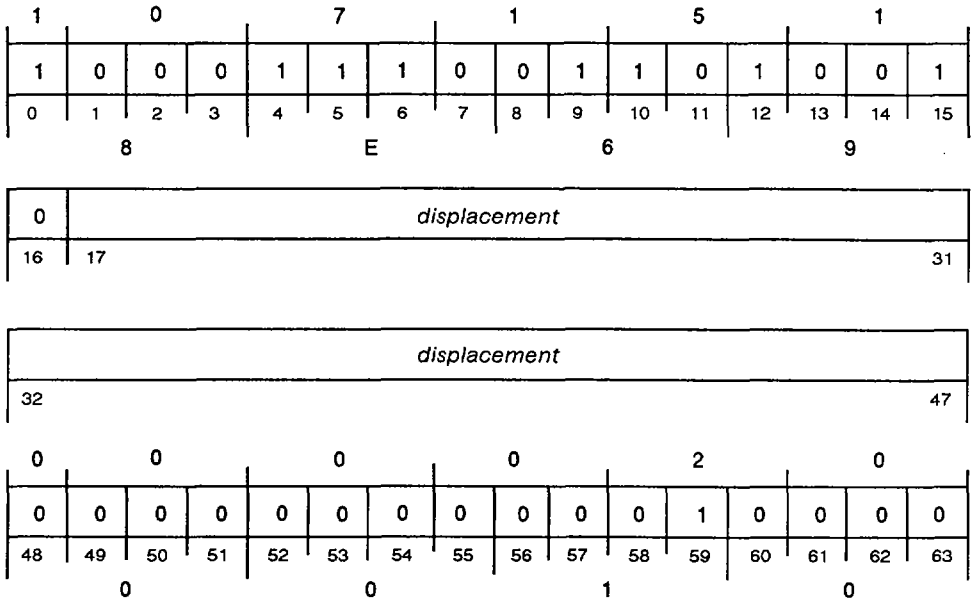
Example

```
WGLDCURS
```

Load Form WGLFORM

Privileged Graphics Instruction

WGLFORM *displacement*



Function: form --> form cache memory

Parameters: AC0 = Operating system's key --> unchanged
AC1 = User's form ID --> unchanged
AC2 = Physical address of form descriptor --> unchanged

WGLFORM loads the form, specified by AC2, into the form cache memory. The form is specified by the tag words in AC0 and AC1. If a form with the specified tag is already in the cache, it is overwritten.

Arguments

displacement Nonindirectable PC-relative offset to runtime routine that performs the **WGLFORM** function (if hardware support is currently unavailable). Effective address generated by instruction confined to the current segment.

Registers, Flags, and Stacks

AC0 Before execution, contains 32-bit operating system key.

After execution, contents unchanged.

AC1 Before execution, contains 32-bit user's form ID.

After execution, contents unchanged.

AC2 Before execution, contains physical address of form descriptor.

After execution, contents unchanged.

AC3 Unused

CARRY	Unchanged
Overflow	Unaffected
PC	PC + 4
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load with immediate
: Use these instructions to place the appropriate values into AC0 and AC1.

LPHY
: Use this instruction to place the physical address into AC2.

WGLDCURS
: Use the Load Cursor instruction to load an updated cursor descriptor block into memory.

LPSHJ, WPOPJ
: If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a nonindirectable PC-relative offset). The processor uses E as the address of a runtime routine that performs the **WGLFORM** function. The **WPOPJ** instruction exits from this software emulator.

Exceptions

WGLFORM will not load a form that has a form ID of 0 (invalid form) in AC1.

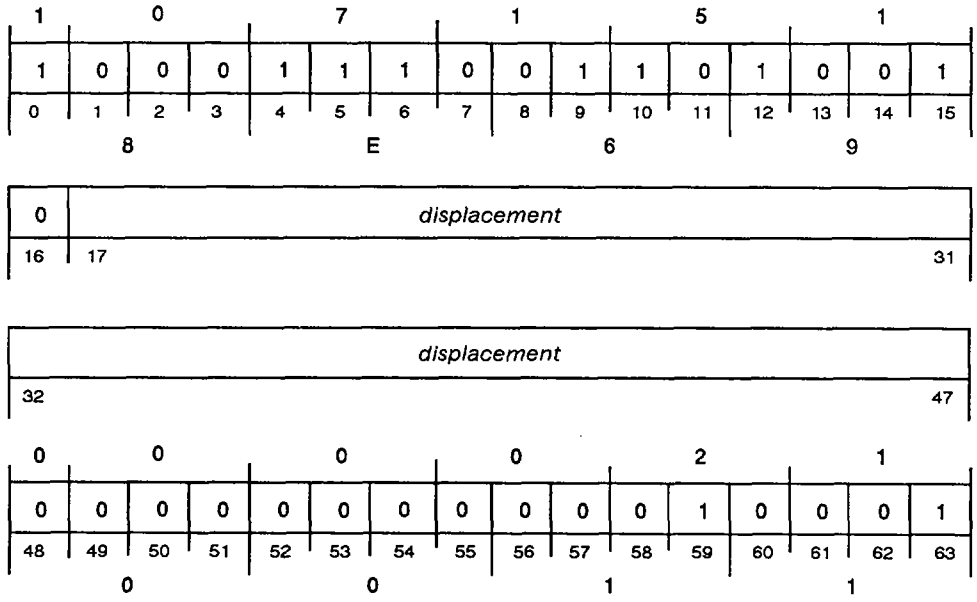
Example

```
WGLFORM
```

Purge Forms WGPFORMS

Privileged Graphics Instruction

WGPFORMS *displacement*



Function: If AC0 = 0, -(AC1) form --> form cache memory
If AC0 ≠ 0, -all forms --> form cache memory

Parameters: AC0 = purge specifier --> unchanged
AC1 = User's form ID --> unchanged
AC2 = ? --> unchanged

WGPFORMS removes one or more forms from the form cache memory. It can purge either a single form, or all forms in the cache depending on the value in AC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime routine that performs the **WGPFORMS** function (if hardware support is currently unavailable). Effective address generated by instruction confined to current segment.

Registers, Flags, and Stacks

AC0 Before execution,

if purging a single form, contains zero.

if purging all forms, contains nonzero.

After execution, contents unchanged.

AC1 Before execution,

if AC0 = 0, contains 32-bit user's form ID.

if AC0 = nonzero, unused.

After execution, contents unchanged.

AC2-AC3 Unused

CARRY	Unchanged
Overflow	Unaffected
PC	PC + 4
PSR	Unchanged
Stack	Unchanged

Related Instructions

WADC Use **WADC** 0,0 to place a zero in AC0.

Load with immediate
Use these instructions to place the form ID into AC1.

LPSHJ, WPOPJ
If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a nonindirectable PC-relative offset). The processor uses E as the address of a runtime routine that performs the **WGPFORMS** function. The **WPOPJ** instruction exits from this software emulator.

Exceptions

WGPFORMS microcode is designed to optimize its speed if certain conditions are met. For the fastest possible execution, only a single form should be purged from the form cache.

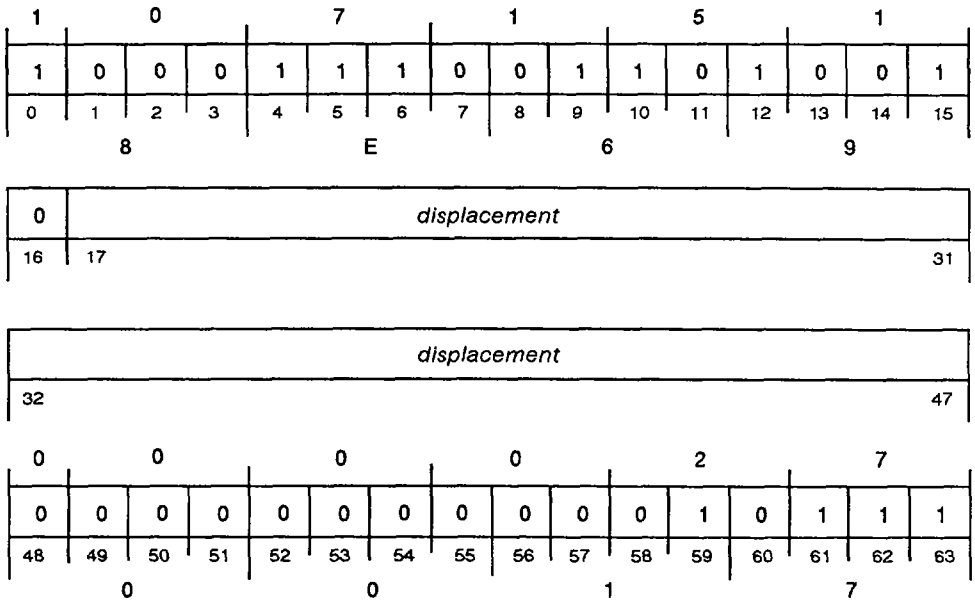
Example

```
WGPFORMS
```

Draw Polyline WGPLINE

Graphics Instruction

WGPLINE *displacement*



Function: Draw line segments in form (combination rule & operation mask & form mask)

Parameters: AC0 = Number of line segments --> unchanged
AC1 = Form ID --> unchanged
AC2 = Address of **WGPLINE** packet --> unchanged

WGPLINE draws one or more line segments in a form. Pixels in the form are modified according to the form's combination rule, operation mask, and form mask. Drawing of foreground and background colors is controlled by the line style and line control double words in the form's attribute block, allowing the creation of dotted, dashed, and other nonsolid lines. The foreground and background colors are specified by the form's foreground and background color attributes.

Arguments

displacement Nonindirectable PC–relative offset to runtime routine that performs the **WGPLINE** function (if hardware support is currently unavailable). Effective address generated by instruction confined to the current segment.

Registers, Flags, and Stacks

AC0 Before execution, contains unsigned 32–bit integer specifying number of line segments to draw.

After execution, contents unchanged.

AC1 Before execution, contains 32–bit form ID.

After execution, contents unchanged.

AC2 Before execution, contains address of **WGPLINE** packet.

After execution, contents unchanged.

Packet consists of 2*(N+1) signed 32-bit integers (N is the number in AC0) as follows:

Double Word #	Contents
1	X coordinate of first endpoint.
2	Y coordinate of first endpoint.
3	X coordinate of second endpoint.
4	Y coordinate of second endpoint.
.	...
2N-1	X coordinate of Nth endpoint.
2N	Y coordinate of Nth endpoint.

AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 4
PSR	Unchanged, unless an overdraw condition occurs.
Stack	Unchanged

Related Instructions

Load with immediate
: Use these instructions to place the appropriate values into AC0 and AC1.

Load effective address
: Use these instructions to place the packet address into AC2.

LPSHJ, WPOPJ
: If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a nonindirectable PC-relative offset). The processor uses E as the address of a runtime routine that performs the **WGPLINE** function. The **WPOPJ** instruction exits from this software emulator.

Exceptions

WGPLINE has no effect on pixels lying outside the form or on pixels on the form that are write inhibited.

If the precision of the line segment specified is greater than 30 bits ($2^{30}-1$), then an overdraw condition occurs, and the PSR(OVR) bit is set to one.

If the number of line segments is outside the range 1 to $2^{31} - 1$, then the value is ignored and no segments are drawn.

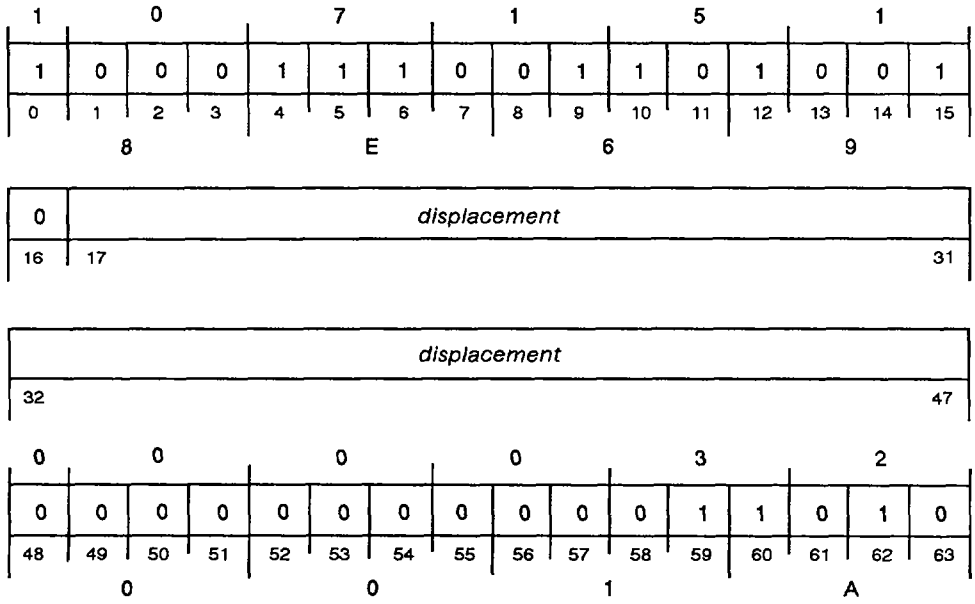
Example

```
WGPLINE
```

Read Attribute WGRDATTR

Graphics Instruction

WGRDATTR *displacement*



Function: read attribute from form

Parameters: AC0 = Attribute number --> unchanged
AC1 = Form ID --> unchanged
AC2 = Address at which to store value --> unchanged

WGRDATTR reads one of the attributes (indexed by AC0) of the specified form, storing its value at the address specified by AC2.

Arguments

displacement Nonindirectable PC-relative offset to runtime routine that performs the **WGRDATTR** function (if hardware support is currently unavailable). Effective address generated by instruction confined to the current segment.

Registers, Flags, and Stacks

AC0 Before execution, contains one of the following attribute numbers:

Attribute Number	Contents
0	Operation mask
1	Combination rule
2	Line control word
3	Line foreground color
4	Line background color
5	Line style
6	Character control word
7	Character foreground color
8	Character background color

After execution, contents unchanged.

AC1	Before execution, contains 32-bit form ID. After execution, contents unchanged.
AC2	Before execution, contains word address at which to store attribute value. After execution, contents unchanged.
AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 4
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load with immediate
: Use these instructions to place the appropriate values into AC0 and AC1.

Load effective address
: Use these instructions to place the address into AC2.

LPSHJ, WPOPJ
: If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a nonindirectable PC-relative offset). The processor uses E as the address of a runtime routine that performs the **WGRDATTR** function. The **WPOPJ** instruction exits from this software emulator.

Exceptions

If the value in AC0 is greater than 8_{10}, an invalid attribute index fault occurs.

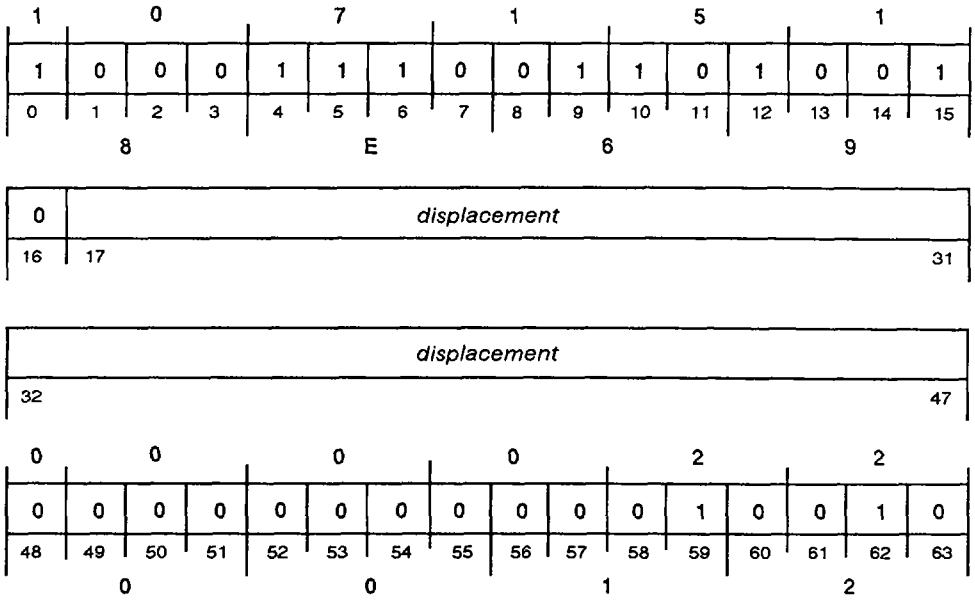
Example

```
WGRDATTR
```

Read Palette WGRDPAL

Privileged Graphics Instruction

WGRDPAL *displacement*



Function: palette entry --> packet

Parameters: AC0 = Palette index --> unchanged
AC1 = Microcode ID for video board --> unchanged
AC2 = Address of palette entry packet --> unchanged

WGRDPAL reads an entry from the palette specified by AC1 into the packet specified by AC2. The values read from the palette may not be the values originally written to the palette.

Arguments

displacement Nonindirectable PC-relative offset to runtime routine that performs the **WGRDPAL** function (if hardware support is currently unavailable). Effective address generated by instruction confined to the current segment.

Registers, Flags, and Stacks

AC0 Before execution, contains palette index.

After execution, contents unchanged.

AC1 Before execution, contains microcode ID value for video board.

After execution, contents unchanged.

AC2 Before execution, contains address of a palette entry packet.

After execution, contents unchanged.

Palette entry packet is set of eight unsigned 32–bit integers, left justified, as follows:

Double Word #	Mnemonic	Description
1	RED_P0	Phase 0 red intensity.
2	GREEN_P0	Phase 0 green intensity.
3	BLUE_P0	Phase 0 blue intensity.
4	GRAY_P0	Phase 0 gray–scale intensity.
5	RED_P1	Phase 1 red intensity.
6	GREEN_P1	Phase 1 green intensity.
7	BLUE_P1	Phase 1 blue intensity.
8	GRAY_P1	Phase 1 gray–scale intensity.

0 = lowest intensity
$FFFFFFFF_{16}$ = highest intensity

AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 4
PSR	Unchanged
Stack	Unchanged

Related Instructions

CONFIG Use this instruction to obtain the microcode ID value of the video board.

LLEF Use this instruction to place logical address into AC2.

Load with immediate
Use these instructions to place the appropriate value into AC0.

LPSHJ, WPOPJ
If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a nonindirectable PC–relative offset). The processor uses E as the address of a runtime routine that performs the **WGRDPAL** function. The **WPOPJ** instruction exits from this software emulator.

Exceptions

WGRDPAL returns the values as stored in the video board. It may not necessarily return the same value that was written to it by a **WGWRPAL** instruction.

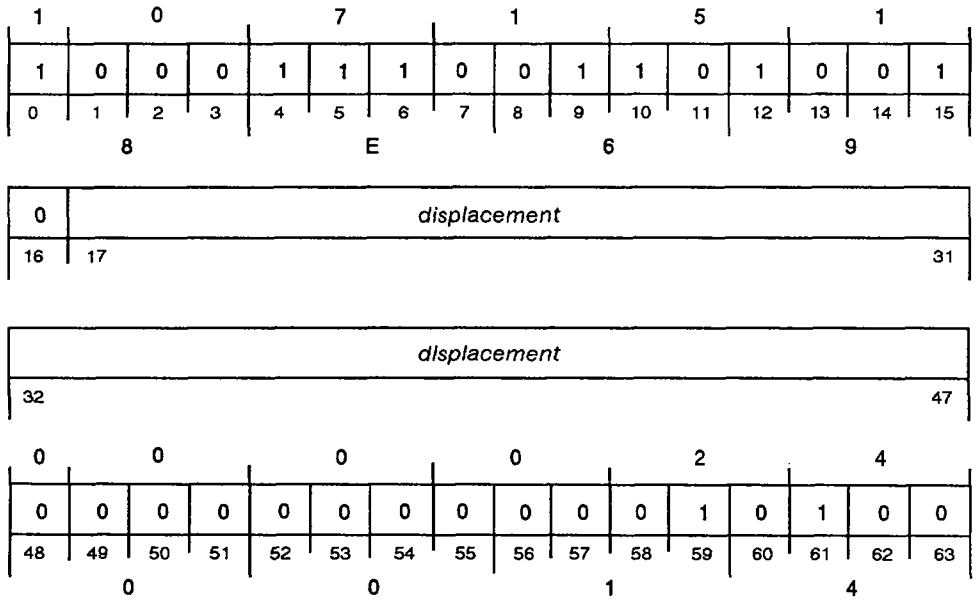
Example

```
WGRDPAL
```

Read Pixel WGRDPIXL

Graphics Instruction

WGRDPIXL *displacement*



Function: pixel value AND form mask --> AC0

Parameters: AC0 = ? --> pixel value
AC1 = Form ID --> unchanged
AC2 = Address of pixel packet --> unchanged

WGRDPIXL reads a single pixel from a form. The pixel value is logically ANDed with the form's form mask, and placed in AC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime routine that performs the **WGRDPIXL** function (if hardware support is currently unavailable). Effective address generated by instruction confined to the current segment.

Registers, Flags, and Stacks

AC0 After execution, if pixel with given coordinates is in specified form, contains 32-bit pixel value, right-justified and zero-extended; otherwise unchanged.

AC1 Before execution, contains 32-bit form ID.

After execution, contents unchanged.

AC2 Before execution, contains address of pixel packet.

After execution, contents unchanged.

Packet consists of two signed, 32-bit integers as follows:

Double Word #	Contents
1	X coordinate of pixel.
2	Y coordinate of pixel.

AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 4
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load with immediate
: Use these instructions to place the appropriate value into AC0.

Load effective address
: Use these instructions to place the packet address into AC2.

LPSHJ, WPOPJ
: If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a nonindirectable PC–relative offset). The processor uses E as the address of a runtime routine that performs the **WGRDPIXL** function. The **WPOPJ** instruction exits from this software emulator.

Exceptions

If the specified X and Y are outside the range of the form, AC0 is unchanged.

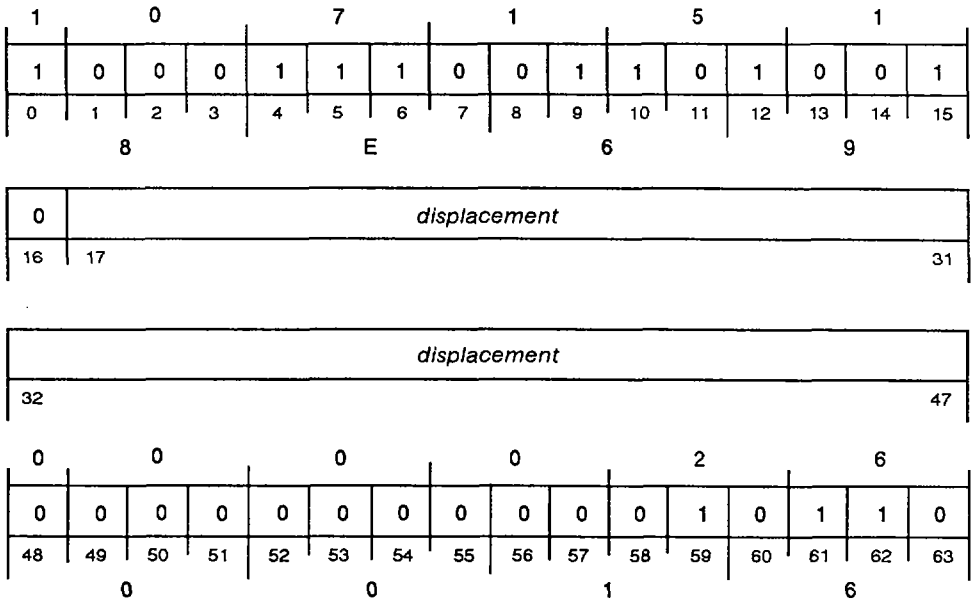
Example

```
WGRDPIXL
```

Fill Rectangle WGRFLOOD

Graphics Instruction

WGRFLOOD *displacement*



Function: Sets rectangle colors (combination rule & operation mask & form mask)

Parameters: AC0 = Color (pixel value) to write --> unchanged
AC1 = Form ID --> unchanged
AC2 = Address of **WGRFLOOD** packet --> unchanged

WGRFLOOD sets all pixels in a rectangular area to a specified color. The actual value written to the form is controlled by the form's combination rule, operation mask, and form mask.

Arguments

displacement Nonindirectable PC-relative offset to runtime routine that performs the **WGRFLOOD** function (if hardware support is currently unavailable). Effective address generated by instruction confined to the current segment.

Registers, Flags, and Stacks

AC0 Before execution, contains right-justified, 32-bit integer specifying color (pixel value) to write.

After execution, contents unchanged.

AC1 Before execution, contains 32-bit form ID.

After execution, contents unchanged.

AC2 Before execution, contains address of **WGRFLOOD** packet.

After execution, contents unchanged.

Packet consists of four 32-bit integers as follows:

Double Word #	Contents
1	X coordinate of rectangle's ULC (signed).
2	Y coordinate of rectangle's ULC (signed)
3	Width of rectangle in pixels (unsigned).
4	Height of rectangle in pixels (unsigned).

AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 4
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load with immediate
: Use these instructions to place the appropriate values into AC0 and AC1.

Load effective address
: Use these instructions to place the packet address into AC2.

LPSHJ, WPOPJ
: If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a nonindirectable PC-relative offset). The processor uses E as the address of a runtime routine that performs the **WGRFLOOD** function. The **WPOPJ** instruction exits from this software emulator.

Exceptions

WGRFLOOD has no effect on pixels lying outside the form or on pixels on the form that are write inhibited.

If either the width or the height of the rectangle (double word 3 or 4 in the packet) is specified as zero, **WGRFLOOD** has no effect.

If an overdraw condition occurs, the PSR(OVR) bit is set to one.

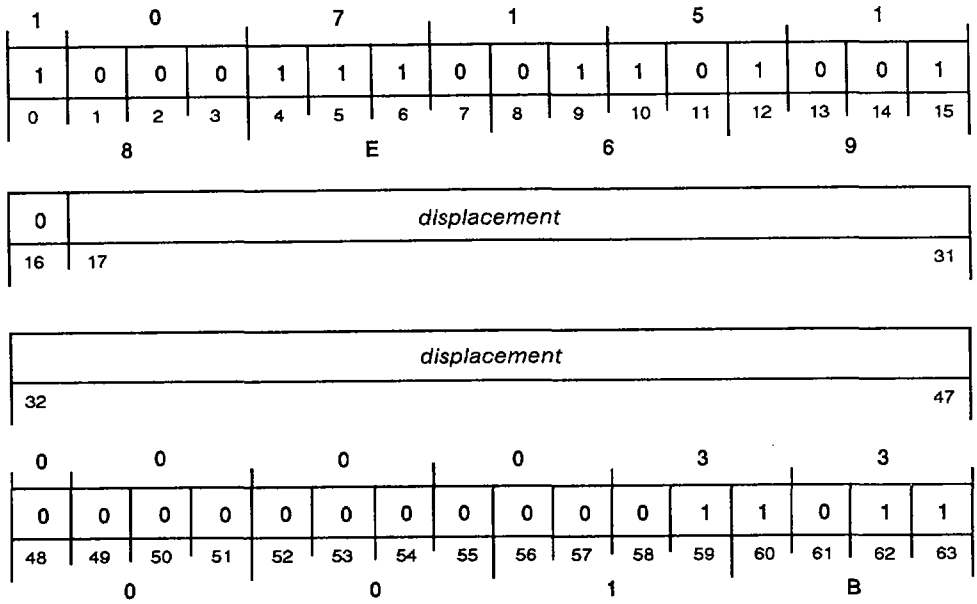
Example

```
WGRFLOOD
```

Write Attribute WGWRATTR

Graphics Instruction

WGWRATTR *displacement*



Function: write attribute to form

Parameters: AC0 = Attribute index --> unchanged
AC1 = Form ID --> unchanged
AC2 = Address of new attribute value --> unchanged

WGWRATTR writes one of the attributes (indexed by AC0) to the specified form descriptor with the value at the address specified by AC2.

Arguments

displacement Nonindirectable PC-relative offset to runtime routine that performs the **WGWRATTR** function (if hardware support is currently unavailable). Effective address generated by instruction confined to the current segment.

Registers, Flags, and Stacks

AC0 Before execution, contains one of the following attribute numbers:

Attribute Number	Contents
0	Operation mask
1	Combination rule
2	Line control word
3	Line foreground color
4	Line background color
5	Line style
6	Character control word
7	Character foreground color
8	Character background color

After execution, contents unchanged.

AC1	Before execution, contains 32-bit form ID. After execution, contents unchanged.
AC2	Before execution, contains address of new attribute value. After execution, contents unchanged.
AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 4
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load with immediate
: Use these instructions to place the appropriate values into AC0 and AC1.

Load effective address
: Use these instructions to place the address into AC2.

LPSHJ, WPOPJ
: If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a nonindirectable PC-relative offset). The processor uses E as the address of a runtime routine that performs the **WGWRATTR** function. The **WPOPJ** instruction exits from this software emulator.

Exceptions

If the value in AC0 is greater than 8_{10}, an invalid attribute index fault will occur.

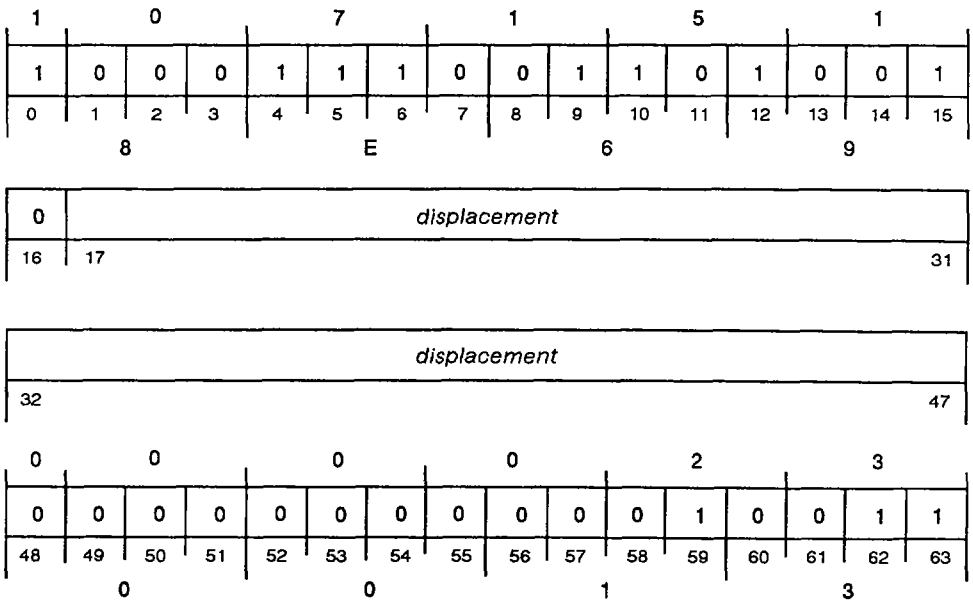
Example

```
WGWRATTR
```

Write Palette WGWRPAL

Privileged Graphics Instruction

WGWRPAL *displacement*



Function: packet entry --> palette

Parameters: AC0 = Palette index --> unchanged
AC1 = Microcode ID for video board --> unchanged
AC2 = Address of palette entry packet --> unchanged

WGWRPAL writes an entry into the packet, specified by AC1, from the packet specified by AC2. The palette entry is specified by AC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime routine that performs the **WGWRPAL** function (if hardware support is currently unavailable). Effective address generated by instruction confined to current segment.

Registers, Flags, and Stacks

AC0 Before execution, contains palette index.

After execution, contents unchanged.

AC1 Before execution, contains microcode ID value for video board.

After execution, contents unchanged.

AC2 Before execution, contains address of a palette entry packet.

After execution, contents unchanged.

Palette packet is set of eight unsigned 32-bit integers, left justified, as follows:

Double Word #	Mnemonic	Description
1	RED_P0	Phase 0 red intensity.
2	GREEN_P0	Phase 0 green intensity.
3	BLUE_P0	Phase 0 blue intensity.
4	GRAY_P0	Phase 0 gray-scale intensity.
5	RED_P1	Phase 1 red intensity.
6	GREEN_P1	Phase 1 green intensity.
7	BLUE_P1	Phase 1 blue intensity.
8	GRAY_P1	Phase 1 gray-scale intensity.

0 = lowest intensity
$FFFFFFFF_{16}$ = highest intensity

AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 4
PSR	Unchanged
Stack	Unchanged

Related Instructions

CONFIG Use this instruction to place microcode ID into AC1.

LLEF Use this instruction to place logical address into AC2.

Load with immediate
Use these instructions to place the appropriate value into AC0.

LPSHJ, WPOPJ
If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a nonindirectable PC-relative offset). The processor uses E as the address of a runtime routine that performs the **WGWRPAL** function. The **WPOPJ** instruction exits from this software emulator.

Exceptions

None

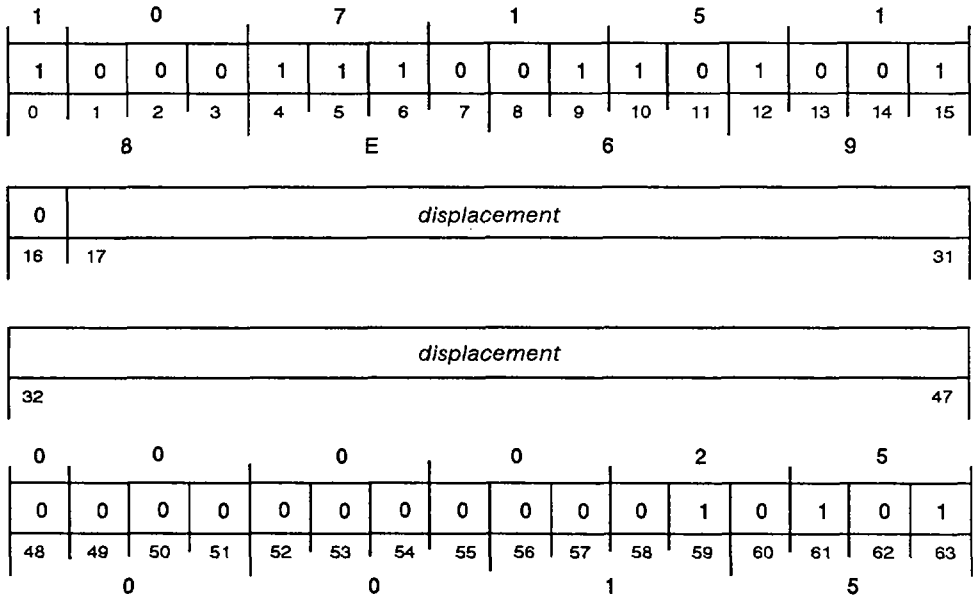
Example

```
WGWRPAL
```

Write Pixel WGWRPIXL

Graphics Instruction

WGWRPIXL *displacement*



Function: pixel value (combination rule & operation mask & form mask) --> form

Parameters: AC0 = pixel value --> unchanged
AC1 = Form ID --> unchanged
AC2 = Address of pixel packet --> unchanged

WGWRPIXL writes the value in AC0 into a pixel of the associated form. The actual value written to the form is controlled by the form's combination rule, operation mask, and form mask.

Arguments

displacement Nonindirectable PC-relative offset to runtime routine that performs the **WGWRPIXL** function (if hardware support is currently unavailable). Effective address generated by instruction confined to current segment.

Registers, Flags, and Stacks

AC0 Before execution, contains right-justified 32-bit pixel value.

After execution, contents unchanged.

AC1 Before execution, contains 32-bit form ID.

After execution, contents unchanged.

AC2 Before execution, contains address of pixel packet.

After execution, contents unchanged.

Packet consists of two signed, 32-bit integers as follows:

Double Word #	Contents
1	X coordinate of pixel.
2	Y coordinate of pixel.

AC3	Unused
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 4
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load with immediate
: Use these instructions to place the appropriate value into AC0 and AC1.

Load effective address
: Use these instructions to place the packet address into AC2.

LPSHJ, WPOPJ
: If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address E (using the *displacement* as a nonindirectable PC-relative offset). The processor uses E as the address of a runtime routine that performs the **WGWRPIXL** function. The **WPOPJ** instruction exits from this software emulator.

Exceptions

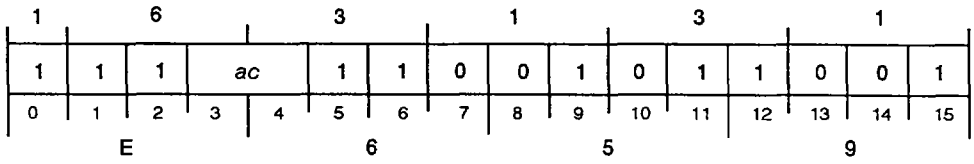
If the pixel specified by (X,Y) is write-inhibited or lies outside the form, **WGWRPIXL** has no effect.

Example

```
WGWRPIXL
```

Wide Halve WHLV

WHLV *ac*



Function: *ac* / 2 -> *ac*

Parameters: None

NOTE: **WHLV** rounds toward 0

WHLV divides the signed 32-bit integer in the specified accumulator by two and rounds the result toward zero.

Arguments

ac	Before execution, contains signed 32-bit integer.
	After execution, contains result.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

HLV	Halve

Exceptions

None

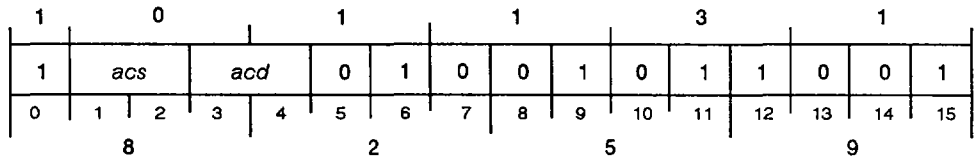
Example

```
;Subroutine to compare two strings
;
;Strings are assumed to be word aligned and to be followed by a
;terminating null (or two, if needed to fill a word).
;
;       AC0 = Byte length of string (without terminator)
;       AC1 = Word pointer to first string
;       AC2 = Word pointer to second string
;
;Returns +1 if they match, 0 if they don't
```

```
CMPAR:   WPSH        3,3          ;Save return address
         LDAFP       3            ;Get frame pointer
         WINC        0,0          ;Get number of words with terminator
         WINC        0,0          ;Number of characters plus 1
         WHLV        0            ;Number of characters plus 1 / 2
         XNSTA       0,WCNT,3     ;Save count
         XWSTA       1,WPTR.W,3   ;Save one of the pointers
CMPLP    XNLDA       0,0,2        ;Pick up a word
         XNLDA       1,@WPTR.W,3  ;and its friend
         WSEQ        0,1          ;See if equal
         WPOPJ                    ;No, return false (0)
         XWISZ       WPTR.W,3     ;Move to next word
         WINC        2,2          ;(S)
         XNDSZ       WCNT,3       ;See if done
         WBR CMPLP                ;
         ISZTS                    ;Bump return (they match)
         WPOPJ                    ;Return
```

Wide Increment WINC

WINC *acs,acd*



Function: *acs* + 1 -> *acd*

Parameters: None

WINC increments the 32-bit contents of *acs* by 1 and loads the result into *acd*.

Arguments

acs	Before execution, contains 32-bit value.
	After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd	After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Set with value of ALU CARRY
Overflow	1 if ALU overflow
PC	PC + 1
PSR	OVR set to 1 if overflow occurs
Stack	Unchanged

Related Instructions

INC	Increment

Exceptions

None

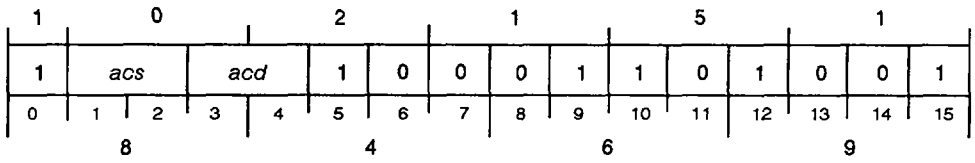
Example

```
;Subroutine to compare two strings
;
;Strings are assumed to be word aligned and to be followed by a
;terminating null (or two, if needed to fill a word).
;
;       AC0 = Byte length of string (without terminator)
;       AC1 = Word pointer to first string
;       AC2 = Word pointer to second string
;
;Returns +1 if they match, 0 if they don't
```

```
CMPAR:   WPSH        3,3           ;Save return address
         LDAFP       3             ;Get frame pointer
         WINC        0,0           ;Get number of words with terminator
         WINC        0,0           ;Number of characters plus 1
         WHLV        0             ;Number of characters plus 1 / 2
         XNSTA       0,WCNT,3      ;Save count
         XWSTA       1,WPTR.W,3    ;Save one of the pointers
CMPLP    XNLDA       0,0,2         ;Pick up a word
         XNLDA       1,@WPTR.W,3   ;and its friend
         WSEQ        0,1           ;See if equal
         WPOPJ                     ;No, return false (0)
         XWISZ       WPTR.W,3      ;Move to next word
         WINC        2,2           ;(S)
         XNDSZ       WCNT,3        ;See if done
         WBR CMPLP                 ;
         ISZTS                     ;Bump return (they match)
         WPOPJ                     ;Return
```

Wide Inclusive OR WIOR

WIOR *acs,acd*



Function: *acs* OR *acd* -> *acd*

Parameters: None

WIOR forms the logical Inclusive OR of corresponding bits of *acs* and *acd*. Sets a bit position in the result to 1 if the corresponding bit position in one or both operands contains a 1; otherwise sets the result bit to 0.

Arguments

acs	Before execution, contains 32-bit value.
	After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd	Before execution, contains 32-bit value.
	After execution, contains result.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

IOR	Inclusive OR
XOR	Exclusive OR
WXOR	Wide Exclusive OR

Exceptions

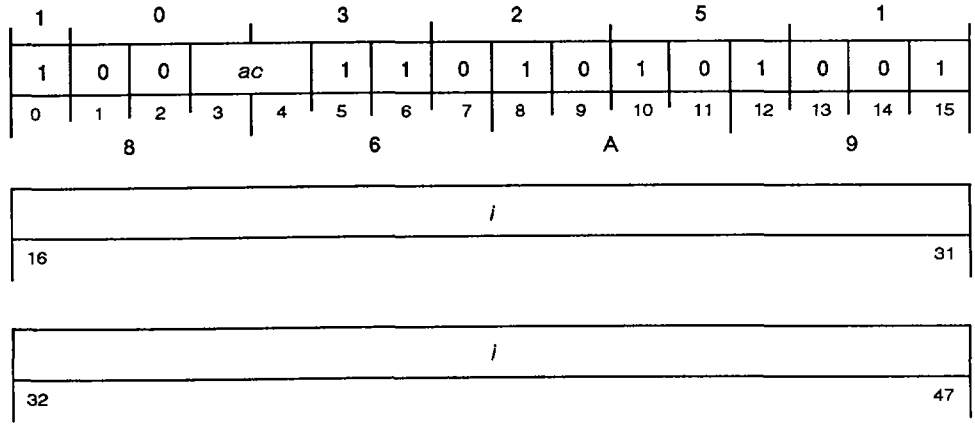
None

Example

```
WIOR  0,1          ;Inclusive OR the contents of AC0 and AC1.
                   ;Result goes to AC1
```

Wide Inclusive OR Immediate WIORI

WIORI *i,ac*



Function: *i* OR *ac* -> *ac*

Parameters: None

WIORI forms the logical inclusive OR of the contents of the immediate field and the contents of the specified accumulator, placing the result in the specified accumulator.

Arguments

i	32-bit value.
ac	Before execution, contains 32-bit value.
	After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 3
PSR	Unchanged
Stack	Unchanged

Related Instructions

IORI	Inclusive OR Immediate

Exceptions

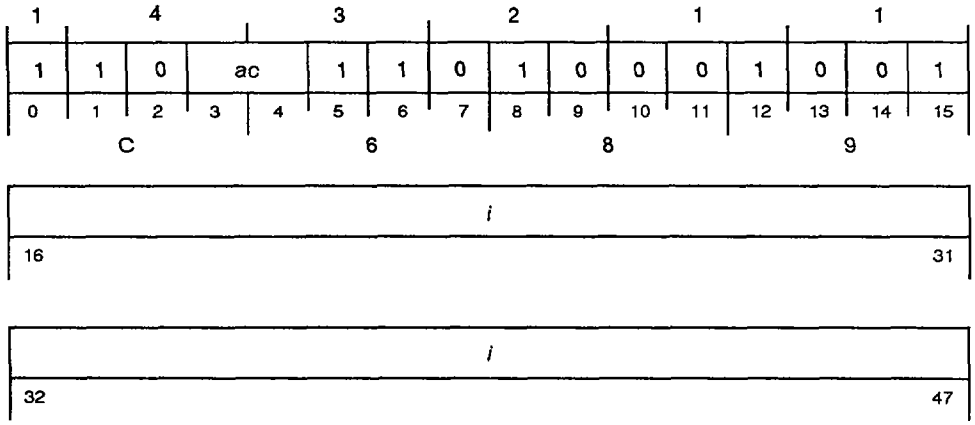
None

Example

```
WIORI 016000000000,0          ;Force ring bits in AC0 to be ring 7.
```

Wide Load with Wide Immediate WLDAI

WLDAI *i,ac*



Function: *i -> ac*

Parameters: None

WLDAI loads an accumulator with a 32-bit immediate value.

Arguments

i	32-bit value
ac	After execution, contains result

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 3
PSR	Unchanged
Stack	Unchanged

Related Instructions

NLDAI	Narrow Load Immediate

Exceptions

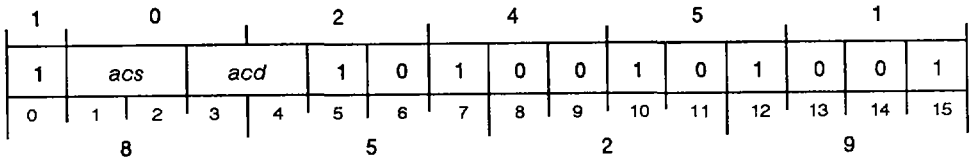
None

Example

```
WLDAI 016000000000,2          ;Put a ring 7 offset 0 address in AC2.
```

Wide Load Byte WLDB

WLDB *acs,acd*



Function: (E)byte --> *acd* [bits 24-31, bits 16-23 set to 0]

Parameters: *acs* = byte pointer --> unchanged

WLDB uses the byte address contained in *acs* to load a byte from memory into *acd*.

Arguments

acs Before execution, contains byte address.

After execution, contents unchanged unless *acs* and *acd* are same accumulator.

acd(24-31) After execution, contains byte with bits 0-23 set to 0.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stacks	Unchanged

Related Instructions

LLEFB, XLEFB

Use these instructions to load an effective byte address into *acs*.

Exceptions

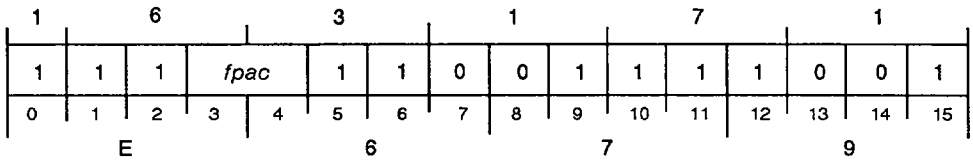
None

Example

```
    ;Convert lower case input to upper case.
    ;
    ;       AC0 = Byte pointer to string
CNVUC:  WLDB        0,2         ;Put a byte of source string into AC2
        WANDI       177,2       ;Mask to seven bits
        WCLM        2,2         ;See if lower case
    .DWORD "A+40                ;Lower limit for compare
    .DWORD "Z+40                ;Upper limit for compare
        WBR NOTLOW              ;Not lower case
        WNADI       -40,2       ;Yes, lower case, convert to upper
```

Wide Load Integer WLDI

WLDI *fpac*



Function: @(AC3)[decimal #] -> *fpac*[normalized floating-point #]
AC3 -> AC2
update -> FPSR(N,Z)

Parameters: AC1 = data-type indicator -> unchanged
AC2 = x -> AC3
AC3 = byte pointer -> last byte pointer + 1

NOTE: A -0 sets *fpac* to True Zero

WLDI fetches a decimal integer (up to 16 digits) from the specified memory location, translates the integer to normalized floating-point format, and loads the result into the specified floating-point accumulator.

For data type 7, the first byte of the number stored must contain the sign and exponent of the floating-point number. The instruction copies each byte (following the lead byte) directly to mantissa of the specified *fpac*.

It then sets to zero each low-order byte in the *fpac* that does not receive data from memory.

Arguments

fpac	After execution, contains translated floating-point integer from specified memory address. Unused lower-order byte locations set to 0.

Registers, Flags, and Stacks

AC0	Unused
AC1	Before execution, contains data type and length of integer to be translated. **WLDI** does not use the scale factor in the data type indicator.
	After execution, contents unchanged.
AC2	After execution, contains initial value of AC3.
AC3	Before execution, contains starting byte address for location in memory.
	After execution, contains address of first byte following integer field.
FPAC0 -FPAC3	Can be individually specified for *fpac*; otherwise not used.
FPSR	Updated Z and N flags.
PC	PC + 1
Stack	Unchanged

Related Instructions

LDI	Load Integer
LDIX	Load Integer Extended
WLDIX	Wide Load Integer Extended

Exceptions

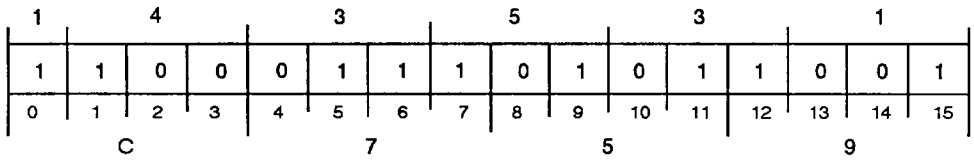
An attempt to load a -0 sets the *fpac* to true zero.

Example

```
XNLDA  1,DTYPE      ;AC1 contains the data type indicator.
XLEF   3,INTEG      ;Word pointer to the integer field.
WADD   3,3          ;AC3 is a byte pointer to the integer.
WLDI   2            ;Convert the specified commercial integer
                    ;to a floating point number in FPAC2.
```

Wide Load Integer Extended WLDIX

WLDIX



Function: @(AC3)[decimal #] -> (FPAC0,1,2,3)[floating-point #]
AC3 -> AC2
? -> FPSR(N,Z)

Parameters: AC1 = data-type indicator -> unchanged
AC2 = x -> AC3
AC3 = byte pointer -> last byte pointer +1

WLDIX fetches a decimal integer from specified memory locations, expands the integer to 32 digits, divides the result into four 8-digit integers, converts the integers to floating-point numbers, and loads them in individual floating-point accumulators.

The sign of the 32-bit integer is stored in each fpac unless the integer in that fpac consists of all zeros, in which case the floating-point accumulator is set to true zero.

The integer fetched from memory must be of data type 0, 1, 2, 3, 4, or 5 and can contain up to 32 digits. The integer is expanded to 32 digits by adding zeros to the high-order bytes.

Arguments

None

Registers, Flags, and Stacks

AC0	Unused
AC1	Before execution, contains data type and length of integer to be translated. **WLDIX** does not use the scale factor in the data type indicator.
	After execution, contents unchanged.
AC2	After execution, contains initial value of AC3.
AC3	Before execution, contains starting byte address for location in memory.
	After execution, contains address of first byte following integer field.
FPAC0	After execution, contains first 8 (high-order) digits from extended integer.
FPAC1	After execution, contains second 8 digits from extended integer.
FPAC2	After execution, contains third 8 digits from extended integer.
FPAC3	After execution, contains last 8 (low-order) digits from extended integer.

FPSR	Z and N flags unpredictable.
Overflow	0
PC	PC + 1
Stack	Unchanged

Related Instructions

LDI	Load Integer
LDIX	Load Integer Extended
WLDI	Wide Load Integer

Exceptions

None

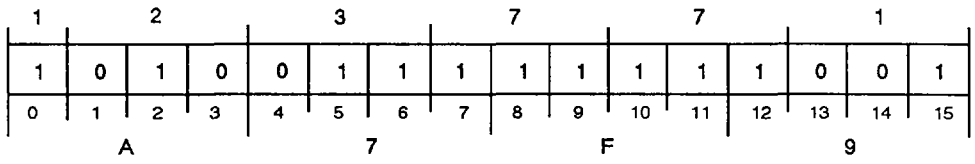
Example

```
XNLDA  1,DTYPE      ;AC1 contains the data type indicator.
XLEF   3,INTEG      ;Word pointer to the integer field.
WADD   3,3          ;AC3 is a byte pointer to the integer.
WLDIX               ;Convert the specified commercial integer
                    ;to 4 floating point numbers, distributed
                    ;among all of the FPACs.
```

Wide Load Map WLMP

Privileged Instruction

WLMP



Function: @(AC2) -> map slots

Parameters: AC0 = #(1st slot #) -> #(last slot # + 1)
AC1 = #(# slots) -> 0
AC2 = map slot -> last map slot + 2

WLMP, in conjunction with three accumulators, loads successive double words from memory into successive Data Channel (DCH) or Burst Multiplexor Channel (BMC) map slots.

Arguments

None

Register, Flags, and Stacks

AC0(21–31) Before execution, contains map slot number of first map slot to be loaded. $0–1777_8$ indicate BMC slots; $2177_8–2777_8$ indicate DCH slots.

If machine implements more than one I/O channel, AC0(18–20) contains channel number.

For each map slot loaded, AC0 is incremented by one.

After execution, references map slot following last slot loaded.

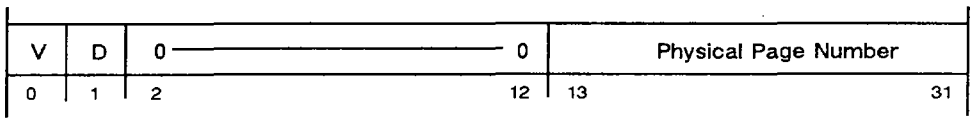
AC1(16–31) Before execution, contains unsigned 16-bit integer specifying count of number of map slots to be loaded.

For each map slot loaded, AC1 is decremented by one.

After execution, contains 0.

AC2 Before execution, contains effective address of first double word to be loaded into referenced slots.

Contents of double words are in following format:



Bits	Name	Contents or Function
0	V	Valid Set to 0: valid Set to 1: access denied
1	D	Data Set to 0: transfer data Set to 1: transfer zeros
2–12	0--0	Must be set to 0
13–31	Physical Page Number	Physical page containing first double word to be involved in I/O operation (unused bits must be 0).

Effects of setting V and D bits and direction of transfer:

V	D	Transfer Direction	Action
0	0	From I/O Port	Transfer data
0	1	From I/O Port	Transfer zeros from either BMC or DCH device
1	--	From I/O Port	Transfer aborted -- flag error to device
0	0	To I/O Port	Transfer data
0	1	To I/O Port	Transfer zeros to either BMC or DCH device
1	--	To I/O Port	Transfer aborted -- flag error to device

From I/O Port implies memory to device;
To I/O Port implies device to memory.

For each map slot loaded, AC2 is incremented by two.

After execution, contains address of word following last double word loaded.

AC3 Unused

CARRY Unchanged

Overflow 0

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

Load with immediate
: Use these instructions to place the appropriate values into AC0 and AC1.

Load effective address
: Use these instructions to load an address into AC2.

Exceptions

If AC1 is initially 0, the instruction becomes a no-op.

Upon detection of an invalid map entry due to an active device:

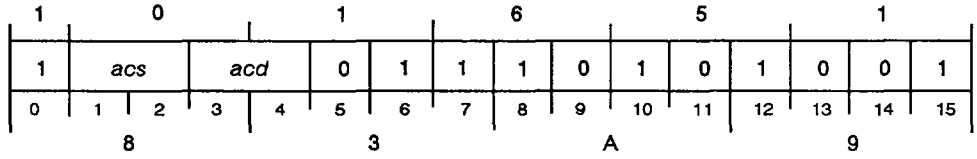
BMC	Active BMC requesting device flagged
DCH	Bit 4 of IOC Status Register set to 1

Example

```
XWLDA  0,SLOT_NUM              ;Get the starting slot number.
NLDAI  4,1                     ;Load 4 map slots.
XLEF   2,SLOT_CONTENTS         ;Get address of data to load.
WLMP                           ;Load all 4 map slots.
       ...
SLOT_CONTENTS:
       .DWORD      0           ;Data for 4 map slots.
       .DWORD      0
       .DWORD      0
       .DWORD      0
```

Wide Locate Lead Bit WLOB

WLOB *acs,acd*



Function: *acs*(# of leading 0s) + *acd* -> *acd*

Parameters: None

WLOB counts the high-order zeros in *acs* and performs an unsigned add of the count to the 32-bit integer in *acd*.

Arguments

acs	Before execution, contains 32-bit value.
	After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd	Before execution, contains 32-bit integer.
	After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

LOB	Locate Lead Bit
LRB, WLRB	Locate the lead bit and reset it.

Exceptions

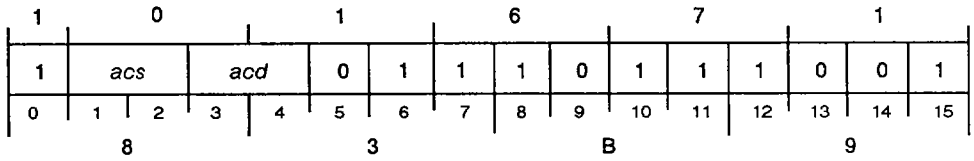
None

Example

```
NLDAI  0777,1        ;A bit pattern into AC1.
WSUB   0,0           ;Set AC0 to zero.
WLOB   1,0           ;Adds 23 to AC0. AC0 now contains 23.
```

Wide Locate and Reset Lead Bit **WLRB**

WLRB *acs,acd*



Function: *acs*(# of leading 0s) + *acd* -> *acd* 0 -> high 1(*acs*)

Parameters: None

NOTE: If *acs* is *acd*; then nothing is added, but 0 -> leading 1

WLRB counts the high-order zeros in *acs* and performs an unsigned add of the count to the 32-bit integer in *acd*. It then sets the leading 1 bit in *acs* to 0.

Arguments

acs	Before execution, contains 32-bit value.
	After execution, leading bit set to 0.
	If *acs* equals *acd*, **WLRB** sets the leading bit to 0 and adds nothing to the contents of *acs*.
acd	Before execution, contains 32-bit integer.
	After execution, contains *acd* value plus number of leading zeros in *acs*.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

LRB	Locate and Reset Lead Bit
LOB, WLOB	Locate the lead bit in an accumulator.

Exceptions

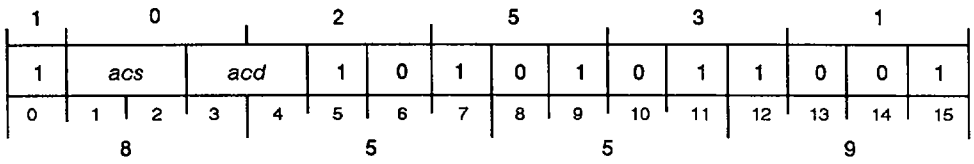
None

Example

```
NLDAI  0777,1        ;A bit pattern into AC1.
WSUB   0,0           ;Set AC0 to zero.
WLRB   1,0           ;Adds 23 to AC0. AC0 now contains 23.
                     ;AC1 now contains 377₈.
```

Wide Logical Shift WLSH

WLSH *acs,acd*



Function: shift *acd*(*acs*(bits 24–31[+ = left,– = right])) -> *acd*

Parameters: None

WLSH shifts the contents of *acd* either left or right, depending on the value contained in *acs*. Bits shifted out are lost; zeros fill the vacated bit positions.

Arguments

acs(24–31) Before execution, contains signed 8-bit integer. Number of bits shifted equal to magnitude. (Bits 0–23 are ignored.)

If bit 24 is 0 (positive), contents of *acd* shifted left; if bit 24 is 1(negative), contents of *acd* shifted right. If zero, no shifting occurs.

After execution, contents unchanged unless *acs* and *acd* are same accumulator.

acd Before execution, contains 32-bit value.

After execution, contains result.

Registers, Flags, and Stacks

AC0–AC3 Can be individually specified as *acs* and *acd*; otherwise unused.

CARRY Unchanged

Overflow 0

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

LSH Logical Shift

Exceptions

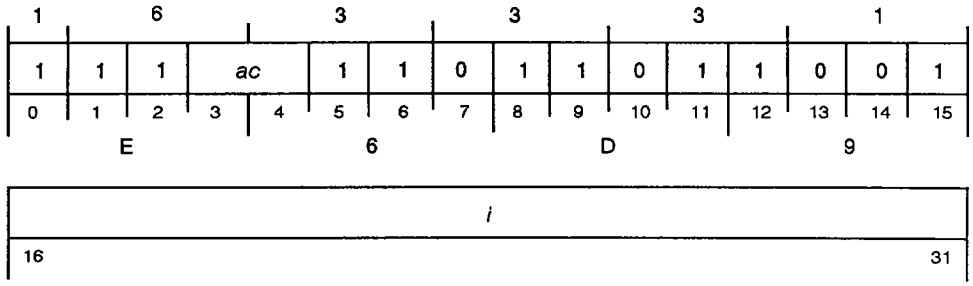
None

Example

```
NLDAI  -4,3          ;Load a -4 into AC3, then
WLSH   3,1           ;Divide contents of AC1 by 16
```

Wide Logical Shift With Narrow Immediate WLSHI

WLSHI *i,ac*



Function: shift *ac*(*i*[+ = left,− = right]) -> *ac*

Parameters: None

WLSHI shifts the 32-bit contents of an accumulator either left or right. The bits shifted out are lost, and zeros fill the vacated bit positions.

Arguments

i(24-31) Signed 8-bit integer specifying number of bits to shift and direction of shifting. Bits 16-23 must be identical to bit 24 (sign bit); otherwise, results indeterminate. Processor sign-extends this value to 32 bits. If bit 24 is 0 (*i* is positive, in range 1 to 32_{10}), **WLSHI** shifts contents of *ac* left.

If bit 24 is 1 (*i* is negative, in range −1 to -32_{10}), **WLSHI** shifts contents of *ac* right.

If *i* is zero, no shifting occurs.

ac Before execution, contains 32-bit value.

After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

Load accumulator

Use these instructions to place value to be shifted into *ac*.

WLSI Wide Logical Shift Immediate

Exceptions

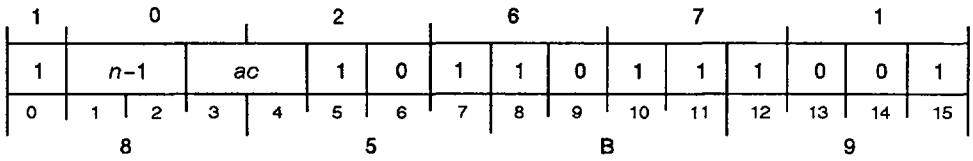
None

Example

```
XLEF  0,VARIABLE   ;Get the address of the variable.
WANDI 01777777777,0;Mask to just the offset portion.
WLSHI -10.,0       ;Right shift to get the page number.
```

Wide Logical Shift Immediate WLSI

WLSI *n,ac*



Function: shift *ac* left(*n*) -> *ac*

Parameters: None

WLSI shifts the contents of the specified accumulator left the number of bits indicated by an immediate value (*n*).

Arguments

n	Integer in range 1–4. Assembler takes coded value of *n* and subtracts one from it before placing it in immediate field. Thus, programmer should code exact value to be shifted.
ac	Before execution, contains 32–bit value. After execution, contains result.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

WLSHI	Wide Logical Shift with Narrow Immediate
WMOVR	Wide Move Right

Exceptions

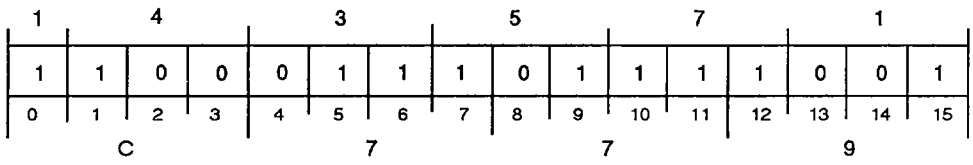
None

Example

```
WLSI  3,3          ;Multiply contents of AC3 by 8
```

Wide Load Sign WLSN

WLSN



Function: (E)[decimal #] = (non0 or 0, + or −)
AC3 -> AC2

Parameters: AC1 = x -> result
+1 (+ non0)
−1 (−non0)
0 (+0)
−2 (−0)
AC3 = byte pointer -> ?

WLSN evaluates a decimal number in the specified memory location and returns a code that classifies the number as zero or nonzero and identifies its sign.

Arguments

None

Registers, Flags, and Stacks

AC0 Unused

AC1 Before execution, specifies data type and length of number to be evaluated.

After execution, contains returned value code as follows:

Value of Number	Code
Positive nonzero	+1
Negative nonzero	−1
Positive zero	0
Negative zero	−2

AC2 After execution, contains original contents of AC3.

AC3 Before execution, contains logical address of byte to be evaluated.

After execution, contents undefined.

CARRY Unchanged

Overflow Unaffected

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

LSN Load Sign

Exceptions

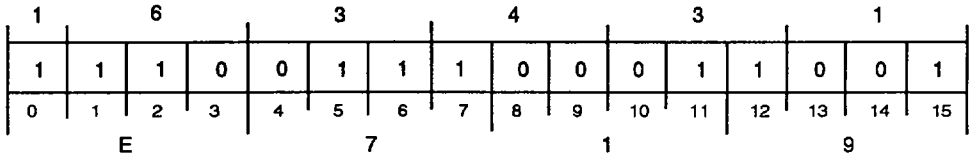
None

Example

```
XNLDA  1,TYPE       ;AC1 contains the data type indicator.
XLEF   3,INTEG      ;Word pointer to the integer field.
WADD   3,3          ;AC3 is a byte pointer to the integer.
WLSN                ;Get a code into AC1 that reflects the
                    ;sign of the commercial integer.
```

Wide Mask, Store and Skip if Equal WMESS

WMESS
(unsuccessful return)
(successful return)



Function:
If (((@AC2) XOR AC0) AND AC3) = 0
Then @(AC2) <-> AC1
Skip unsuccessful return
Else
@(AC2) --> AC1

Parameters:
AC0 = # comparison --> unchanged
AC1 = swap bits --> (E)
AC2 = address --> unchanged
AC3 = mask --> unchanged

WMESS tests and sets multiple bits of a double word in memory. The instruction reads the double word addressed by AC2 and then performs an exclusive OR of the double word with the contents of AC0. If all resultant bits that equal 1 from the exclusive OR correspond to bits that equal 0 in AC3, the comparison is successful. If any bit resulting from the exclusive OR equals 1, and it corresponds to a bit that equals 1 in AC3, the comparison is unsuccessful. The processor exchanges the values in AC1 with the double word in memory.

WMESS is guaranteed to be indivisible when the memory word is double-word aligned.

Arguments

None

Registers, Flags, and Stacks

AC0	Before execution, contains 32 bits that processor compares (exclusive OR) with 32 bits in memory.
	After execution, contents unchanged.
AC1	Before execution, contains 32 bits that processor exchanges with 32 bits in memory.
	After execution, always contains initial 32 bits of double word addressed by AC2.
AC2	Before execution, contains address of data element to test; no indirect addressing.
	After execution, contents unchanged.
AC3	Before execution, contains 32 bits that processor compares (logical AND) with results of exclusive OR.
	After execution, contents unchanged.
CARRY	Unchanged
Overflow	0

PC	PC + 1 (unsuccessful return) If any bit resulting from exclusive OR equals 1 and it corresponds to bit that equals 1 in AC3. PC + 2 (successful return) If all resultant bits that equal 1 from exclusive OR correspond to bits that equal 0 in AC3.
PSR	Unchanged
Stack	Unchanged

Related Instructions

LLEF, XLEF	Use a load effective address instruction to calculate and load the double-word memory address into AC2.
WBR	Use the Wide Branch instruction to jump to the code for handling the unsuccessful return.

Exceptions

None

Example

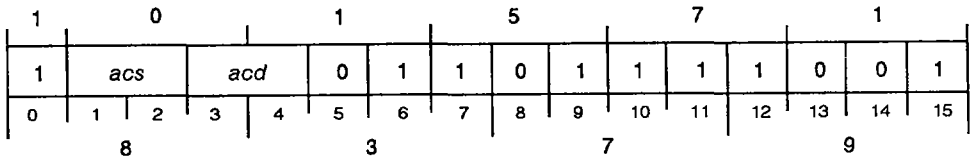
```
    XWLDA 0,COM         ;Load AC0 with value for XOR comparison
    XWLDA 1,SETBTS      ;Load AC1 with the value to store in memory
    XLEF  2,BITEST      ;Load AC2 with the address of the double word
                        ;to test
    XWLDA 3,MASK        ;Load AC3 with mask bits for the AND
    WMESS               ;comparison
    WBR   NOGOOD        ;Invalid comparison
          .             ;Valid comparison
    ...                 ;
```

WMESS first performs an exclusive OR between the bits addressed by BITEST and the bits in AC0. The instruction then performs a logical AND between the XOR result and the bits in AC3. If the final result equals zero, the instruction exchanges the bits in AC1 with the bits in memory and executes the instruction following the WBR instruction. If the final result is not zero, AC1 is loaded with the bits in memory and WBR is executed.

```
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;The following example shows how WMESS can be used to indivisibly add
;a specified value to a location in memory and return the updated value.
;
        XLEF 2,COUNT ;Load AC2 with address of double word to
                     ;indivisibly update.
        WCOM 3,3     ;Set AC3 (mask) to all ones, testing all
                     ;bits in result.
RETRY:  XWLDA 1,COUNT;Fetch current value to be updated.
        WMOV 1,0     ;Put current value in AC0 for comparison.
        WADI 4,1     ;Update the value by 4. Compare current value
        WMESS        ;in memory with value in AC0. If equal, put
                     ;the new value in AC1 in memory and skip the
                     ;next instruction.
        WBR  RETRY   ;If comparison fails, value in memory has
                     ;been modified. Try again.
        .    .       ;Update succeeded.
   ...
COUNT:  .DWORD 0
```

Wide Move WMOV

WMOV *acs,acd*



Function: *acs -> acd*

Parameters: None

WMOV moves a copy of the 32-bit contents of *acs* into *acd*.

Arguments

acs	Before execution, contains 32-bit value.
	After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd	Before execution, contains 32-bit value.
	After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

MOV	Move

Exceptions

None

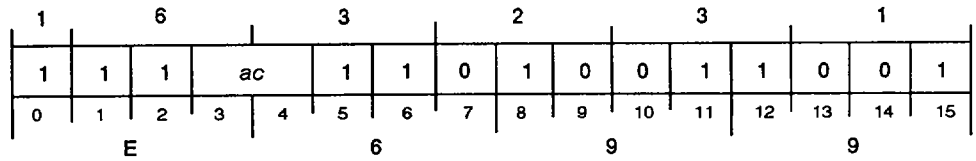
Example

```
;This subroutine deques 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
;                    <return>
;      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 dequeueing element 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!
```

Wide Move Right WMOVR

WMOVR *ac*



Function: shift *ac* right 1 --> *ac*
byte pointer --> word pointer

Parameters: None

WMOVR shifts the contents of *ac* right one bit and shifts a zero into bit 0. This instruction can be used to convert a byte pointer into a word pointer.

Arguments

ac	Before execution, contains 32-bit value.
	After execution, contains 32-bit value.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

WLSI 1,ac	Wide Logical Shift Immediate (convert word pointer in ac to byte pointer).
XLEFB	Load Effective Byte Address (Extended Displacement)
VBP	Skip on Valid Byte Pointer
VWP	Skip on Valid Word Pointer

Exceptions

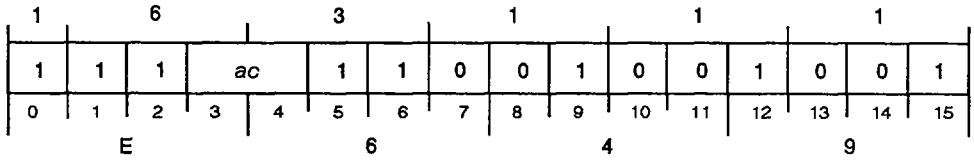
None

Example

```
XWLDA 1,BYTE_ADDRESS        ;Get the byte address.
WMOVR 1                     ;Convert it to a word address.
```

Wide Modify Stack Pointer WMSP

WMSP *ac*



Function: wsp + 2 * *ac* -> wsp

Parameters: None

WMSP adds twice the value of the specified accumulator to the wide stack pointer and tests for potential stack overflow or underflow. It does this by shifting the number in the specified accumulator left one bit and then adding it to the current value of the stack pointer. The result is placed in temporary storage. The instruction then checks for a fixed-point overflow resulting from the shift. If the temporary value is within the limits, it is stored as the new value of the stack pointer.

Arguments

ac	Before execution, contains signed 32-bit integer specifying number of double words to adjust wide-stack pointer. After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Wide stack pointer updated to new value

Related Instructions

MSP	Modify Stack Pointer

Exceptions

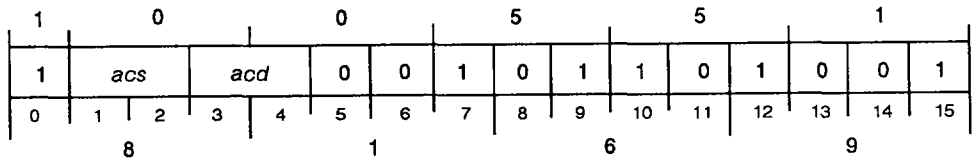
If the shifted value would produce a fixed-point overflow, the stack is not modified. Instead, a return block is pushed, using the original stack pointer as the reference, and the processor jumps to the fault handler routine. Upon the return, processing resumes with **WMSP**.

Example

```
NLDAI  4,0          ;Get a constant 4.
WMSP   0            ;Increment the stack pointer by 4 double words.
```

Wide Multiply WMUL

WMUL *acs,acd*



Function: *acs * acd -> acd*

Parameters: None

WMUL performs a signed multiply of the 32-bit integer contained in *acd* and the 32-bit integer contained in *acs*. *Acd* will contain the least significant 32 bits of the result.

Arguments

acs	Before execution, contains signed 32-bit integer.
	After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd	Before execution, contains signed 32-bit integer.
	After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	1 if result outside specified range; otherwise 0.
PC	PC + 1
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

MUL	Unsigned Multiply
MULS	Singed Multiply
WMULS	Wide Signed Multiply
NMUL	Narrow Multiply

Exceptions

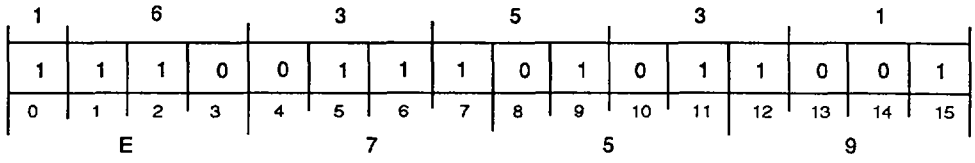
If result is outside the range, -2,147,483,648 to +2,147,483,647 inclusive, PSR(OVR) is set to 1, and *acd* contains least significant 32 bits of result.

Example

```
XWLDA  2,MULTIPLICAND          ;Get the multiplicand.
XWLDA  3,MULTIPLIER            ;Get the multiplier.
WMUL   3,2                     ;Multiply.
XWSTA  2,RESULT                ;Store the result.
```

Wide Signed Multiply WMULS

WMULS



Function: AC1 * AC2 + AC0 -> AC0[2# high]&AC1[2# low]

Parameters: None

WMULS multiplies the signed 32-bit integer contained in AC1 by the signed 32-bit integer contained in AC2. Then it adds the signed 32-bit integer contained in AC0 to the 64-bit result and loads the result into AC0 and AC1.

Arguments

None

Registers, Flags, and Stacks

AC0	Before execution, contains signed 32-bit integer to be added to result.
	After execution, contains 32 high-order bits of result.
AC1	Before execution, contains signed 32-bit integer.
	After execution, contains 32 low-order bits of result.
AC2	Before execution, contains signed 32-bit integer.
	After execution, contents unchanged.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

MUL	Unsigned Multiply
MULS	Signed Multiply
NMUL	Narrow Multiply
WMUL	Wide Multiply

Exceptions

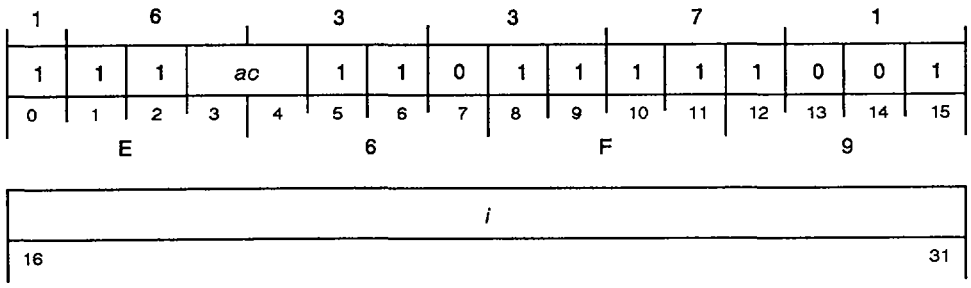
None

Example

```
XWLDA 1,MLTPCAND   ;Get one number to multiply.
XWLDA 2,MLTPER     ;Get the other number to multiply.
XWLDA 0,ADDEND     ;Get the number to add to the product.
WMULS              ;Multiply and add (signed).
XWSTA 0,HIGH_RESULT;Store the high order result.
XWSTA 1,LOW_RESULT ;Store the low order result.
```

Wide Add with Narrow Immediate WNADI

WNADI *i,ac*



Function: *i*[16-bit 2#] + *ac* -> *ac*
ALU CRY -> CRY

Parameters: None

WNADI sign-extends the signed 16-bit immediate value to 32 bits. Then it adds this value to the signed 32-bit integer contained in the specified accumulator.

Arguments

i	Signed 16-bit integer (processor sign-extends to 32 bits).
ac	Before execution, contains signed 32-bit integer.
	After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Set with value of ALU CARRY.
Overflow	1 if ALU overfow.
PC	PC + 2
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

ADDI, NADDI, WADDI
Add a signed 16- or 32-bit immediate value to an accumulator.

ADI, NADI, WADI
Add a 2-bit immediate value to an accumulator.

Exceptions

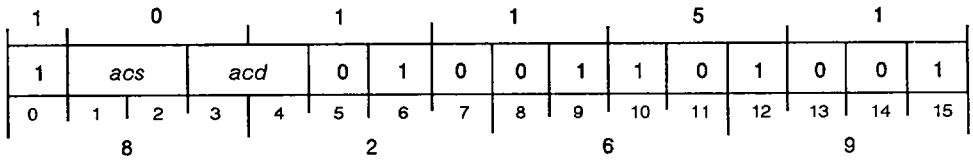
None

Example

```
    ;Convert lower case input to upper case.
    ;
    ;       AC0 = Byte pointer to string
CNVUC:  WLDB        0,2         ;Put a byte of source string into AC2
        WANDI       177,2       ;Mask to seven bits
        WCLM        2,2         ;See if lower case
        .DWORD "A+40            ;Lower limit for compare
        .DWORD "Z+40            ;Upper limit for compare
        WBR NOTLOW              ;Not lower case
        WNADI       -40,2       ;Yes, lower case, convert to upper
```

Wide Negate WNEG

WNEG *acs,acd*



Function: *-acs -> acd*

Parameters: None

NOTE: ALU carry -> CRY
If *acs* = 10000000000_8, overflow = 1

WNEG forms the two's complement of the 32-bit contents of *acs*, placing the result into *acd*.

Arguments

acs	Before execution, contains 32-bit integer.
	After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd	After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Set according to value of ALU CARRY
Overflow	1 if the largest negative 32-bit integer (10000000000_8) is negated; otherwise 0.
PC	PC + 1
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

NEG	Negate
NNEG	Narrow Negate

Exceptions

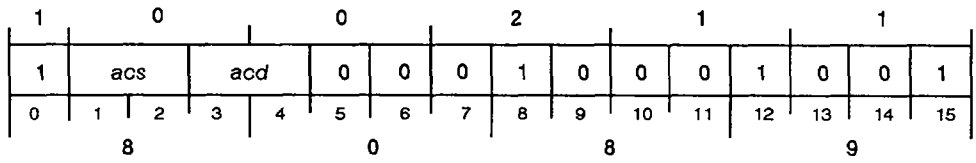
If the largest negative 32-bit integer (10000000000_8) is negated, PSR(OVR) is set to 1.

Example

```
WNEG  0,0          ;Negate the value in AC0.
```

Wide Pop Accumulators WPOP

WPOP *acs,acd*



Function: wide stack -> *acs* to *acd*
-n(1-4) double words --> wide stack
1st stack double word --> *acs*
nth stack double word --> *acd*
wsp-2*n --> wsp

Parameters: None

NOTE: If *acs* is *acd*, 1 double word is popped

WPOP pops double words off the wide stack and loads them into the specified accumulators. The number of double words popped is equal to the number of accumulators specified by *acs* through *acd* inclusive. The accumulators are filled in descending order, starting with the *acs* and continuing downward numerically through the accumulators to and including *acd*, wrapping around, if necessary, with AC3 following AC0. If the same accumulator is specified for *acs* and *acd*, only one double word is popped and it is placed in the specified accumulator.

WPOP decrements the contents of the wide stack pointer by twice the number of double words popped and then checks for stack underflow.

Arguments

acs	Starting accumulator of set; receives first double word popped from stack.
acd	Ending accumulator of set; receives last double word popped from stack.

Registers, Flags, and Stacks

AC0–AC3	Individually included or excluded by beginning and ending boundaries specified by *acs* and *acd*. If excluded from set, contents unchanged.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

WPSH	Wide Push Accumulators

Exceptions

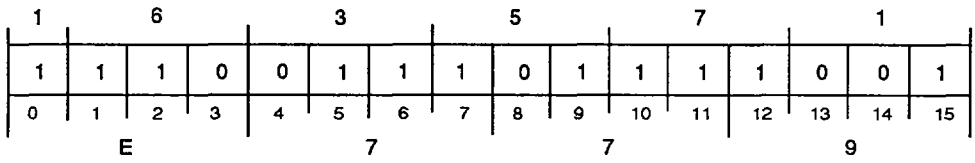
None

Example

```
WPSH   2,0          ;Push AC2, AC3, and AC0 onto the wide stack.
...
WPOP   0,2          ;Pop words off the stack and restore bits
                    ;0-31 of AC0, AC2, and AC3 to their values
                    ;at the time of the WPSH.
```

Wide Pop Block WPOPB

WPOPB



Function: stack -> registers
stack -> -6 double words (wide return block)
wsp -> wsp-(6th double word (bits 17-31)*2+12)

Parameters: None

WPOPB returns control from an intermediate-level interrupt, from an extended operation (**WXOP**), or from a breakpoint (**BKPT**) handler routine (after removing the **BKPT** instruction). It pops six double words off the wide stack and places them in the following locations:

Double Word Popped	Destination
1	Bit 0 into carry; bits 1-31 into PC
2	AC3
3	AC2
4	AC1
5	AC0
6	Bits 0-15 into PSR; bits 17-31 specify wide stack frame size (bit 16 is 0)

If the return is within the current ring, execution continues with the location addressed by the program counter.

If the return is to an outer ring, the instruction stores the WSP and WFP in the appropriate page-zero locations of the current segment; then performs the outer ring crossing and loads the wide stack registers with the contents of the appropriate page-zero locations of the new ring. The value loaded into the WSP is derived as follows:

(current contents of WSP) - (2 x frame size)

Execution continues with the location addressed by the program counter.

Arguments

None

Registers, Flags, and Stacks

AC0-AC3	After execution, contains words popped from stack as specified above.
CARRY	After execution, contains bit 0 of first word popped from stack
Overflow	Unaffected
PC	After execution, contains bits 1-31 of first word popped from stack
Stack	Wide-stack pointer decremented by six double words.

Related Instructions

POPB Pop Block

Exceptions

Two exceptions may be encountered: a protection fault, if a return involves an inward ring crossing; or a stack underflow fault, if the number of words popped exceeds the lower stack limit. When a fault occurs, the appropriate fault handler is entered.

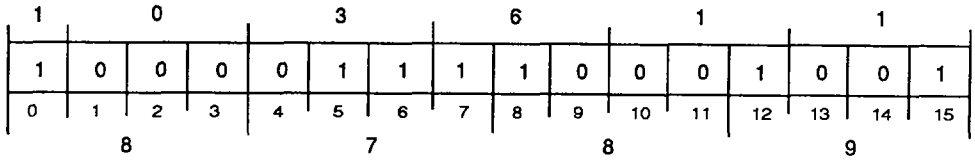
Note that the return block pushed as a result of a fault may contain undefined information; however, AC0 contains the PC value of the instruction wherein the fault occurred and AC1 contains the fault code (8 = ring protection fault; 3 = stack underflow). In case of a ring protection fault, the stack does not get popped.

Example

```
WPOPB           ;Pop 6 double words off the stack, setting
                ;AC0-3, the PSR, and the PC.
```

Wide Pop PC and Jump WPOPJ

WPOPJ



Function: top stack double word -> PC
wsp -> wsp-2

Parameters: None

WPOPJ pops a double word off of the wide stack and places the least significant 28 bits into the program counter. Sequential operation continues with the word addressed by the updated value of the program counter. Underflow is checked after the pop is completed.

Arguments

None

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	Least significant 28 bits popped from wide stack; most significant 3 bits set to the current segment.
PSR	Unchanged
Stack	Wide stack pointer decremented by two; frame pointer unchanged.

Related Instructions

POP, POPB, WPOP, WPOPB
Pop information from the stacks.

Exceptions

If a stack underflow occurs, the stack fault handler is executed.

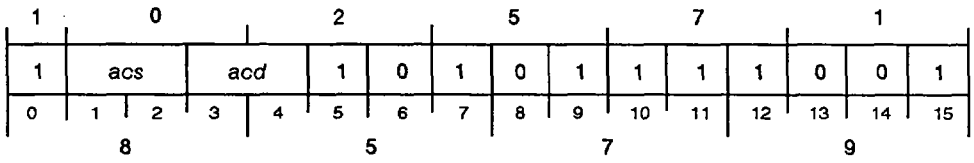
Example

```
;Subroutine to compare two strings
;
;Strings are assumed to be word aligned and to be followed by a
;terminating null (or two, if needed to fill a word).
;
;       AC0 = Byte length of string (without terminator)
;       AC1 = Word pointer to first string
;       AC2 = Word pointer to second string
;
;Returns +1 if they match, 0 if they don't.
```

```
CMPAR:  WPSH        3,3           ;Save return address
        LDAFP       3             ;Get frame pointer
        WINC        0,0           ;Get number of words with terminator
        WINC        0,0           ;Number of characters plus 1
        WHLV        0             ;Number of characters plus 1 / 2
        XNSTA       0,WCNT,3      ;Save count
        XWSTA       1,WPTR.W,3    ;Save one of the pointers
CMPLP:  XNLDA       0,0,2         ;Pick up a word
        XNLDA       1,@WPTR.W,3   ;and its friend
        WSEQ        0,1           ;See if equal
        WPOPJ                     ;No, return false (0)
        XWISZ       WPTR.W,3      ;Move to next word
        WINC        2,2           ;(S)
        XNDSZ       WCNT,3        ;See if done
        WBR CMPLP                 ;
        ISZTS                     ;Bump return (they match)
        WPOPJ                     ;Return
```

Wide Push Accumulators WPSH

WPSH *acs,acd*



Function: *acs* to *acd* -> stack +(1-4) double words -> wide stack
wsp -> wsp +2*(1-4)

Parameters: None

NOTE: If *acs* is *acd*, 1 ac is pushed

WPSH pushes double words from the specified accumulators onto the wide stack. The number of double words pushed is equal to the number of accumulators specified by *acs* through *acd* inclusive. The accumulators are pushed in ascending order, starting with the *acs* and continuing upward numerically through the accumulators to and including the *acd*, wrapping around, if necessary, with AC0 following AC3. If the same accumulator is specified for *acs* and *acd*, only one accumulator, the one specified, is pushed.

WPSH increments the contents of the wide stack pointer by two times the number of accumulators pushed and then checks for stack overflow.

Arguments

acs	Starting accumulator of set; contains first double word pushed onto stack.
acd	Ending accumulator of set; contains last double word pushed onto stack.

Registers, Flags, and Stacks

AC0-AC3	Individually included or excluded by beginning and ending boundaries specified by *acs* and *acd*. After execution, contents unchanged.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

WPOP	Wide Pop Accumulators

Exceptions

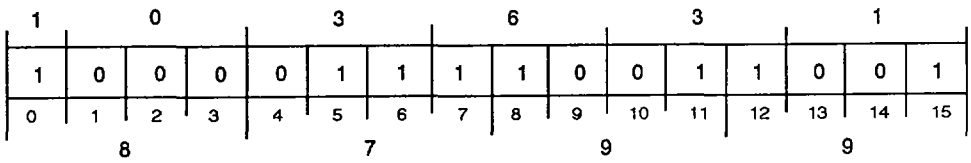
None

Example

```
WPSH  2,0          ;Push AC2, AC3, and AC0 onto the wide stack.
...
WPOP  0,2          ;Pop words off the stack and restore bits
                   ;0-31 of AC0, AC2, and AC3 to their values
                   ;at the time of the WPSH.
```

Wide Restore WRSTR

WRSTR



Function: stack -> locations
stack -> -11 double words (wide return block)
6th double word -> PSR,0
7th double word (1-15) -> stack fault address
8th double word -> wsb
9th double word -> wsl
10th double word -> wsp
11th double word -> wfp

Parameters: None

WRSTR returns control from a base-level interrupt by popping eleven double words off the wide stack and placing them into the following locations:

Double Word Popped	Destination	Notes
1	Carry,PC	Top of wide stack
2	AC3	
3	AC2	
4	AC1	
5	AC0	
6	PSR,0	
7	0,SFA	Stack fault address (bits 1-15)
8	WSB	
9	WSL	
10	WSP	
11	WFP	

If the return is within the current ring, the instruction places the popped stack management information into the four stack registers, stores the stack fault address in the stack fault pointer of the current segment, and continues execution with the location addressed by the program counter.

If the return is an outward crossing to another ring, the instruction stores the popped stack management information into the appropriate page zero locations of the current segment; then performs the outward ring crossing and loads the wide stack registers with the contents of the appropriate page zero locations of the new segment.

Arguments

None

Registers, Flags, and Stacks

AC0-AC3 After execution, contains double words popped from stack as specified above.

CARRY After execution, contains bit 0 of first double word popped

Overflow	Unaffected
PC	After execution, contains bits 1–31 of first double word popped
PSR	After execution, contains bits 0–15 of sixth double word popped.
Stack	After execution, wide stack environment set as described above.

Related Instructions

POP, POPB, WPOP, WPOPB
Pop information from the stacks.

Exceptions

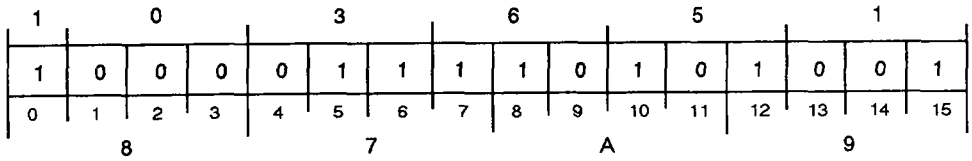
Two exceptions may be encountered: a protection fault, if a return involves an inward ring crossing; or a stack underflow fault, if the number of words popped exceeds the lower stack limit. When a fault occurs, the appropriate fault handler is entered. Note that the return block pushed as a result of the fault may contain undefined information; however, AC0 contains the PC value of the instruction wherein the fault occurred, and AC1 contains the fault code (8 = ring protection fault; 3 = stack underflow). In the case of an underflow fault, the fault handler routine uses the original stack parameters, not the new ones.

Example

```
WRSTR   ;Pop 11 double words off the stack and
        ;restore procesor state in returning from an
        ;interrupt.
```

Wide Return WRTN

WRTN



Function: stack -> registers

Parameters: stack -> -6 double words (wide return block)
wsp-(6th double word (bits 17-31)*2 + 12) -> wsp
AC3(popped) -> wfp

WRTN returns control from a subroutine (that at entry point executed an instruction such as **WSAVS, WSAVR, WSSVS,** or **WSSVR**) by setting the WSP to equal the WFP and then popping six double words off the wide stack. The popped value of AC3 is placed into the WFP as the updated value.

Words popped off the stack are placed into the following locations:

Double Word Popped	Destination
1	Bit 0 into carry; bits 1-31 into PC
2	AC3
3	AC2
4	AC1
5	AC0
6	Bits 0-15 into PSR; bits 17-31 specify wide stack frame size (bit 16 is 0)

If the return is within the current ring, execution continues with the location addressed by the program counter.

If the return is to an outer ring, the instruction stores the WSP and WFP in the appropriate page-zero locations of the current segment; then it performs the outer ring crossing and loads the wide stack registers with the contents of the appropriate page-zero locations of the new ring. The value loaded into the WSP is derived as follows:

(current contents of WSP) - (2 x frame size)

Execution continues with the location addressed by the program counter.

Arguments

None

Registers, Flags, and Stacks

AC0-AC3 After execution, contains double words popped from stack as specified above.

CARRY After execution, contains bit 0 of first double word popped from stack

Overflow Unaffected

PC	After execution, contains bits 1–31 of first double word popped from stack
PSR	After execution, contains bits 0–15 of sixth double word popped.
Stack	After execution, wide stack pointer contains value described above; wide frame pointer contains popped value of AC3.

Related Instructions

POPB, WPOPB Pop a return block from the narrow or wide stack.

Exceptions

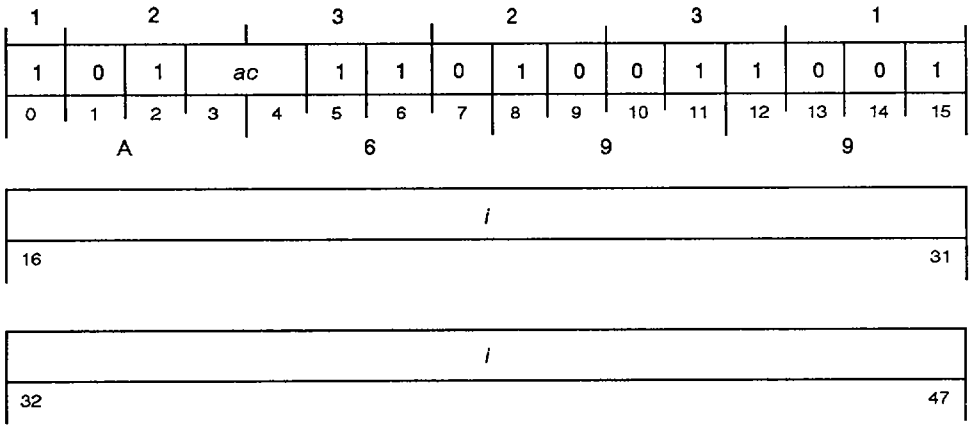
Two exceptions may be encountered: a protection fault, if a return involves an inward ring crossing; or a stack underflow fault, if the number of words popped exceeds the lower stack limit. When a fault occurs, the appropriate fault handler is entered. Note that the return block pushed as a result of the fault may contain undefined information; however, AC0 contains the PC value of the instruction wherein the fault occurred and AC1 contains the fault code (8 = ring protection fault; 3 = stack underflow). In the case of a ring protection fault, the stack does not get popped.

Example

```
;This subroutine deques 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
;                    <return>
;      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 dequeueing element 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!
```

Wide Skip on All Bits Set in Accumulator WSALA

WSALA *i,ac*
(AND ≠ 0 return)
(AND = 0 return)



Function: If *i* AND $\overline{ac}$ = 0 then skip

Parameters: None

WSALA logically ANDs the contents of the immediate field with the complement of the contents of the specified accumulator and skips if the result is zero.

Arguments

i	32–bit value
ac	Before execution, contains 32–bit value. After execution, contents unchanged.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 3 (AND ≠0) PC + 4 (AND = 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WSALM, NSALA, NSALM
Skip on all bits set in accumulator or memory.

Exceptions

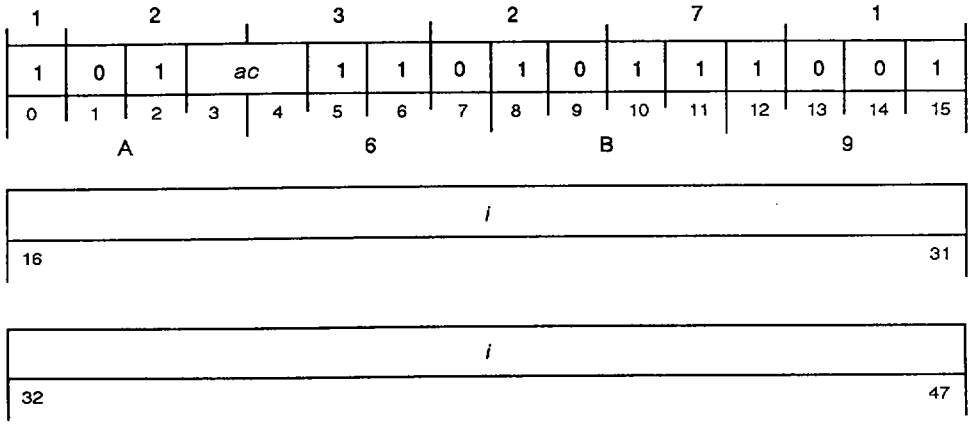
None

Example

```
    XWLDA 2,FLAGS       ;Get the flags double word.
    WSALA 140002,2      ;Are bits 16, 17, and 30 all set?
    WBR   FAIL          ;No. One or more are zero.
    . . .               ;Yes. All bits are set.
    ...
FLAGS:    .DWORD 0      ;Flags double word.
```

Wide Skip on All Bits Set in Double-Word Memory Location — WSALM

WSALM *i,ac*
(AND ≠0 return)
(AND = 0 return)



Function: If *i* AND ($\overline{ac}$) = 0 then skip

Parameters: None

WSALM logically ANDs the contents of the immediate field with the complement of the double word addressed by the specified accumulator and skips if the result is zero.

Arguments

i	32-bit value
ac	Before execution, contains word address of 32-bit value. After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 3 (AND ≠0) PC + 4 (AND = 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WSALA, NSALA, NSALM
Skip on all bits set in accumulator or memory.

Exceptions

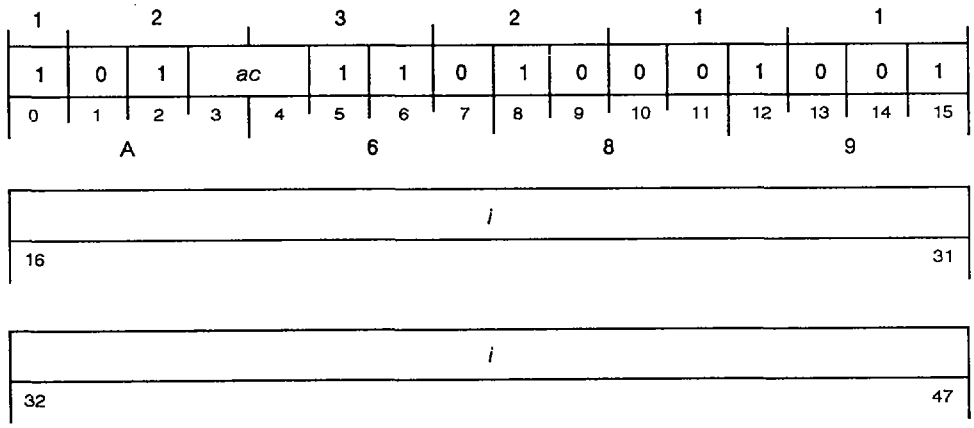
None

Example

```
    XLEF   2,FLAGS      ;Get address of the flags double word.
    WSALM  140002,2     ;Are bits 16, 17, and 30 all set?
    WBR    FAIL         ;No. One or more are zero.
    . . .               ;Yes. All bits are set.
    ...
FLAGS:     .DWORD 0     ;Flags double word.
```

Wide Skip on Any Bit Set in Accumulator WSANA

WSANA *i,ac*
(AND = 0 return)
(AND = non 0 return)



Function: If *i* AND *ac* ≠ 0 then skip
Parameters: None

WSANA logically ANDs the contents of the immediate field with the contents of the specified accumulator and skips on a nonzero result.

Arguments

i	32-bit value
ac	Before execution, contains 32-bit value.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 3 (AND = 0) PC + 4 (AND = non 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WSANM, NSANA, NSANM
Skip on any bit set in accumulator or memory.

Exceptions

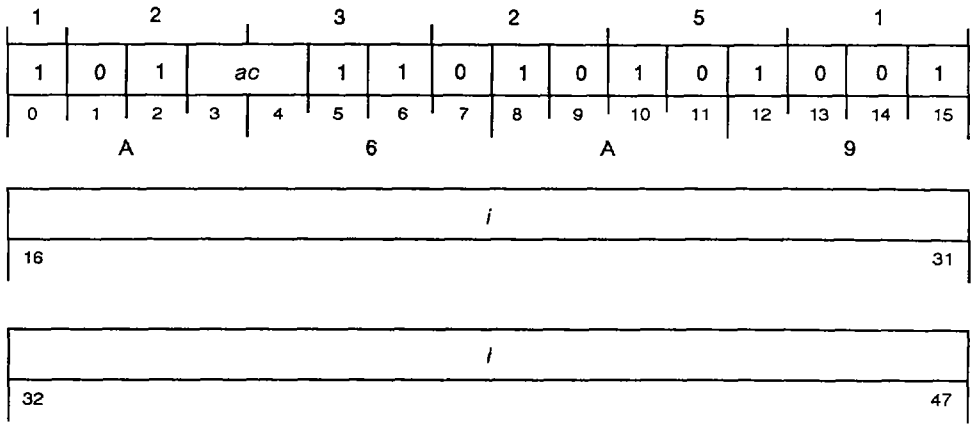
None

Example

```
        XWLDA  2,FLAGS      ;Get the flags double word.
        WSALA  140002,2     ;Are any of bits 16, 17, and 30 set?
        WBR    FAIL         ;No. All three bits are zero.
        . . .               ;Yes. One or more of the three are set.
        . . .
FLAGS:  .DWORD 0            ;Flags double word.
```

Wide Skip on Any Bit Set in Double-Word Memory Location — WSANM

WSANM *i,ac*
(AND = 0 return)
(AND = non 0 return)



Function: If *i* AND (*ac*) ≠ 0 then skip

Parameters: None

WSANM logically ANDs the contents of the immediate field with the contents of the double word addressed by the specified accumulator and skips if the result is not zero.

Arguments

i	32-bit value
ac	Before execution, contains word address of 32-bit value. After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 3 (AND = 0) PC + 4 (AND = non 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WSANA, NSANA, NSANM
Skip on any bit set in accumulator or memory.

Exceptions

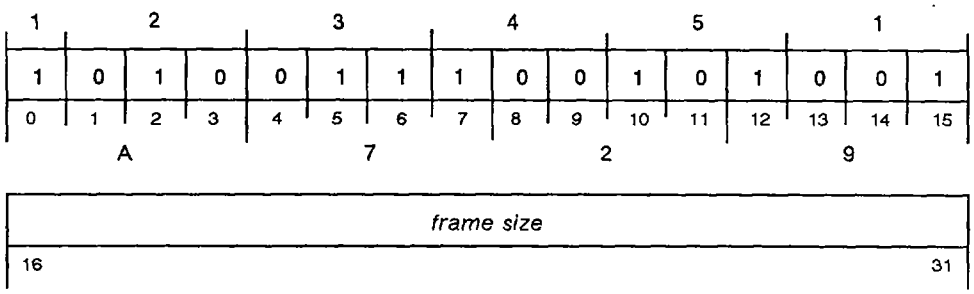
None

Example

```
    XLEF   2,FLAGS      ;Get the flags double word.
    WSALA  140002,2     ;Are any of bits 16, 17, and 30 set?
    WBR    FAIL         ;No. All three bits are zero.
    . . .               ;Yes. One or more of the three are set.
    ...
FLAGS:    .DWORD 0      ;Flags double word.
```

Wide Save/Reset Overflow Mask WSAVR

WSAVR *frame size*



Function: wfp -> AC3
5 double words -> wide stack (partial wide return block)
wsp(after push) -> AC3
wsp(after push) -> wfp
wsp + (*frame size**2) -> wsp
0 -> OVK

Parameters: *frame size* = #(16-bit) -> unchanged

NOTE: 1st double word should be pushed by the {L/X}CALL instruction

WSAVR pushes a return block of 5 double words onto the wide stack, resets the OVK flag bit in the Processor Status Register, and increments the wide stack pointer by the *frame size*.

The return block pushed has the following contents:

Double Word Pushed	Contents
1	AC0
2	AC1
3	AC2
4	Previous WFP
5	Bit 0 = CARRY; bits 1-31 = AC3 bits 1-31 (or return PC value for XCALL and LCALL)

Note that the five words pushed may not make up all of the return block.

An **LCALL** or an **XCALL** instruction pushes the first double word of the return block formatted as follows:

Bits 0-15 contain current PSR.
Bits 16-31 contain **L/XCALL** *argument count*.

After pushing the return block, the instruction places the value of the WSP into both the WFP and AC3, increments the WSP by twice the specified *frame size* (reserving space on the stack for local variables), and sets the PSR(OVK) flag bit to 0, disabling integer overflow.

Arguments

frame size Unsigned 16-bit integer specifying size of frame area (in double words) for storing data on stack after storage of return block.

Registers, Flags, and Stacks

AC0	First double word pushed
AC1	Second double word pushed
AC2	Third double word pushed
AC3	Before execution, bits 1–31 placed in fifth double word pushed.
	After execution, contains WFP.
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 2
PSR	OVK bit set to 0
Stack	Wide stack pointer incremented by five double words.

Related Instructions

SAVZ, WSAVS, WSSAVR, WSSAVS
Push a return block onto a stack.

WRTN Wide Return

Exceptions

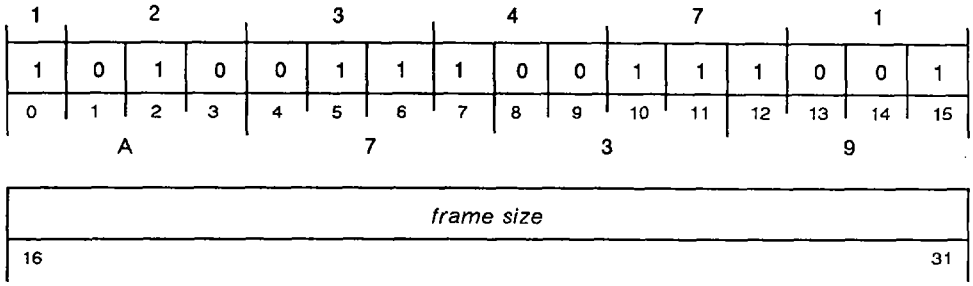
A check for stack overflow is made before the return block is pushed.

Example

```
    LCALL  SUBROUT,0,0              ;Subroutine call
    ...
SUBROUT:
           WSAVR        2           ;Save ACs, update FP, and allocate 2
           . . .                    ;double words for local storage.
           WRTN                     ;Return from subroutine, restoring
                                    ;ACs.
```

Wide Save/Set Overflow Mask WSAVS

WSAVS *frame size*



Function: wfp -> AC3
5 double words -> wide stack (partial wide return block)
wsp(after push) -> AC3
wsp(after push) -> wfp
wsp + (*frame size**2) -> wsp
1 -> OVK

Parameters: *frame size* = #(16-bit) -> unchanged

NOTE: 1st double word should be pushed by the {L/X}CALL instruction

WSAVS pushes a return block of 5 double words onto the wide stack, resets the WSP and WFP, sets the Processor Status Register OVK flag bit, and increments the wide stack pointer by the *frame size*.

The return block pushed has the following contents:

Double Word Pushed	Content
1	AC0
2	AC1
3	AC2
4	Previous WFP
5	Bit 0 = CRY, bits 1-31 = AC3, bits 1-31 (or return PC value for XCALL and LCALL)

Note that the five double words pushed may not make up all of the return block. An **LCALL** or an **XCALL** instruction pushes the first double word of the return block formatted as follows:

Bits 0-15 contain current PSR.
Bits 16-31 contain **L/XCALL** *argument count*.

After pushing the return block, the instruction places the value of the WSP into both the WFP and AC3, increments the WSP by twice the specified *frame size* (reserving space on the stack for local variables), and sets the PSR(OVK) flag bit to 1, enabling integer overflow.

Arguments

frame size Unsigned 16-bit integer specifying size of frame area (in double words) for storing data on stack after storage of return block.

Registers, Flags, and Stacks

AC0	First double word pushed
AC1	Second double word pushed
AC2	Third double word pushed
AC3	Before execution, bits 1–31 placed in fifth double word pushed.
	After execution, contains WFP.
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 2
PSR	OVK bit set to 1
Stack	Wide stack pointer incremented by five double words.

Related Instructions

SAVZ, WSAVS, WSSAVR, WSSAVS
Push a return block onto a stack.

WRTN Wide Return

Exceptions

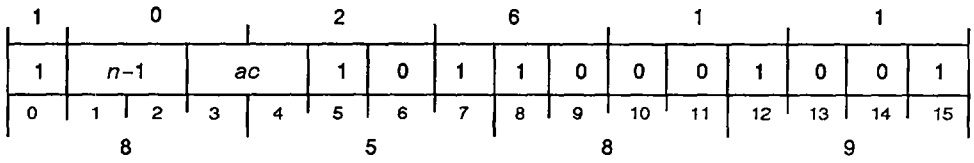
A check for stack overflow is made before the return block is pushed.

Example

```
LCALL SUBROUT,0,0  ;Subroutine call.
. . .
SUBROUT:
      WSAVS 2      ;Save ACs, update FP, and allocate 2
      . . .        ;double words for local storage.
      WRTN         ;Return from the subroutine, restoring ACs.
```

Wide Subtract Immediate WSBI

WSBI *n,ac*



Function: *ac - n -> ac*
ALU CRY -> CRY

Parameters: None

WSBI subtracts an integer in the range of 1 to 4 from a signed 32-bit integer contained in the specified accumulator, storing the result in the specified accumulator.

Arguments

n	Integer in range 1-4. Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field. Thus, programmer should code exact value to be subtracted.
ac	Before execution, contains signed 32-bit integer. After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Set with value of ALU CARRY.
Overflow	1 if an ALU overflow.
PC	PC + 1
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

SBI	Subtract Immediate
NSBI	Narrow Subtract Immediate

Exceptions

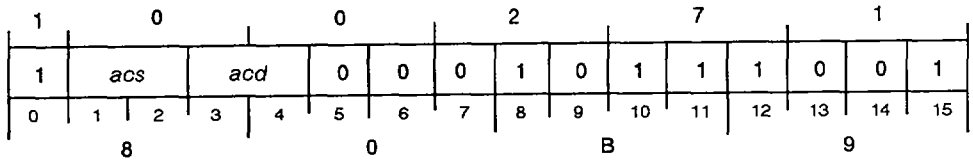
None

Example

```
XWLDA 3,FIRST      ;Get first value.
WSBI  4,3          ;Subtract a constant 4 from AC3.
XWSTA 3,RESULT     ;Store the result.
```

Wide Skip if Equal to WSEQ

WSEQ *acs,acd*
(*acs* ≠ *acd* return)
(*acs* = *acd* return)



Function: If *acs* = *acd* then skip

Parameters: None

NOTE: If *acd* is *acs*, *acs* is compared with 0

WSEQ compares the 32-bit value in *acs* to the 32-bit value in *acd* and skips if the two are equal.

If *acs* and *acd* are the same accumulator, the instruction compares the integer contained in the accumulator to 0.

Arguments

acs	Before execution, contains 32-bit value.
	After execution, contents unchanged.
acd	Before execution, contains 32-bit value.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1 (*acs* ≠ *acd*) PC + 2 (*acs* = *acd*)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WSEQI	Wide Skip if AC Equal to Immediate

Exceptions

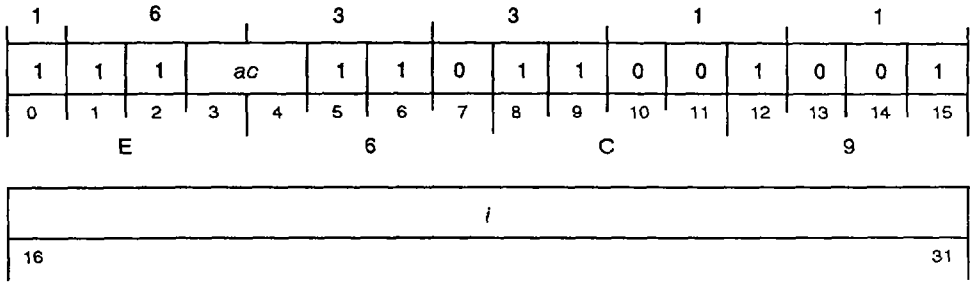
None

Example

```
WSEQ   2,3          ;Are AC2 and AC3 equal?
WBR    NOT_EQUAL    ;No.
. . .               ;Yes.
```

Wide Skip if AC Equal to Immediate WSEQI

WSEQI *i,ac*
(*ac* ≠ *i* return)
(*ac* = *i* return)



Function: If *ac* = *i* then skip

Parameters: None

WSEQI sign-extends the 16-bit immediate field. Then it compares this 32-bit integer to the 32-bit integer in *ac* and skips depending on result.

Arguments

i	Signed 16-bit integer (instruction sign-extends to 32 bits).
ac	Before execution, contains signed 32-bit integer.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.	
CARRY	Unchanged	
Overflow	0	
PC	PC + 2	(if *ac* ≠ *i*)
	PC + 3	(if *ac* = *i*)
PSR	Unchanged	
Stack	Unchanged	

Related Instructions

WSGTI, WSNEI, WSLEI
Compare an immediate value with the contents of an accumulator and skip depending on the result.

Exceptions

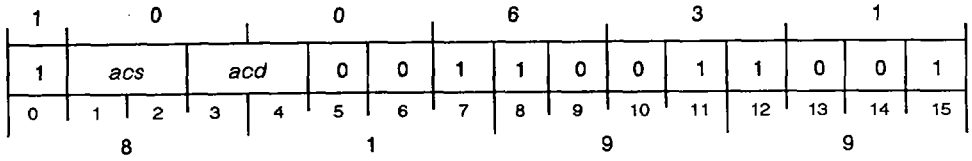
None

Example

```
WSEQI  5,3          ;Does AC3 contain 5?
WBR    NOT_5        ;No.
. . .               ;Yes.
```

Wide Signed Skip if Greater than or Equal to WSGE

WSGE *acs,acd*
(*acs* < *acd* return)
(*acs* >= *acd* return)



Function: If *acs* > = *acd* then skip

Parameters: None

NOTE: If *acd* is *acs*, *acs* is compared with 0

WSGE performs a signed comparison of the 32-bit integer contained in *acs* and the 32-bit integer in *acd* and skips if the first is greater than or equal to the second.

If *acs* and *acd* are the same accumulator, the instruction compares the integer contained in the accumulator to 0.

Arguments

acs	Before execution, contains signed 32-bit integer.
	After execution, contents unchanged.
acd	Before execution, contains signed 32-bit integer.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *acs* and *acd*; otherwise unused.	
CARRY	Unchanged	
Overflow	0	
PC	PC + 1	(*acs* < *acd*)
	PC + 2	(*acs* >= *acd*)
PSR	Unchanged	
Stack	Unchanged	

Related Instructions

WSEQ, WSNE, WSLE, WSLT, WSGT
Wide signed conditional skips.

Exceptions

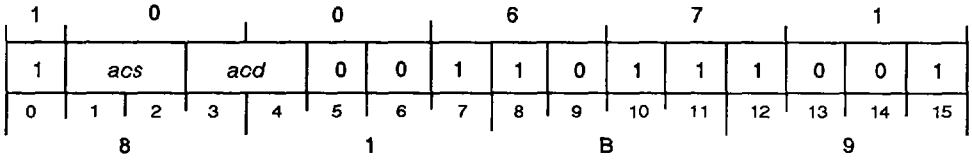
None

Example

```
WSGE   1,2          ;Is contents of AC1 >= contents of AC2?
WBR    AC2_GREATER  ;AC2 is greater.
. . .               ;AC1 is greater than or equal.
```

Wide Signed Skip if Greater than WSGT

WSGT *acs,acd*
(*acs* <= *acd* return)
(*acs* > *acd* return)



Function: If *acs* > *acd* then skip

Parameters: None

NOTE: If *acd* is *acs*, *acs* is compared with 0

WSGT performs a signed comparison of the 32-bit integer in *acs* and the 32-bit integer in *acd* and skips if the first is greater than the second.

If *acs* and *acd* are the same accumulator, the instruction compares the integer contained in the accumulator to 0.

Arguments

acs	Before execution, contains signed 32-bit integer.
	After execution, contents unchanged.
acd	Before execution, contains signed 32-bit integer.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *acs* and *acd*; otherwise unused.	
CARRY	Unchanged	
Overflow	0	
PC	PC + 1	(*acs* <= *acd*)
	PC + 2	(*acs* > *acd*)
PSR	Unchanged	
Stack	Unchanged	

Related Instructions

WSEQ, WSNE, WSLE, WSGE, WSLT
Wide signed conditional skip instructions.

Exceptions

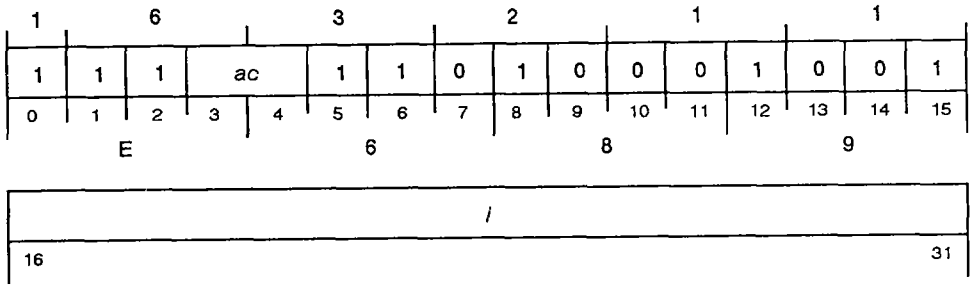
None

Example

```
WSGT   1,2          ;Is contents of AC1 > contents of AC2?
WBR    AC2_GREATER  ;AC2 is greater than or equal.
. . .               ;AC1 is greater.
```

Wide Skip if AC Greater than Immediate WSGTI

WSGTI *i,ac*
(*ac* <= *i* return)
(*ac* > *i* return)



Function: If *ac* > *i* then skip

Parameters: None

WSGTI performs a signed comparison of the signed 32-bit integer in *ac* and the sign-extended immediate value and skips if *ac* is greater.

Arguments

i	Signed 16-bit integer (processor sign-extends to 32 bits).
ac	Before execution, contains signed 32-bit integer.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2 (if *ac* <= *i*) PC + 3 (if *ac* > *i*)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WSEQI, WSNEI, WSLEI
Compare the contents of an accumulator with an immediate value and conditionally skip.

Exceptions

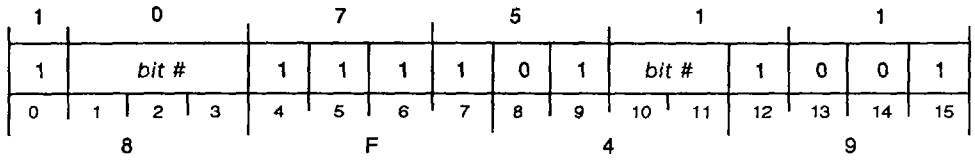
None

Example

```
WSGTI  400,2        ;Is contents of AC2 > constant 400(8)?
WBR    AC2_LE       ;AC2 is less than or equal to 400(8).
. . .               ;AC2 is greater than 400(8).
```

Wide Skip on Bit Set to One — WSKBO

WSKBO *bit number*
(bit = 0 return)
(bit = 1 return)



Function: If *bit #*(AC0) = 1 then skip

Parameters: None

WSKBO tests a specified bit in AC0 and skips if the bit is 1.

Arguments

bit number — Bits 1–3 and 10–11 specify bit position in range 0–31 in AC0. Value 0 specifies highest-order bit and value 31 specifies lowest-order bit.

Registers, Flags, and Stacks

AC0	Before execution, contains 32-bit value. After execution, contents unchanged.
AC1–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1 (if bit value = 0) PC + 2 (if bit value = 1)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WSKBZ — Wide Skip on Bit Set to Zero

Exceptions

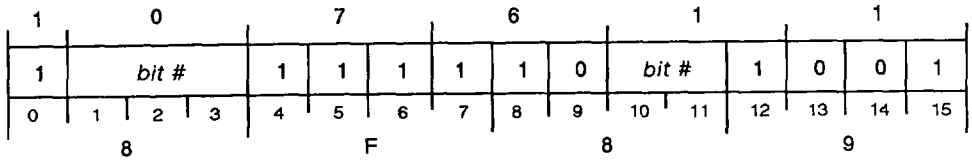
None

Example

```
XWLDA 0,FLAGS      ;Get the flags double word.
WSKBO 18.          ;Skip if bit 18 is one.
WBR   NOT_ONE      ;Bit 18 is zero.
. . .              ;Bit 18 is one.
```

Wide Skip on Bit Set to Zero WSKBZ

WSKBZ *bit number*
(bit = 1 return)
(bit = 0 return)



Function: If *bit* #(AC0) = 0 then skip

Parameters: None

WSKBZ tests a specified bit in AC0 and skips if the bit is 0.

Arguments

bit number Bits 1–3 and 10–11 specify bit position in range 0–31 in AC0. Value 0 specifies highest-order bit and value 31 specifies lowest-order bit.

Registers, Flags, and Stacks

AC0	Before execution, contains 32-bit value. After execution, contents unchanged.
AC1–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1 (if bit value = 1) PC + 2 (if bit value = 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WSKBO Wide Skip on Bit Set to One

Exceptions

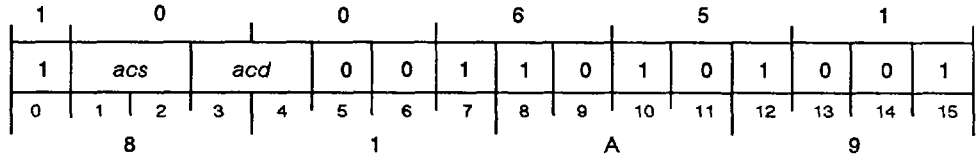
None

Example

```
XWLDA 0,FLAGS      ;Get the flags double word.
WSKBZ 18.          ;Skip if bit 18 is zero.
WBR   NOT_ZERO     ;Bit 18 is one.
. . .              ;Bit 18 is zero.
```

Wide Signed Skip if Less than or Equal to WSLE

WSLE *acs,acd*
(*acs* > *acd* return)
(*acs* <= *acd* return)



Function: If *acs* < = *acd* then skip

Parameters: None

NOTE: If *acd* is *acs*, *acs* is compared with 0

WSLE performs a signed comparison of the 32-bit integer in *acs* and the 32-bit integer in *acd* and skips depending on the result.

If *acs* and *acd* are the same accumulator, the instruction compares the integer contained in the accumulator to 0.

Arguments

acs	Before execution, contains signed 32-bit integer .
	After execution, contents unchanged.
acd	Before execution, contains signed 32-bit integer.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1 (*acs* > *acd*) PC + 2 (*acs* <= *acd*)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WSEQ, WSNE, WSLE, WSGE, WSGT
Wide signed conditional skip instructions.

Exceptions

None

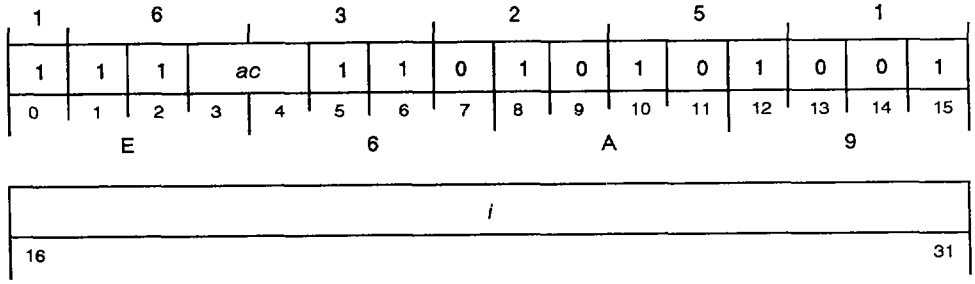
Example

```
WSLE 1,2  ;If the integer contained in AC1 is less than or equal to
          ;the integer contained in AC2, the next word is skipped;
          ;otherwise, the next sequential word is executed.
```

Wide Skip if AC Less than or Equal to Immediate

WSLEI

WSLEI *i,ac*
(*ac* > *i* return)
(*ac* <= *i* return)



Function: If *ac* < = *i* then skip

Parameters: None

WSLEI performs a signed comparison of the signed 32-bit integer in *ac* to the sign-extended immediate value and skips if *ac* is less than or equal.

Arguments

i	Signed 16-bit integer (processor sign-extends to 32 bits).
ac	Before execution, contains signed 32-bit integer. After execution, contents unchanged.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2 (if *ac* > *i*) PC + 3 (if *ac* <= *i*)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WSEQI, WSGTI, WSNEI
Compare the contents of an accumulator with an immediate value and conditionally skip.

Exceptions

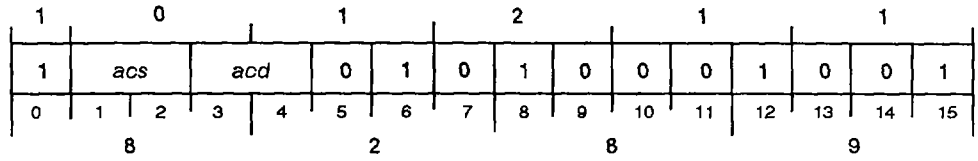
None

Example

```
WSLEI  400,2        ;Is contents of AC2 <= constant 400₈?
WBR    AC2_GT       ;AC2 is greater than 400₈.
. . .               ;AC2 is less than or equal to 400₈.
```

Wide Signed Skip if Less than WSLT

WSLT *acs,acd*
(*acs* >= *acd* return)
(*acs* < *acd* return)



Function: If *acs* < *acd* then skip

Parameters: None

NOTE: If *acd* is *acs*, *acs* is compared with 0

WSLT performs a signed comparison of the 32-bit integer in *acs* and the 32-bit integer in *acd* and skips if the first is less than the second.

If *acs* and *acd* are the same accumulator, the instruction compares the integer contained in the accumulator to 0.

Arguments

acs	Before execution, contains signed 32-bit integer.
	After execution, contents unchanged.
acd	Before execution, contains signed 32-bit integer.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1 (*acs* >= *acd*) PC + 2 (*acs* < *acd*)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WSEQ, WSNE, WSLE, WSGE, WSGT
Wide signed conditional skip instructions.

Exceptions

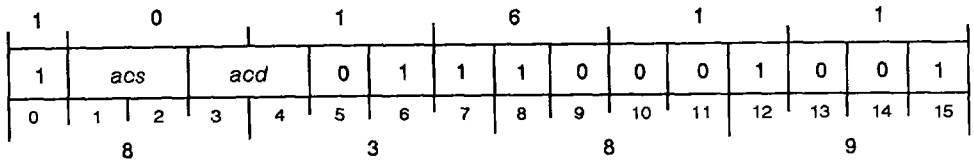
None

Example

```
WSLT   1,2          ;Is contents of AC1 < contents of AC2?
WBR    AC2_LESS     ;AC2 is less than or equal.
. . .               ;AC1 is less than AC2.
```

Wide Skip on Nonzero Bit — WSNB

WSNB *acs,acd*
(bit = 0 return)
(bit = 1 return)



Function: If (E)bit = 1 then skip

Parameters: *acs* = base word pointer -> unchanged
acd = word offset & bit identifier -> unchanged

WSNB forms a bit pointer from the contents of *acs* and *acd* and skips if the bit referenced by the bit pointer is one.

Arguments

acs — Before execution, contains high-order bits of bit pointer.
If *acs* and *acd* are the same accumulator, the high-order bits are treated as if they were zero in the current segment.
After execution, contents unchanged.

acd — Before execution, contains low-order bits of bit pointer.
After execution, contents unchanged.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1 (bit = 0) PC + 2 (bit = 1)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WSZB — Wide Skip on Zero Bit

Exceptions

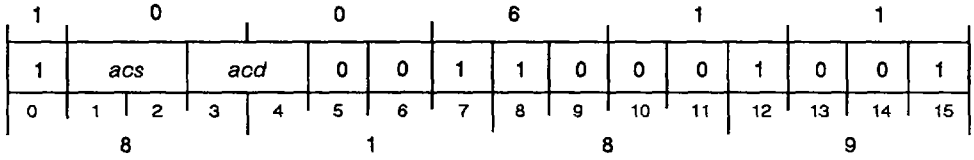
None

Example

```
XLEF   0,FLAGS      ;Get word address of flags word.
NLADI  3,1          ;Get a 3 in AC1.
WSNB   0,1          ;Is bit 3 of the flags word set?
WBR    NOT_SET      ;No.
. . .               ;Yes.
...
FLAGS: .WORD 0      ;Flags word.
```

Wide Skip if Not Equal to WSNE

WSNE *acs,acd*
(*acs* = *acd* return)
(*acs* ≠ *acd* return)



Function: If *acs* ≠ *acd* then skip

Parameters: None

NOTE: If *acd* is *acs*, *acs* is compared with 0

WSNE compares the value in *acs* to the value in *acd* and skips if the two are not equal.

If *acs* and *acd* are the same accumulator, then the instruction compares the integer contained in the accumulator to 0.

Arguments

acs	Before execution, contains 32-bit value.
	After execution, contents unchanged.
acd	Before execution, contains 32-bit value.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1 (*acs* = *acd*) PC + 2 (*acs* ≠ *acd*)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WSEQ, WSLE, WSGE, WSLT, WSGT
Wide signed conditional skip instructions.

Exceptions

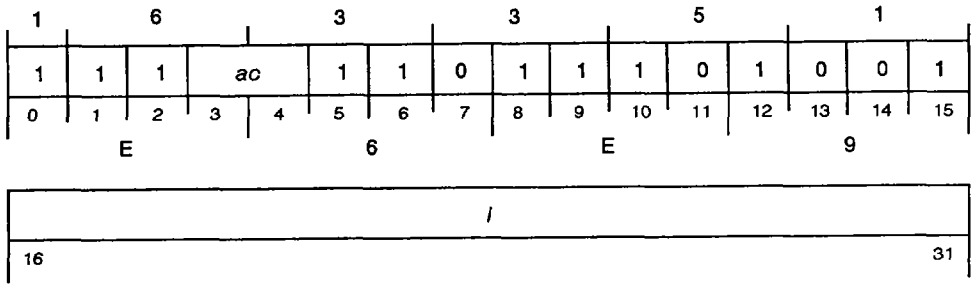
None

Example

```
WSNE   2,3          ;Are AC2 and AC3 equal?
WBR    EQUAL        ;Yes.
. . .               ;No.
```

Wide Skip if AC Not Equal to Immediate WSNEI

WSNEI *i,ac*
(*ac* = *i* return)
(*ac* ≠ *i* return)



Function: If *ac* ≠ *i* then skip

Parameters: None

WSNEI performs a signed comparison of the signed 32-bit integer in *ac* to the sign-extended immediate value and skips if they are not equal.

Arguments

i	Signed 16-bit integer (processor sign-extends to 32 bits).
ac	Before execution, contains signed 32-bit integer.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2 (*ac* = *i*) PC + 3 (*ac* ≠ *i*)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WSEQI, WSGTI, WSLEI
Compare the contents of an accumulator with an immediate value and conditionally skip.

Exceptions

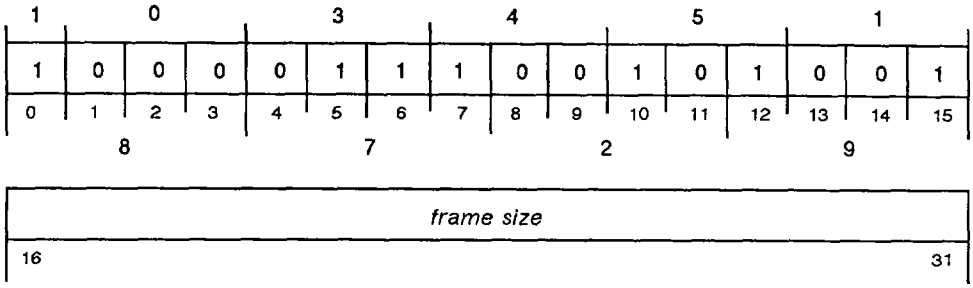
None

Example

```
WSNEI  5,3          ;Does AC3 contain 5?
WBR    EQUAL_5      ;Yes.
. . .               ;No.
```

Wide Special Save/Reset Overflow Mask WSSVR

WSSVR *frame size*



Function:
6 double words -> wide stack
PSR + 0s -> 1st double word
AC0 -> 2nd double word
AC1 -> 3rd double word
AC2 -> 4th double word
wfp(previous) -> 5th double word
CRY -> 6th double word(bit 0)
AC3 -> 6th double word(bits 1-31)
wsp(after push) -> wfp
wsp(after push) -> AC3
wsp + (*frame size**2) -> wsp
0 -> OVR
0 -> OVK

Parameters: *frame size* = #(16-bit) -> unch

WSSVR pushes a return block of 6 double words (representing the current operating environment) onto the wide stack, resets the Processor Status Register OVK and OVR flag bits, and increments the wide stack pointer by the *frame size*.

The return block pushed has the following contents:

Double Word Pushed	Contents
1	PSR (in bits 0-15; bits 16-31 set to 0)
2	AC0
3	AC1
4	AC2
5	Previous WFP
6	Bit 0 = CRY, bits 1-31 = AC3 (or return PC value)

After pushing the return block, the instruction places the value of the WSP into both the WFP and AC3, increments the WSP by twice the specified *frame size* (reserving space on the stack for local variables), and sets the Processor Status Register OVK and OVR flag bits to 0, disabling integer overflow.

WSSVR is typically used after an **XJSR** or **LJSR** instruction, which performs an intra-ring transfer to a subroutine that requires no parameters and that uses a **WRTN** to return control back to the calling sequence.

Arguments

frame size Unsigned 16-bit integer specifying size of frame area (in double words) for storing data on stack after storage of return block.

Registers, Flags, and Stacks

AC0	Second double word pushed
AC1	Third double word pushed
AC2	Fourth double word pushed
AC3	Before execution, bits 1–31 pushed in fifth double word.
	After execution, contains new WFP.
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 2
PSR	OVK and OVR bits set to 0.
Stack	Wide stack pointer incremented by six double words.

Related Instructions

SAVZ, WSAVR, WSAVS,WSSAVS
Push a return block on a stack.

WRTN Wide Return

Exceptions

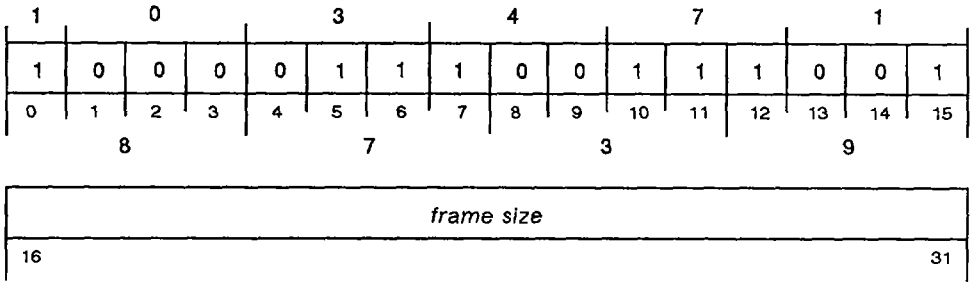
A check for stack overflow is made before the return block is pushed.

Example

```
LJSR   SUBROUT      ;Subroutine call.
       .
       .
       .
SUBROUT:
       WSSVR 2      ;Save ACs, update FP, and allocate 2
       .            ;double words for local storage.
       .
       .
       WRTN         ;Return from the subroutine, restoring ACs.
```

Wide Special Save/Set Overflow Mask WSSVS

WSSVS *frame size*



Function:
6 double words -> wide stack
PSR + 0s -> 1st double word
AC0 -> 2nd double word
AC1 -> 3rd double word
AC2 -> 4th double word
wfp(previous) -> 5th double word
CRY -> 6th double word(bit 0)
AC3 -> 6th double word(bits 1-31)
wsp(after push) -> wfp
wsp(after push) -> AC3
wsp + (*frame size**2) -> wsp
0 -> OVR
1 -> OVK

Parameters: *frame size* = #(16-bit) -> unch

WSSVS pushes a return block of 6 double words (representing the current operating environment) onto the wide stack, resets the WSP and WFP, sets the Processor Status Register OVK flag bit to 1 and OVR flag bit to 0, and increments the wide stack pointer by the *frame size*.

The return block has the following contents:

Double Word Pushed	Contents
1	PSR (in bits 0-15; bits 16-31 set to 0)
2	AC0
3	AC1
4	AC2
5	Previous WFP
6	Bit 0 = CARRY, bits 1-31 = AC3 (or return PC value)

After pushing the return block, the instruction places the value of the WSP into both the WFP and AC3, increments the WSP by twice the specified *frame size* (reserving space on the stack for local variables), and sets the Processor Status Register OVK flag bit to 1 and OVR bit to 0, enabling integer overflow.

WSSVS is typically used after an **XJSR** or **LJSR** instruction, which performs an intra-ring transfer to a subroutine that requires no parameters and that uses a **WRTN** to return control back to the calling sequence.

Arguments

frame size Unsigned 16-bit integer specifying size of frame area (in double words) for storing data on stack after storage of return block.

Registers, Flags, and Stacks

AC0	Second double word pushed
AC1	Third double word pushed
AC2	Fourth double word pushed
AC3	Before execution, bits 1–31 pushed in fifth double word.
	After execution, contains new WFP.
CARRY	Unchanged
Overflow	Unaffected
PC	PC + 2
PSR	OVK bit set to 1; OVR bit set to 0.
Stack	Wide stack pointer incremented by six double words.

Related Instructions

SAVZ, WSAVS,WSSAVS
Push a return block onto a stack.

WRTN Wide Return

Exceptions

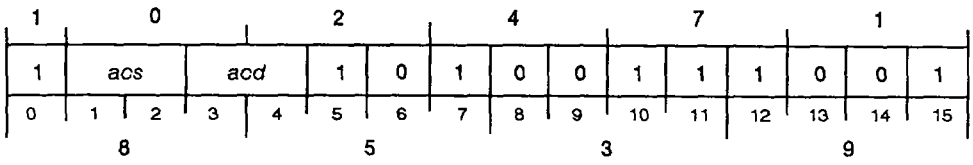
A check for stack overflow is made before the return block is pushed.

Example

```
LJSR   SUBROUT      ;Subroutine call.
       .
       .
       .
SUBROUT:
       WSSVS 2      ;Save ACs, update FP, and allocate 2
       .            ;double words for local storage.
       .
       .
       WRTN         ;Return from the subroutine, restoring ACs.
```

Wide Store Byte WSTB

WSTB *acs,acd*



Function: *acd* [right byte] --> (E)byte

Parameters: *acs* = byte pointer --> unchanged

WSTB stores a copy of the rightmost byte of *acd* into memory at the address specified by *acs*.

Arguments

acs Before execution, contains 32-bit byte address.

After execution, contents unchanged.

acd(24-31) Before execution, contains byte to be stored.

After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

STB	Store Byte

Exceptions

None

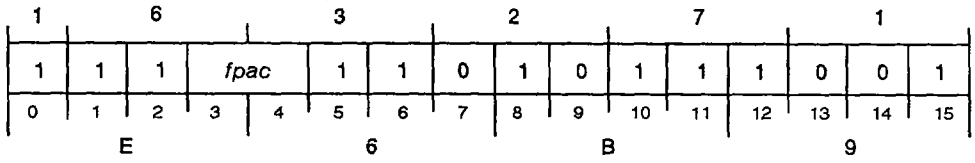
Example

```
;This subroutine appends one or two nulls to the end of a string,
;filling to a word boundary.
;
;Calling conventions:          XJSR NFILL
;                              <return>
;
;       AC1 = Byte pointer to start of string
;       AC2 = Length of string
;       AC3 = Return address
```

```
NFILL:  WPSH    3,3         ;Save return address
        WSUB    3,3         ;Get a zero
        WADD    2,1         ;Get end of string
        WSTB    1,3         ;Append a null
        WINC    1,1         ;Bump pointer
        MOVR#   1,1,SZC     ;Check if odd (middle of word)
        WSTB    1,3         ;Yes, append another null
        LDAFP   3           ;AC3 contains frame pointer
        WPOPJ               ;Return
```

Wide Store Integer WSTI

WSTI *fpac*



Function: *fpac*[fp#] -> @(AC3)[#]
AC3 -> AC2
0 -> CRY

Parameters: AC1 = data-type indicator -> unchanged
AC2 = x -> AC3
AC3 = byte pointer -> last byte pointer + 1
fpac = floating-point # -> unchanged

WSTI converts the contents of a floating-point accumulator to an integer of the specified data type and length, and stores the result as a string in memory beginning at the specified byte location. The digits are stored right-aligned with the high-order byte at the specified byte location; the low-order bytes follow in subsequent locations.

Arguments

fpac	Before execution, contains 64-bit floating-point number to be converted.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0	Unused
AC1	Before execution, contains data type and string size of converted data. **WSTI** does not use the scale factor in the data type indicator.
	After execution, contents unchanged.
AC2	After execution, contains initial value of AC3.
AC3	Before execution, contains starting byte location for high-order byte; contents increment by 1 with each byte stored.
	After execution, contains address of next byte following last byte of string.
CARRY	Set to 1 if number of significant digits to be stored is larger than specified string length; otherwise set to 0.
FPAC0-FPAC3	Can be individually specified for *fpac*; otherwise unused.
FPSR	Unchanged
Overflow	0
PC	PC + 1
Stack	Unchanged

Related Instructions

FINT	Integerize
STI	Store Integer
STIX	Store Integer Extended
WSTIX	Wide Store Integer Extended

Exceptions

If the number in the specified *fpac* has any fractional part, the result is undefined. Use the Integerize (**FINT**) instruction to clear any fractional part.

If the number in *fpac* is too large to convert to the specified data type, a commercial fault is initiated.

If the number to be stored is too large to fit in the destination field, **WSTI** discards high-order digits until the number fits. The instruction stores the remaining low-order digits and sets CARRY to 1.

If the number to be stored does not completely fill the destination field, the data type determines the instruction's actions:

For data types 0 through 5, the high-order digits are set to 0.

For data type 6, the high-order digits are set to the value of the sign.

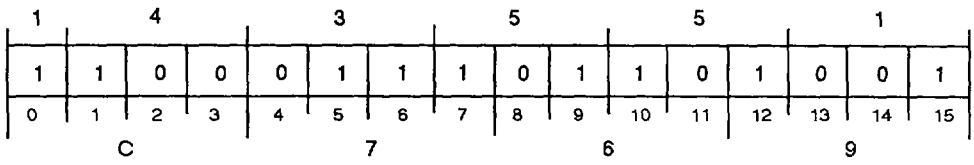
For data type 7, the low-order bytes are set to 0.

Example

```
XNLDA  1,DTYPE      ;AC1 contains the data type indicator.
XLEF   3,FIELD      ;Word pointer to the integer field.
WADD   3,3          ;AC3 is a byte pointer to the integer.
WSTI   2            ;Convert the contents of FPAC2 into a
                    ;commercial of the type specified by
                    ;AC1, at the location specified by AC3.
```

Wide Store Integer Extended WSTIX

WSTIX



Function: fpac(1-4)[fp#] -> (E)[#]
AC3 -> AC2
0 -> CRY

Parameters: AC1 = data-type indicator -> unchanged
AC2 = x -> AC3
AC3 = byte pointer -> last byte pointer + 1

WSTIX converts the contents of the four floating-point accumulators to an integer of the specified data-type format (0 through 5), and stores the result as a string in memory beginning at the specified byte location.

The string is derived from four 8-digit frames, each frame comprising the low-order 8 digits from an fpac conversion. The digits are stored right-aligned and in sequence, with the least significant 8 digits (derived from FPAC3) stored at the higher address locations of the string. The digits derived from FPAC2, FPAC1, and FPAC0 (most-significant digits) are stored sequentially downward in the string. The sign of the stored integer is the logical OR of the signs of all four fpacs.

Arguments

None

Registers, Flags, and Stacks

AC0	Unused
AC1	Before execution, contains data type and number of digits for converted data. **WSTIX** does not use the scale factor in the data type indicator.
	After execution, contents unchanged.
AC2	After execution, contains initial value of AC3.
AC3	Before execution, contains starting byte location for high-order byte.
	After execution, contains address of next byte following last byte of string.
CARRY	0
FPAC0-FPAC3	Before execution, each fpac holds 64-bit floating-point value to be converted. FPAC0 contains high-order digit; FPAC3 contains low-order digit.
	After execution, contents unchanged.
FPSR	Undefined
Overflow	0
PC	PC + 1
Stack	Unchanged

Related Instructions

STI	Store Integer
STIX	Store Integer Extended
WSTI	Wide Store Integer

Exceptions

If data types 6 or 7 are specified, a commercial fault is initiated.

If any of the fpacs contains a value greater than $10^{16}-1$, a commercial fault is initiated.

If the integer is too large to fit in the destination field, **WSTIX** discards high-order digits until the integer fits. Then it stores the remaining low-order digits and sets CARRY to 1.

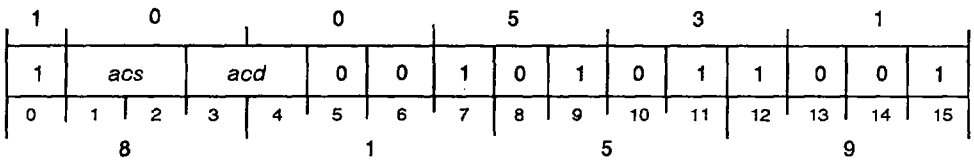
If the integer does not completely fill the destination field, the high-order digits are set to 0.

Example

```
XNLDA  1,TYPE       ;AC1 contains the data type indicator.
XLEF   3,FIELD      ;Word pointer to the integer field.
WADD   3,3          ;AC3 is a byte pointer to the integer.
WSTIX               ;Convert the contents of all four FPACs
                    ;into a commercial integer of the type
                    ;specified by AC1, at the location
                    ;specified by AC3.
```

Wide Subtract WSUB

WSUB *acs,acd*



Function: *acd - acs -> acd*

Parameters: None

WSUB subtracts the signed 32-bit integer contained in *acs* from the signed 32-bit integer contained in *acd*, placing the result in *acd*.

Arguments

acs	Before execution, contains signed 32-bit integer.
	After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd	Before execution, contains signed 32-bit integer.
	After execution, contains result.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Set according to value of ALU carry.
Overflow	1 if ALU overflow
PC	PC + 1
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

SUB	Subtract
NSUB	Narrow Subtract

Exceptions

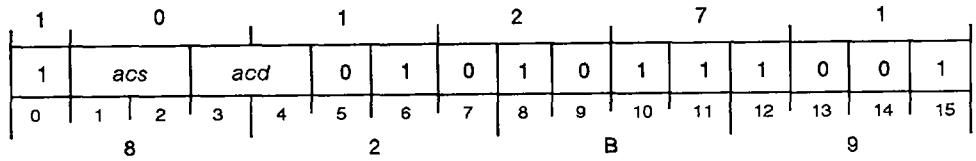
If an overflow occurs, PSR(OVR) is set to 1.

Example

```
WSUB   3,3          ;Get a zero
```

Wide Skip on Zero Bit — WSZB

WSZB *acs,acd*
(bit = 1 return)
(bit = 0 return)



Function: If (E) bit = 0 then skip

Parameters: *acs* = base word pointer -> unchanged
acd = word offset & bit identifier -> unchanged

WSZB forms a bit pointer from the contents of *acs* and *acd* and skips if the addressed bit is 0.

Arguments

acs — Before execution, contains high-order bits of bit pointer.

If *acs* and *acd* are the same accumulator, the high-order bits are treated as if they were zero in the current ring.

After execution, contents unchanged.

acd — Before execution, contains low-order bits of bit pointer.

After execution, contents unchanged.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1 (bit = 1) PC + 2 (bit = 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WSNB	Wide Skip on Nonzero Bit
WSZBO	Wide Skip on Zero Bit and Set to One

Exceptions

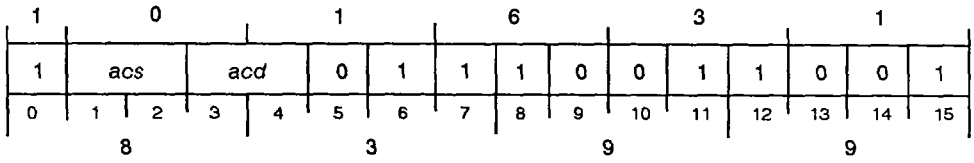
None

Example

```
XLEF   0,FLAGS      ;Get word address of flags word.
NLDAI  3,1          ;Get a 3 in AC1.
WSZB   0,1          ;Is bit 3 of the flags word set?
WBR    SET          ;Yes.
. . .               ;No.
...
FLAGS: .WORD 0      ;Flags word.
```

Wide Skip on Zero Bit and Set Bit to One WSZBO

WSZBO *acs,acd*
(bit = 1 return)
(bit = 0 return)



Function: If (E)bit = 0 then skip
1 -> (E)bit

Parameters: *acs* = base word pointer -> unch
acd = word offset & bit identifier -> unch

WSZBO forms a bit pointer from the contents of *acs* and *acd*. If the addressed bit is 0, it sets the bit to 1 and skips the next sequential word.

WSZBO facilitates the use of bit maps for allocation of facilities like memory blocks and I/O devices to several processes or tasks, that may interrupt one another. **WSZBO** is also useful in a multiprocessor environment. The bit is tested and set to 1 atomically.

Arguments

acs Before execution, contains high-order bits of bit pointer.

If *acs* and *acd* are the same accumulator, the high-order bits are treated as if they were zero in the current segment.

After execution, contents unchanged.

acd Before execution, contains low-order bits of bit pointer.

After execution, contents unchanged.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *acs* and *acd*; otherwise unused.	
CARRY	Unchanged	
Overflow	0	
PC	PC + 1	(bit = 1)
	PC + 2	(bit = 0)
PSR	Unchanged	
Stack	Unchanged	

Related Instructions

WSZB Wide Skip on Zero Bit

WSNB Wide Skip on Nonzero Bit

Exceptions

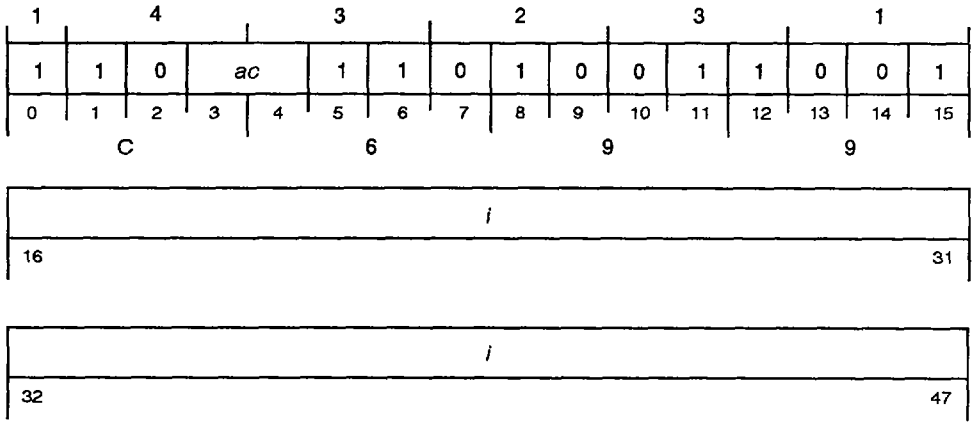
None

Example

```
;This subroutine deques 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
;                              <return>
;      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 dequeueing element 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!
```

Wide Unsigned Skip if AC Greater than Immediate — WUGTI

WUGTI *i,ac*
(*ac* <= *i* return)
(*ac* > *i* return)



Function: If *ac* > *i* then skip

Parameters: None

WUGTI performs an unsigned comparison of *ac* to 32-bit immediate field and skips depending on result.

Arguments

i	unsigned 32-bit integer
ac	Before execution, contains unsigned 32-bit integer. After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 3 (*ac* <= *i*) PC + 4 (*ac* > *i*)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WULEI	Wide Unsigned Skip if AC Less Than or Equal to Immediate

Exceptions

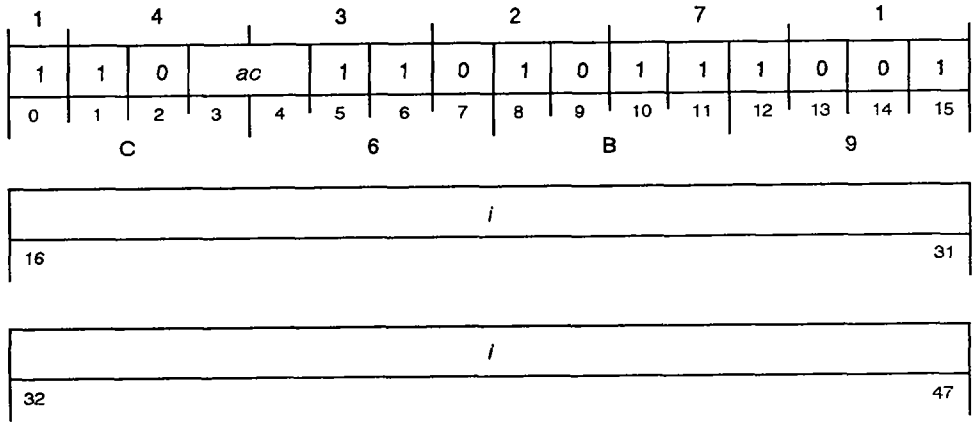
None

Example

```
WUGTI 016000000000,2          ;Is AC2 greater than the constant?
WBR   LESS_EQUAL              ;No. AC2 is <= constant.
. . .                         ;Yes. AC2 is > constant.
```

Wide Unsigned Skip if AC Less than or Equal to Immediate — WULEI

WULEI *i,ac*
(*ac* > *i* return)
(*ac* <= *i* return)



Function: If *ac* <= *i* then skip

Parameters: None

WULEI performs an unsigned comparison of the contents of *ac* to the 32-bit immediate integer and skips depending on the result.

Arguments

i	Unsigned 32-bit integer.
ac	Before execution, contains unsigned 32-bit integer.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 3 (*ac* > *i*) PC + 4 (*ac* <= *i*)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WUGTI	Wide Unsigned Skip if AC Greater Than Immediate

Exceptions

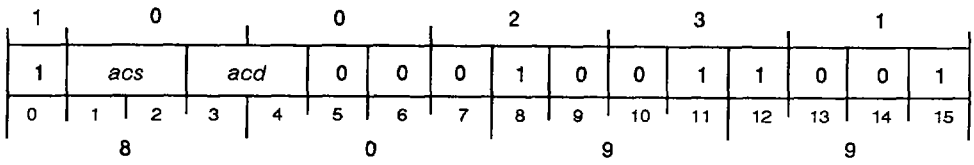
None

Example

```
WULEI 016000000000,2 ;Is AC2 less than or equal to the constant?
WBR   LESS_EQUAL     ;No. AC2 is > constant.
. . .                ;Yes. AC2 is <= constant.
```

Wide Unsigned Skip if Greater than or Equal to — WUSGE

WUSGE *acs,acd*
(*acs* < *acd* return)
(*acs* >= *acd* return)



Function: If *acs* > = *acd* then skip

Parameters: None

NOTE: Compares unsigned numbers

WUSGE performs an unsigned comparison of an unsigned integer in *acs* to an unsigned integer in *acd* and skips if the first is greater than or equal to the second.

If *acs* and *acd* are the same accumulator, then the instruction compares the integer contained in the accumulator to 0.

Arguments

acs	Before execution, contains unsigned 32-bit integer.
	After execution, contents unchanged.
acd	Before execution, contains unsigned 32-bit integer.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1 (*acs* < *acd*) PC + 2 (*acs* >= *acd*)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WUSGT	Wide Unsigned Skip if Greater than

Exceptions

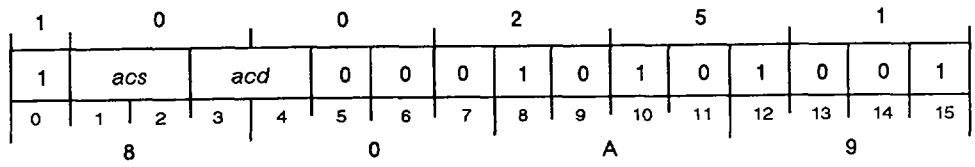
If *acs* and *acd* are the same accumulator, **WUSGE** always skips.

Example

```
WUSGE 1,2          ;Is contents of AC1 >= contents of AC2?
WBR   AC2_GREATER  ;AC2 is greater.
. . .              ;AC1 is greater than or equal.
```

Wide Unsigned Skip if Greater than — WUSGT

WUSGT *acs,acd*
(*acs* <= *acd* return)
(*acs* > *acd* return)



Function: If *acs* > *acd* then skip

Parameters: None

NOTE: compares unsigned numbers
If *acd* is *acs*, *acs* is compared with 0

WUSGT compares one unsigned integer in *acs* to another in *acd* and skips if the first is greater than the second.

If *acs* and *acd* are the same accumulator, then the instruction compares the integer contained in the accumulator to 0.

Arguments

acs	Before execution, contains unsigned 32-bit integer.
	After execution, contents unchanged.
acd	Before execution, contains unsigned 32-bit integer.
	After execution, contents unchanged.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1 (*acs* <= *acd*) PC + 2 (*acs* > *acd*)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WUSGE	Wide Unsigned Skip if Greater then or Equal to

Exceptions

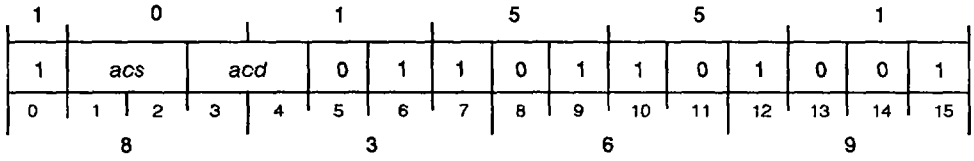
None

Example

```
WUSGT 1,2          ;Is contents of AC1 > contents of AC2?
WBR   AC2_GREATER  ;AC2 is greater than or equal.
. . .              ;AC1 is greater.
```

Wide Exchange WXCH

WXCH *acs,acd*



Function: *acs <-> acd*

Parameters: None

WXCH exchanges the 32-bit contents of two accumulators.

Arguments

acs	Before execution, contains 32-bit value.
	After execution, contains 32-bit value from *acd*.
acd	Before execution, contains 32-bit value.
	After execution, contains 32-bit value from *acs*.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

XCH	Exchange Accumulators

Exceptions

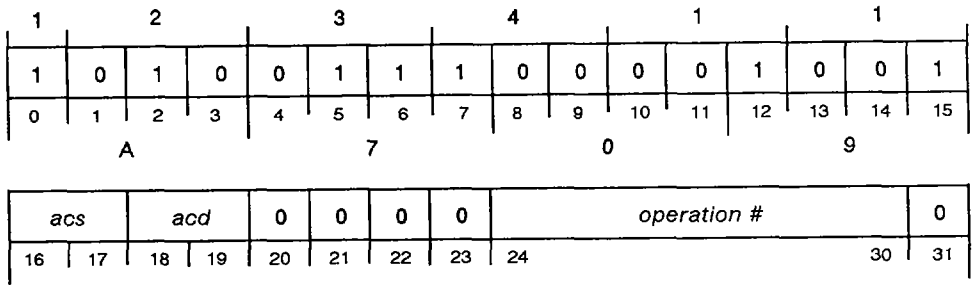
None

Example

```
WXCH  0,2          ;Exchange the contents of AC0 and AC2.
```

Wide Extended Operation WXOP

WXOP *acs,acd,operation #*



Function: 6 double words -> wide stack (wide return block)
(E) -> PC
address of *acs* in stack -> AC2
address of *acd* in stack -> AC3

Parameters: (12-13)page zero = (table) -> unch
E = (*operation #* *2) + (12-13)page zero -> unch

WXOP pushes a return block onto the wide stack and transfers control to a procedure pointed to by the selected address in an extended operations table (**WXOP** table). It is an efficient way to transfer control from one procedure to another.

The return block has the following contents:

Double Word Pushed	Contents
1	PSR (in bits 0-15; bits 16-31 set to 0)
2	AC0
3	AC1
4	AC2
5	AC3
6	CARRY in bit 0; (WXOP address + 2) in bits 1-31

After the return block is pushed, the stack address of the stored accumulator designated as *acs* is loaded into AC2, and the stack address of the stored accumulator designated as *acd* is loaded into AC3.

WXOP then uses the specified operation number as an offset to the starting address of the **WXOP** table from which it fetches the intermediate address of the extended operations procedure. The **WXOP** table is an array of double words, each of which is an address. The resulting effective address is loaded into the PC as the starting address of the new procedure.

The **WXOP** table can contain up to 200_8 procedure entry points (intermediate addresses). Its starting address is stored in page zero locations 12_8-13_8 of reserved memory.

The contents of the **WXOP** table starting address are unaffected. All addresses must be in the current segment.

Arguments

acs	Specifies accumulator whose address on wide stack is stored into AC2.
acd	Specifies accumulator whose address on wide stack is stored into AC3.
operation #	Unsigned 7-bit integer specifying offset from **WXOP** table starting address.

Registers, Flags, and Stacks

AC0	Second double word pushed.
AC1	Third double word pushed.
AC2	Fourth double word pushed.
	After execution, contains stack address of pushed contents of *acs*.
AC3	Fifth double word pushed.
	After execution, contains stack address of pushed contents of *acd*.
CARRY	Unchanged
Overflow	0
PC	Effective address derived from address fetched from table.
PSR	Unchanged
Stack	Wide stack pointer incremented by six double words.

Related Instructions

XOP0	Extended Operation
WPOPB	Use the Wide Pop Block instruction to restore pushed values and return.

Exceptions

If a stack overflow occurs, the stack fault routine is executed.

Example

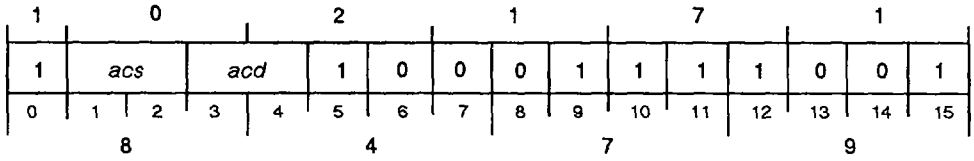
```
        .TITLE WXOP
        .RDX   16
        .RDXO  16
        .ENT START, ERROR, BYE, OP0, OP4, OP5, OP6, OPER_A, OPER_B,OPER_C
              AC0 =         0
              AC1 =         1
              AC2 =         2
              AC3 =         3
        ;
        ;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
        ;
        ;This program is a small subset of a calculator program. WXOP is used
        ;to implement calls to subroutines which perform various operations.
        ;WXOP is highly suited for a use such as this where the subroutines
        ;require few parameters, and can easily be represented by a number.

        ;Set up PAGE ZERO, WXOP table pointer
              .LOC          0A
              TABLE
        ;
              .LOC          50              ;error destination
ERRFLG:       ?RFCF+?RFER
ERROR:        XWLDA         2,ERRFLG
              ?RETURN
        ;
        ;good completion
              .LOC          7F
BYE:          WSUB          2,2
              ?RETURN
```

```
; start of program
START:    LLEF        0,STACK-2    ;Set up the Stack Parameters
          STASB       0            ;Stack Base
          STASP       0            ;Stack Pointer
          STAFP       0            ;Frame Pointer
          LLEF        0,STKEND
          STASL       0            ;Stack Limit
    ;
    ;perform the operations
          WXOP        AC0,AC0,0    ;MUL
          WXOP        AC0,AC0,4    ;MOV C to A
          WXOP        AC0,AC0,6    ;ADD
          WBR BYE
    ;
    ;bunch of errors
          WBR ERROR
          WBR ERROR
          WBR ERROR
    ;
    ;     perform M O V E operation
    ;     A <- C
OP4:      XWLDA       0,OPER_C
          XWSTA       0,OPER_A
          WPOPB
    ;     B <- C
    ;
OP5:      XWLDA       0,OPER_C
          XWSTA       0,OPER_B
          WPOPB
    ;
    ;     perform A D D operation
    ;     C <- A + B
    ;
OP6:      XWLDA       0,OPER_A
          XWLDA       1,OPER_B
          WADD        0,1
          XWSTA       1,OPER_C
          WPOPB
    ;
    ;     perform M U L T I P L Y operation
    ;     C <- A * B
    ;
OP0:      XWLDA       0,OPER_A
          XWLDA       1,OPER_B
          WMUL        0,1
          XWSTA       1,OPER_C
          WPOPB
    ;
    ; WXOP table
TABLE:    OP0         ;MUL
          ERROR       ;            DIV
          ERROR
          ERROR
          OP4         ;MOVE C to A
          OP5         ;MOVE C to B
          OP6         ;ADD
    ;
    ;     Variables
OPER_A:   25.
OPER_B:   3.
OPER_C:   0.
    ;
    ; stack
STACK:    .BLK        66.
STKEND:   .BLK        48.
          .END        START
```

Wide Exclusive OR WXOR

WXOR *acs,acd*



Function: *acs* XOR *acd* -> *acd*

Parameters: None

WXOR forms the logical Exclusive OR between corresponding bits of *acs* and *acd*, placing the result into *acd*.

Arguments

acs	Before execution, contains 32-bit value.
	After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd	Before execution, contains 32-bit value.
	After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

XOR	Exclusive OR
IOR	Inclusive OR
WIOR	Wide Inclusive OR

Exceptions

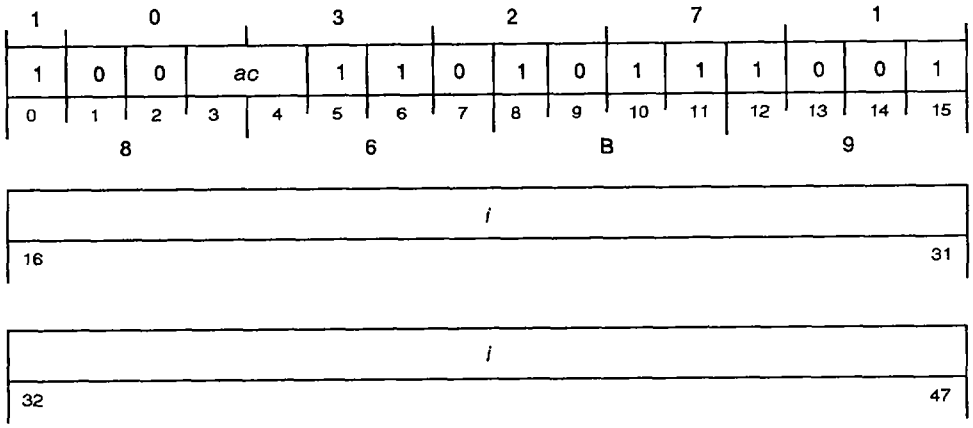
None

Example

```
WXOR  2,2          ;Set AC2 to all zeros.
```

Wide Exclusive OR Immediate WXORI

WXORI *i,ac*



Function: *i* XOR *ac* -> *ac*

Parameters: None

WXORI forms the logical exclusive OR of corresponding bits of the specified accumulator and the value contained in the immediate field, placing the result in the specified accumulator.

Arguments

i	32-bit value
ac	Before execution, contains 32-bit value.
	After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 3
PSR	Unchanged
Stack	Unchanged

Related Instructions

XORI	Exclusive OR Immediate

Exceptions

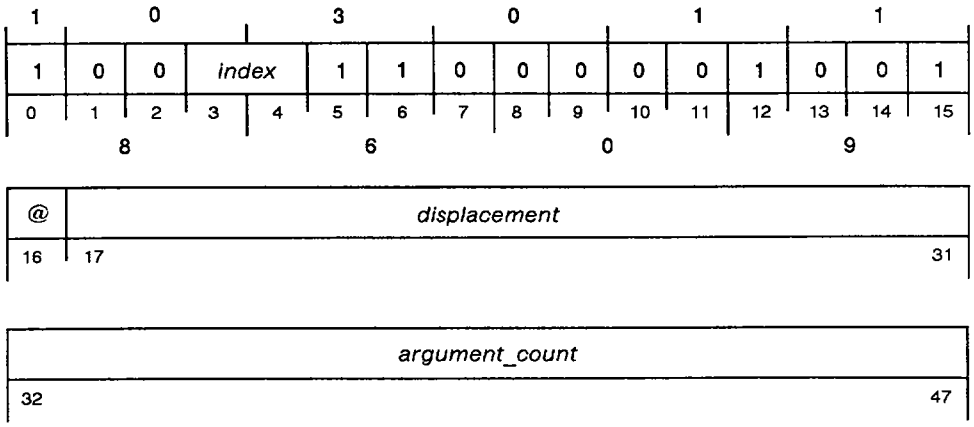
None

Example

```
WXORI 377,0          ;Take the one's complement of the low order
                     ;byte of AC0.
```

Call Subroutine (Extended Displacement) XCALL

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



Function: PC + 3 -> AC3
If E = valid, E -> PC
0 -> OVR

Parameters: None

NOTE: Valid E = inward ring cross and legal gate. If *argument_count* bit 0 = 0, PSR and *argument_count* pushed onto stack.

XCALL transfers program control to a subroutine in the current segment or through a gate array in a lower-numbered segment to a subroutine in that lower-numbered segment.

XCALL loads the program counter, plus three, into AC3. If the effective address (target address) is legal, the instruction checks the *argument count* field. **XCALL** then sets PSR(OVR) to zero and loads the program counter with the target address.

For more information on **XCALL**, refer to "Transferring Program Control to Another Segment" in the Program Flow Management chapter.

Arguments

[@]*displacement*[,*index*]

Effective address (target address) generated by instruction may specify the current ring, an inner ring, or an outer ring.

If target address specifies an *outward ring* crossing, protection fault occurs and error code 7 placed in AC1. PC in return block undefined.

If target address specifies an *inward ring* call, **XCALL** assumes target address has following format:

Bits 1–3 contain new segment number.
Bits 4–15 unused.
Bit 16 must contain 0 or results undefined.
Bits 17–31 contain gate number in inner ring.

If gate is illegal, a protection fault occurs, error code 6 placed in AC1, and no subroutine call made. PC in return block undefined.

If gate legal, or target address specifies the *current ring*, **XCALL** then checks *argument count* field.

[*argument_count*]

Contains 16-bit value specifying number of arguments pushed onto stack. **XCALL** creates a PSR/*argument count* double word depending on value of high bit (bit 48) of *argument count*.

If high bit is 0, **XCALL** pushes onto the wide stack a double word with the following format:

Bits 0–15 contain current PSR.
Bits 16–31 contain *argument count*.

If high bit is 1 (negative), **XCALL** assumes top double word of wide stack has following format:

Bits 0–15 undefined.
Bits 16–31 contain *argument count* with bit 16 = 0.

The instruction uses the wide stack double word and ignores *argument count* coded with **XCALL** instruction. The instruction then places current PSR into bits 0–15 of stack double word.

(If target address is in inner segment, **XCALL** copies the number of double words specified in *argument count* from the outer segment stack to the inner segment stack, and then pushes the PSR/*argument count* double word onto inner stack.)

Registers, Flags, and Stacks

AC0–AC2	Unused
AC3	After execution, contains PC + 3 (always references current segment)
CARRY	Unchanged
Overflow	Unaffected
PC	Target address
PSR	OVR set to 0
Stack	Wide stack in current segment contains arguments. If target address is current segment, wide stack also contains PSR/*argument count* double word. If target address is inner segment, inner segment wide stack contains copy of arguments and PSR/*argument count* double word.

Related Instructions

WRTN Pops six double words from wide stack. Generally, should be the last instruction in the subroutine.

WSAVR, WSAVS
Push five double words onto the wide stack. Generally, should be the first instruction in the subroutine.

Exceptions

If the target address specifies an outward ring crossing, or an inward ring call with an illegal gate, a protection fault occurs. The processor pushes a fault–return block onto the wide stack in the current segment (PC contents are undefined), loads AC1 with an error code, and transfers program control to the protection violation fault handler.

If a wide stack overflow occurs while **XCALL** is pushing the PSR/*argument count* double word, a stack overflow occurs. The processor clears the PSR, pushes a fault return block (PC contents are undefined) onto the wide stack in the destination segment, loads AC1 with an error code, and transfers program control to the wide stack fault handler in the destination segment.

The error codes returned to AC1 are:

Error Code	Description	PC
2	Wide stack overflow	Wide stack fault handler
3	Invalid segment	Protection violation fault handler
6	Invalid gate	Protection violation fault handler
7	Illegal outward call	Protection violation fault handler

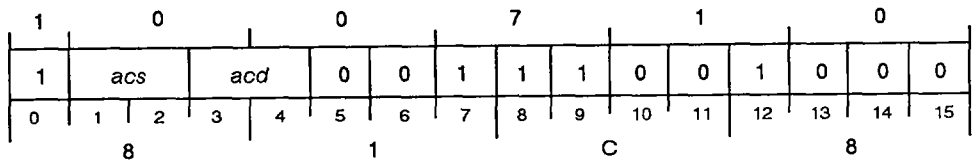
Example

```
    XCALL 4S3+2,0,6     ;XCALL transfers program control to segment 4
                        ;through the second element in the gate array.
                        ;(Second element contains the address of
                        ;INET.) XCALL passes 6 arguments to the
INET:     WSAVS 5       ;subroutine.
          .
          .
          .
          WRTN
```

Exchange Accumulators XCH

ECLIPSE Instruction

XCH *acs,acd*



Function: *acs* <-> *acd*

Parameters: None

XCH exchanges the 16-bit contents of two accumulators.

Arguments

acs(16-31)	Before execution, contains 16-bit value.
	After execution, contains 16-bit value from *acd*.
acd(16-31)	Before execution, contains 16-bit value.
	After execution, contains 16-bit value from *acs*.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

WXCH	Wide Exchange

Exceptions

None

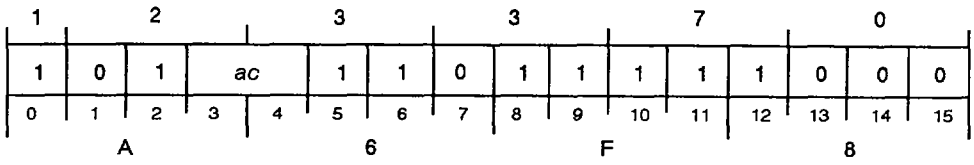
Example

```
XCH   0,2          ;Exchange the contents of AC0[16-31] and
                   ;AC2[16-31].
```

Execute XCT

ECLIPSE Instruction

XCT *ac*



Function: execute(*ac*)

Parameters: *ac* = instruction -> unchanged

NOTE: If *ac* = 1st word of {2,3,4}-word instruction;
(XCT) + {1,2,3} = {2nd,3rd,4th} word

XCT executes the contents of an accumulator as an instruction, treating the *ac* contents as if it were in main memory in the location occupied by the **XCT** instruction.

If the specified accumulator contains the first word of a multiple-word instruction, the words following the **XCT** instruction are used as the remainder of the instruction. Normal sequential operation then continues with the word following these.

Do not use the **XCT** instruction to execute an instruction that requires all four accumulators, such as **CMV, CMT, CMP, CTR,** or **BAM.**

Arguments

ac(16-31) Before execution, contains 16-bit instruction or first word of multiple-word instruction.

After execution, contents unchanged unless modified by *ac* instruction.

Registers, Flags, and Stacks

AC0-AC3 Can be individually specified as *ac*; otherwise unused.

CARRY Unchanged by **XCT**, but possibly affected by executed instruction.

Overflow 0 (but possibly affected by executed instruction)

PC PC + n (number of words in *ac* instruction)

PSR Unchanged by **XCT**, but possibly affected by executed instruction.

Stack Unchanged by **XCT**, but possibly affected by executed instruction.

Related Instructions

Load accumulator
Use narrow load accumulator instructions to place an instruction in *ac*.

Exceptions

If *ac* contains an instruction that modifies *ac*, then the results of **XCT** are undefined.

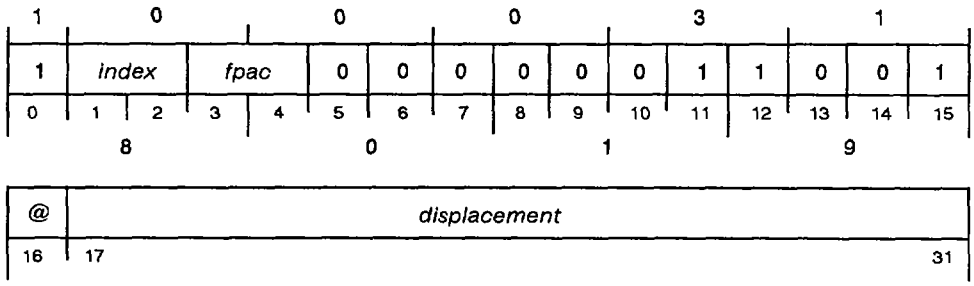
If the instruction in *ac* is an Execute instruction that specifies the same *ac*, the processor is placed in a one-instruction loop. Because of this possiblility, this instruction is interruptible. An I/O interrupt can occur immediately prior to each time the instruction in *ac* is executed. If an I/O interrupt does occur, the program counter in the return block pushed onto the system stack points to the **XCT** instruction in main memory. This manner of executing an **XCT** instruction makes a "wait for I/O interrupt" instruction.

Example

```
    XLEF   2,IO_TABLE   ;Get the I/O instruction table address.
    WADD   1,2          ;Add the index into the table.
    XNLDA  0,0,2        ;Get the instruction.
    XCT    0            ;Execute it.
    .
    .
    .
IO_TABLE:               ;Table is indexed by an integer that
           DIA 0,27     ;determines which of these instructions
           DIB 0,27     ;is executed.
           DIC 0,27
           DOA 0,27
           DOB 0,27
           DOC 0,27
```

Add Double (Memory to FPAC) (Extended Displacement) XFAMD

XFAMD *fpac*,[@]*displacement*[,*index*]



Function: (E) + *fpac* -> *fpac*

Parameters: None

XFAMD adds a 64-bit floating-point number contained in the specified memory location to the 64-bit floating-point number in the specified *fpac* and places the normalized result into the *fpac*.

Arguments

fpac Before execution, contains 64-bit floating-point number.

After execution, contains normalized 64-bit result.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0–FPAC3 Can be individually specified as *fpac*; otherwise unused.

PC PC + 2

FPSR Updated Z and N flags.

Stack Unchanged

Related Instructions

FAMD, LFAMD, FAMS, XFAMS, LFAMS
Add the contents of memory to a floating-point accumulator.

Exceptions

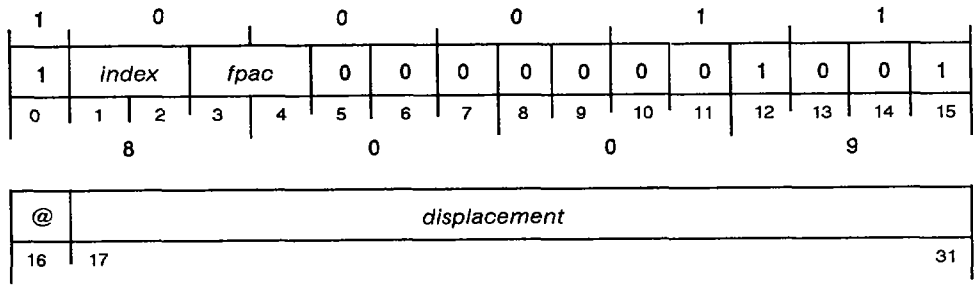
If addition produces an exponent overflow, the processor sets the FPSR(OVF) flag to 1 and terminates the instruction.

Example

```
XFLDD 2,FLPT1      ;Add the two double precision
XFAMD 2,FLPT2      ;numbers at locations FLPT1 and FLPT2,
XFSTD 2,FLPT3      ;and store the result at location FLPT3.
```

Add Single (Memory to FPAC) (Extended Displacement) XFAMS

XFAMS *fpac*,[@]*displacement*[,*index*]



Function: (E) + *fpac* -> *fpac*

Parameters: None

XFAMS adds a 32-bit floating-point number contained in the specified memory location to the 32-bit floating-point number in the specified *fpac* and places the normalized result into the *fpac*.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contains normalized 32-bit result with bits 32-63 set to 0.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
PC	PC + 2
FPSR	Updated Z and N flags.
Stack	Unchanged

Related Instructions

FAMS, LFAMS, FAMD, XFAMD, LFAMD
Add the contents of memory to a floating-point accumulator.

Exceptions

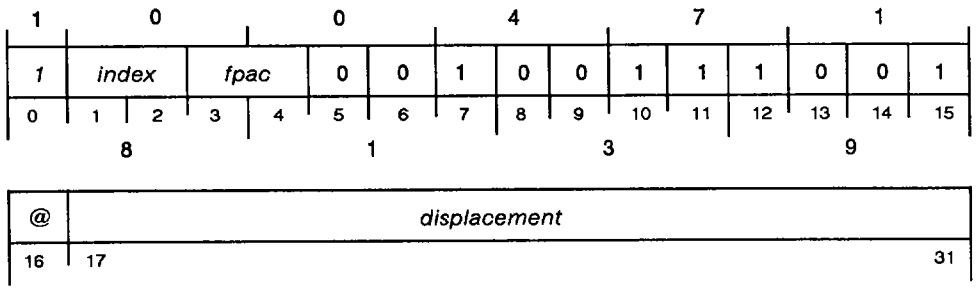
If the addition produces an exponent overflow, the processor sets the FPSR(OVF) flag to 1 and terminates the instruction.

Example

```
XFLDS 2,FLPT1      ;Add the two single-precision
XFAMS 2,FLPT2      ;numbers at locations FLPT1 and FLPT2,
XFSTS 2,FLPT3      ;and store the result at location FLPT3.
```

Divide Double (FPAC by Memory) (Extended Displacement) XFDMD

XFDMD *fpac*,[@]*displacement*[,*index*]



Function: *fpac* / (E) -> *fpac*

Parameters: None

XFDMD divides the 64-bit floating-point number in the specified *fpac* by a 64-bit floating-point number contained in the specified memory location and places the normalized result into the *fpac*.

Arguments

fpac Before execution, contains 64-bit floating-point number.

After execution, contains normalized 64-bit result.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
PC	PC + 2
FPSR	Updated Z and N flags.
Stack	Unchanged

Related Instructions

FDMD, LFDMD, FDMS, XFDMS, LFDMS
Divide a floating-point accumulator by the contents of memory.

Exceptions

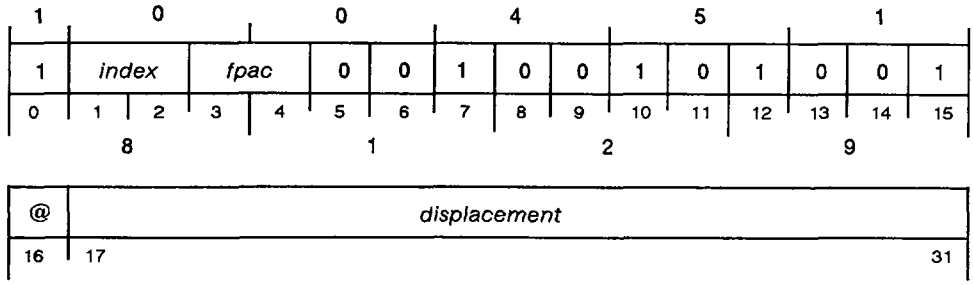
If the divisor (in memory) is zero, the processor sets the FPSR(INV) flag to 1, places error code 0 in the FPSR(INP) bits and the address of the instruction in the FPSR(FPPC), and terminates the instruction.

Example

```
XFLDD 1,DATA1      ;Divide the double precision number at
XFDMD 1,DATA2      ;location DATA1 by the double precision
XFSTD 1,RESULT     ;number at location DATA2, and store the
                   ;result at location RESULT.
```

Divide Single (FPAC by Memory) (Extended Displacement) XFDMS

XFDMS *fpac*,[@]*displacement*[,*index*]



Function: *fpac* / (E) -> *fpac*

Parameters: None

XFDMS divides the 32-bit floating-point number in the specified *fpac* by a 32-bit floating-point number contained in the specified memory location and places the normalized result into the *fpac*.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contains normalized 32-bit result with bits 32-63 set to 0.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
PC	PC + 2
FPSR	Updated Z and N flags.
Stack	Unchanged

Related Instructions

FDMS, LFDMS, FDMD, XFDMD, LFDMD
Divide a floating-point accumulator by the contents of memory.

Exceptions

If the divisor (in memory) is zero, the processor sets the FPSR(INV) flag to 1, places error code 0 in the FPSR(INP) bits and the address of the instruction in the FPSR(FPPC), and terminates the instruction.

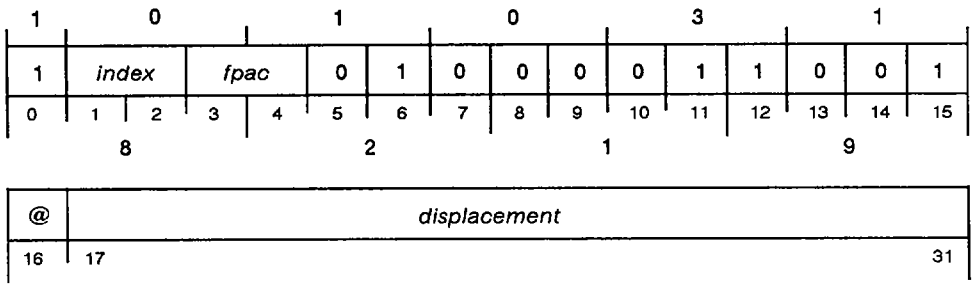
Example

```
XFLDS 3,DIVDND     ;Divide the single precision number at
XFDMS 3,DIVSOR     ;location DIVDND by the single precision
XFSTS 3,QUOTNT     ;number at location DIVSOR, and store
                   ;the result at location QUOTNT.
```

QUOTNT = DIVDND / DIVSOR

Load Floating-Point Double (Extended Displacement) XFLDD

XFLDD *fpac*,[@]*displacement*[,*index*]



Function: (E) -> *fpac*

Parameters: None

XFLDD loads a 64-bit floating-point number from the specified memory location into the specified *fpac*. Unnormalized data is moved without change.

Arguments

fpac After execution, contains 64-bit floating-point number.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be individually specified as *fpac*; otherwise unused.

PC PC + 2

FPSR Updated Z and N flags; undefined for unnormalized data.

Stack Unchanged

Related Instructions

FLDD, LFLDD, FLDS, XFLDS, LFLDS
Load a floating-point accumulator with the contents of memory.

Exceptions

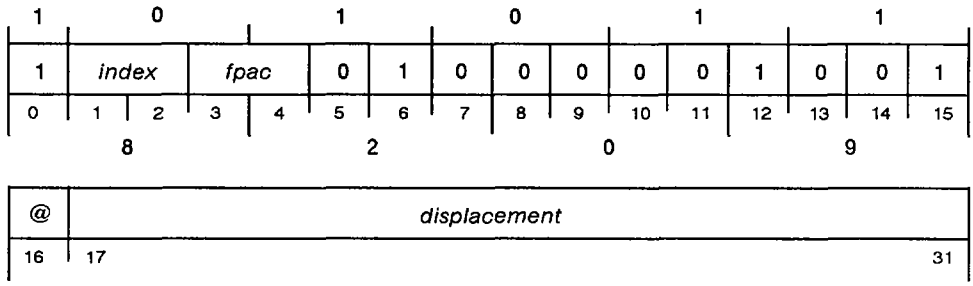
None

Example

```
XFLDD 2,FLPT2      ;Load the double precision number at
                   ;memory location FLPT2 into FPAC2.
```

Load Floating-Point Single (Extended Displacement) XFLDS

XFLDS *fpac*,[@]*displacement*[,*index*]



Function: (E) -> *fpac*

Parameters: None

XFLDS loads a 32-bit floating-point number from the specified memory location into the specified *fpac*. Unnormalized data is moved without change.

Arguments

fpac(0-31) After execution, contains 32-bit floating-point number with bits 32-63 set to 0.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be individually specified as *fpac*; otherwise unused.

PC PC + 2

FPSR Updated Z and N flags; undefined for unnormalized data.

Stack Unchanged

Related Instructions

FLDS, LFLDS, FLDD, XFLDD, LFLDD
Load a floating-point accumulator with the contents of memory.

Exceptions

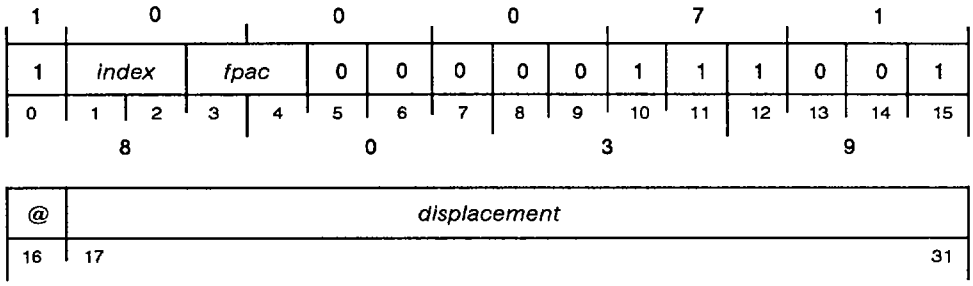
None

Example

```
XFLDS 1,VALUE      ;Load the single precision number at
                   ;memory location VALUE into FPAC1.
```

Multiply Double (FPAC by Memory) (Extended Displacement) XFMMD

XFMMD *fpac*,[@]*displacement*[,*index*]



Function: *fpac* * (E) -> *fpac*

Parameters: None

XFMMD multiplies a 64-bit floating-point number contained in the specified memory location by the 64-bit floating-point number in the specified *fpac* and places the normalized result into the *fpac*.

Arguments

fpac Before execution, contains 64-bit floating-point number.

After execution, contains normalized 64-bit result.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be individually specified as *fpac*; otherwise unused.

PC PC + 2

FPSR Updated Z and N flags.

Stack Unchanged

Related Instructions

FMMD, LFMMD, FMMS, XFMMS, LFMMS
Multiply a floating-point accumulator by the contents of memory.

Exceptions

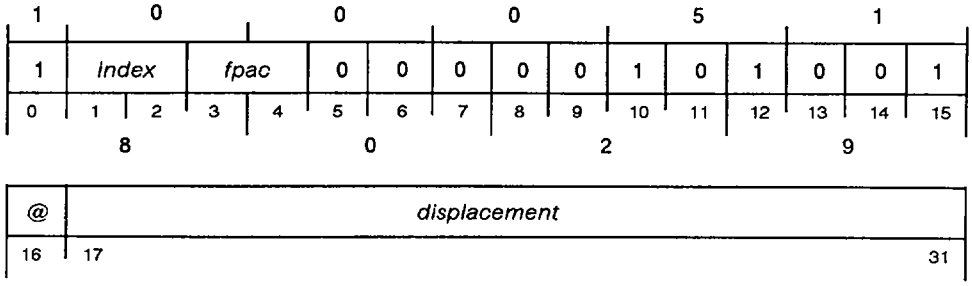
If multiplication produces an exponent overflow, the processor sets the FPSR(OVF) flag to 1, and terminates the instruction.

Example

```
XFLDD 2,FLOATX     ;Multiply the two double precision
XFMMD 2,FLOATY     ;numbers at locations FLOATX and FLOATY,
XFSTD 2,FLOATZ     ;and store the result at location FLOATZ.
```

Multiply Single (FPAC by Memory) (Extended Displacement) XFMMS

XFMMS *fpac*,[@]*displacement*[,*index*]



Function: *fpac* * (E) -> *fpac*

Parameters: None

XFMMS multiplies a 32-bit floating-point number contained in the specified memory location by the 32-bit floating-point number in the specified *fpac* and places the normalized result into the *fpac*.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contains normalized 32-bit result with bits 32-63 set to 0.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
PC	PC + 2
FPSR	Updated Z and N flags.
Stack	Unchanged

Related Instructions

FMMS, LFMMS, FMMD, XFMMD, LFMMD
Multiply a floating-point accumulator by the contents of memory.

Exceptions

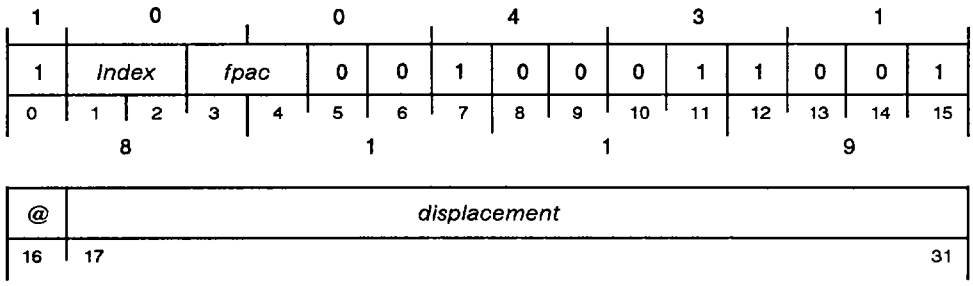
If multiplication produces an exponent overflow, the processor sets the FPSR(OVF) flag to 1, and terminates the instruction.

Example

```
XFLDS 0,DATA1      ;Multiply the two single precision
XFMMS 0,DATA2      ;numbers at locations DATA1 and DATA2,
XFSTS 0,DATA3      ;and store the result at location DATA3.
```

Subtract Double (Memory from FPAC) (Extended Displacement) XFSMD

XFSMD *fpac*,[@]*displacement*[,*index*]



Function: *fpac* - (E) -> *fpac*

Parameters: None

XFSMD subtracts a 64-bit floating-point number contained in the specified memory location from the 64-bit floating-point number in the specified *fpac*, and places the normalized result into the *fpac*.

Arguments

fpac Before execution, contains 64-bit floating-point number.

After execution, contains normalized 64-bit result.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be individually specified as *fpac*; otherwise unused.

PC PC + 2

FPSR Updated Z and N flags.

Stack Unchanged

Related Instructions

FSMD, LFSMD, FSMS, XFSMS, LFSMS
Subtract the contents of memory from a floating-point accumulator.

Exceptions

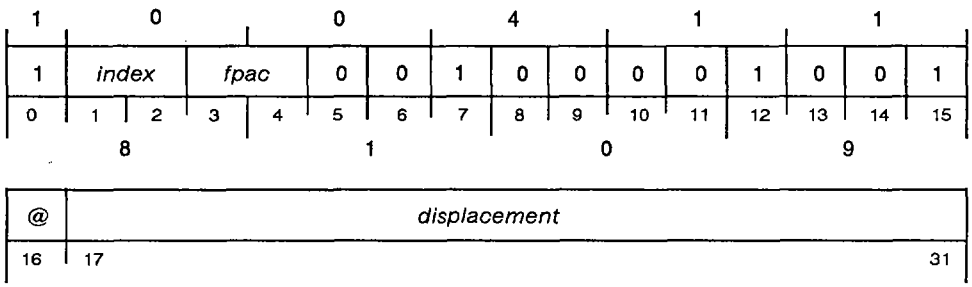
If subtaction produces an exponent overflow or underflow, the processor sets the approriate FPSR flag to 1, and terminates the instruction.

Example

```
XFLDD 1,X        ;Subtract the double precision number at
XFSMD 1,Y        ;location Y from the double precision
XFSTD 1,Z        ;number at location X, and store the
                 ;result at location Z.
```

Subtract Single (Memory from FPAC) (Extended Displacement) XFSMS

XFSMS *fpac*,[@]*displacement*[,*index*]



Function: *fpac* - (E) -> *fpac*

Parameters: None

XFSMS subtracts a 32-bit floating-point number contained in the specified memory location from the 32-bit floating-point number in the specified *fpac*, and places the normalized result into the *fpac*.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contains normalized 32-bit result with bits 32-63 set to 0.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3	Can be individually specified as *fpac*; otherwise unused.
PC	PC + 2
FPSR	Updated Z and N flags.
Stack	Unchanged

Related Instructions

FSMS, LFSMS, FSMD, XFSMD, LFSMD
Subtract the contents of memory from a floating-point accumulator.

Exceptions

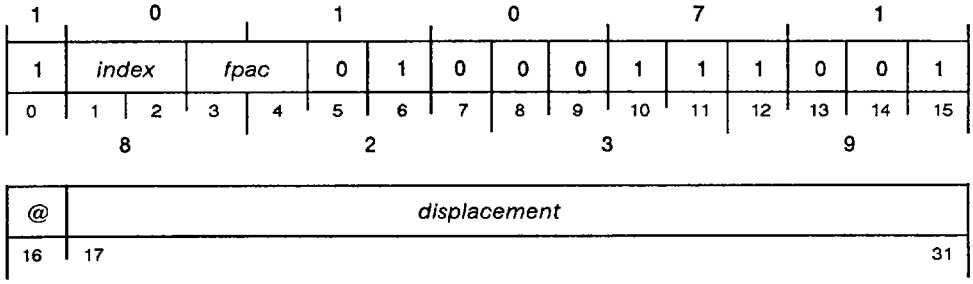
If subtraction produces an exponent overflow or underflow, the processor sets the appropriate FPSR flag to 1, and terminates the instruction.

Example

```
XFLDS 3,FIRST      ;Subtract the single precision number at
XFSMS 3,SECOND     ;location SECOND from the single precision
XFSTS 3,THIRD      ;number at location FIRST, and store the
                   ;result at location THIRD.
```

Store Floating-Point Double (Extended Displacement) XFSTD

XFSTD *fpac*,[@]*displacement*[,*index*]



Function: *fpac* -> (E)

Parameters: None

XFSTD stores the 64-bit contents of the specified floating-point accumulator into four sequential 16-bit memory locations, with the beginning address specified by the arguments. Unnormalized data is moved without change.

Arguments

fpac Before execution, contains 64-bit floating-point number.

After execution, contents unchanged.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be individually specified as *fpac*; otherwise unused.

PC PC + 2

FPSR Unchanged

Stack Unchanged

Related Instructions

FSTD, LFSTD, FSTS, XFSTS, LFSTS
Store a floating-point accumulator to memory.

Exceptions

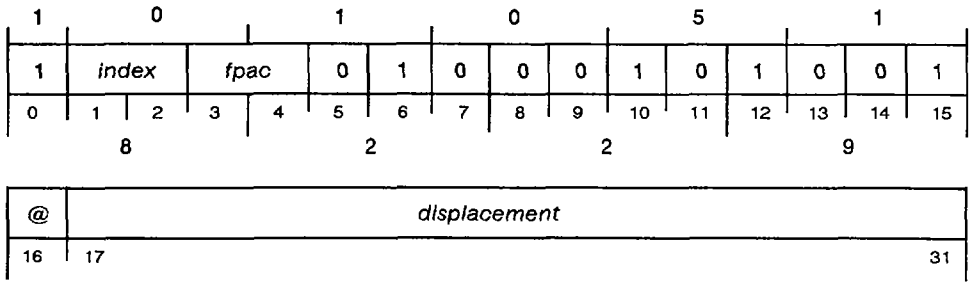
None

Example

```
FMD   1,2          ;Multiply FPAC2 by FPAC1, and store the
XFSTD 2,RSLT       ;double precision result at location RSLT.
```

Store Floating-Point Single (Extended Displacement) XFSTS

XFSTS *fpac*,[@]*displacement*[,*index*]



Function: *fpac* -> (E)

Parameters: None

XFSTS stores the high-order 32 bits of the specified floating-point accumulator into two sequential 16-bit memory locations, with the beginning address specified by the arguments. Unnormalized data is moved without change.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contents unchanged.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

FPAC0-FPAC3 Can be individually specified as *fpac*; otherwise unused.

PC PC + 2

FPSR Unchanged

Stack Unchanged

Related Instructions

FSTS, LFSTS, FSTD, XFSTD, LFSTD
Store a floating-point accumulator into memory.

Exceptions

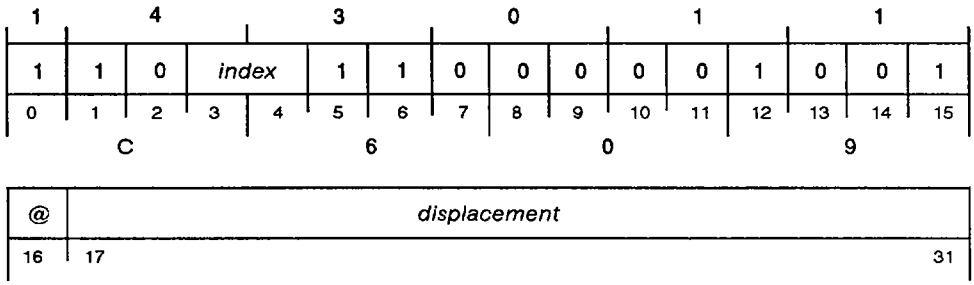
None

Example

```
FAS   0,1          ;Add FPAC0 to FPAC1, and store the single
XFSTS 1,RESULT     ;precision result at location RESULT.
```

Jump (Extended Displacement) XJMP

XJMP [@]*displacement*[,*index*]



Function: E -> PC

Parameters: None

XJMP calculates an effective address and loads it into the program counter.

Arguments

[@]*displacement*[,*index*]
Effective address generated is confined to current segment

Flags, Registers, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	Effective address
PSR	Unchanged
Stack	Unchanged

Related Instructions

JMP, EJMP, LJMP
Jump to a subroutine.

Exceptions

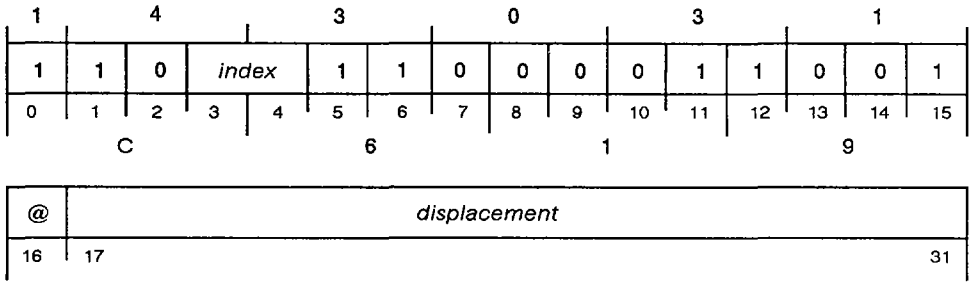
None

Example

```
    XJMP   CURR_SEG     ;Jump to a location in the current segment.
     .
     .
     .
CURR_SEG:     . . .
```

Jump to Subroutine (Extended Displacement) XJSR

XJSR [@]*displacement*[,*index*]



Function: E -> PC
PC + 2 -> AC3

Parameters: None

XJSR calculates the effective address, loads AC3 with the current 31-bit value of the program counter plus three, and transfers control to a subroutine.

Arguments

[@]*displacement*[,*index*]
Effective address generated is confined to current segment.

Registers, Flags, and Stacks

AC0–AC2	Unused
AC3	After execution, contains 31-bit value of PC(before execution) + 3
CARRY	Unchanged
Overflow	0
PC	Effective address
PSR	Unchanged
Stack	Unchanged

Related Instructions

JSR, EJSR, LJSR
Jump to a subroutine, saving a return address.

Exceptions

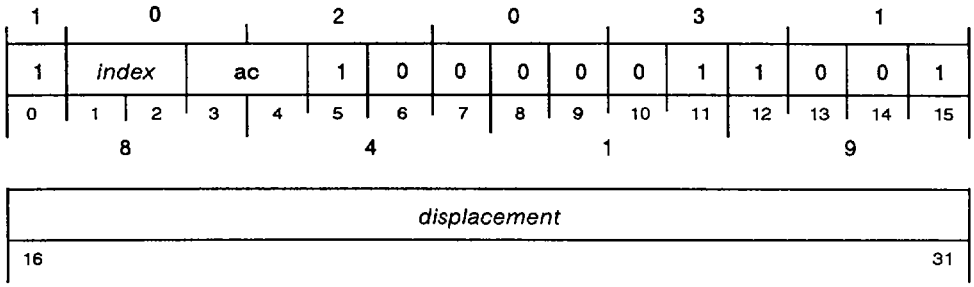
None

Example

```
      XJSR  SUBROUT      ;Jump to subroutine. Return PC is put in AC3.
      ...
SUBROUT:
            WSSVR 0      ;Save ACs and return address.
            .
            .            ;Do the subroutine.
            .
            WRTN         ;Go back to the caller. ACs are restored.
```

Load Byte (Extended Displacement) XLDB

XLDB *ac,displacement*[,*index*]



Function: (E)byte --> *ac* [bits 24-31, bits 16-23 set to 0]

Parameters: None

XLDB calculates the effective byte address and uses it to reference a byte in memory. Then it loads the addressed byte into the specified accumulator and zero-extends the value to 32 bits.

Arguments

ac(24-31) After execution, contains result of operation.

displacement[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PSR	Unchanged
PC	PC + 2
Stack	Unchanged

Related Instructions

ELDB, LLDB Load a byte from memory into an accumulator.

Exceptions

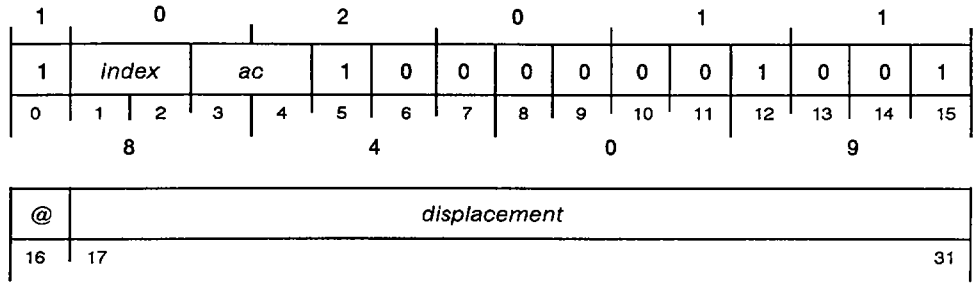
None

Example

```
    XLDB   2,(BYTE_PAIR*2)+1 ;Load AC2 with the low order byte
    .                        ;from the word.
    .
BYTE_PAIR:
    .WORD  0                 ;Location containing a pair of bytes.
```

Load Effective Address (Extended Displacement) XLEF

XLEF *ac*,[@]*displacement*[,*index*]



Function: E --> *ac*

Parameters: None

XLEF calculates the effective address and loads it into the specified accumulator. Bit 0 of the result is guaranteed to be 0.

Arguments

ac After execution, contains result.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Set to 0
Stack	Unchanged

Related Instructions

LEF, ELEF, LLEF
Load an effective address into an accumulator.

Exceptions

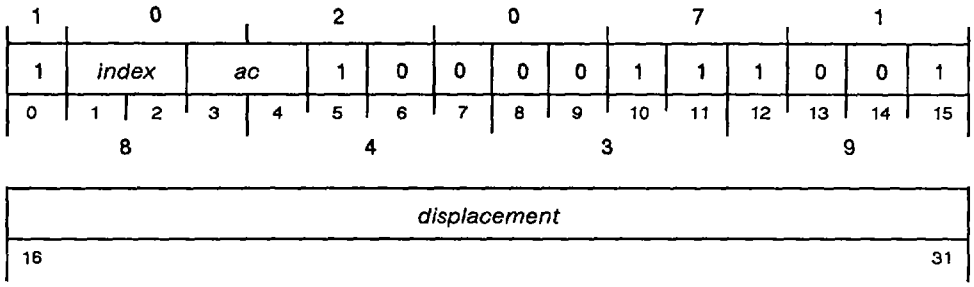
None

Example

```
    XLEF   2,WORD_ARRAY ;Get starting address of array of words.
           WADD         1,2          ;Add the word index from AC1.
           XNLDA        0,0,2        ;Get the word into AC0.
           .
           .
WORD_ARRAY:
           .BLK         16.          ;Array of 16 words.
```

Load Effective Byte Address (Extended Displacement) XLEFB

XLEFB *ac,displacement*[*,index*]



Function: E[byte] --> *ac*

Parameters: None

XLEFB calculates the effective byte address and loads it into the specified accumulator.

Arguments

ac After execution, contains result.

displacement[*,index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

LLEFB Load Effective Byte Address (Extended Displacement)

Exceptions

None

Example

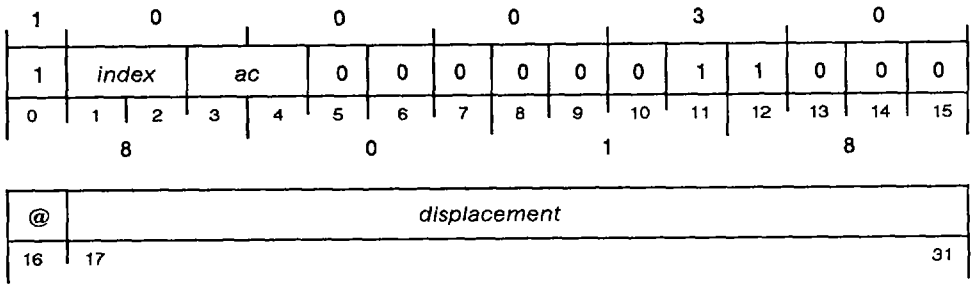
```
XLEFB 2,DEST*2     ;Get the destination byte address.
XLEFB 3,SOURCE*2   ;Get the source byte address.
NLDAI 32.,0        ;Set up to move 32 bytes to destination.
WMOV  0,1          ;Also 32 bytes from the source.
WCMV               ;Move them all.

DEST:     .BLK 16. ;32 bytes.
SOURCE:   .BLK 16. ;32 bytes.
```

Narrow Add Memory Word to Accumulator XNADD

(Extended Displacement)

XNADD *ac*,[@]*displacement*[,*index*]



Function: (E) + *ac* -> *ac*
ALU CRY -> CRY

Parameters: None

XNADD calculates the effective address and adds the signed 16-bit integer contained in this location to the signed 16-bit integer contained in the specified accumulator. Then it sign extends the 16-bit result to 32 bits and loads it into the specified accumulator.

Arguments

ac(16-31) Before execution, contains signed 16-bit integer.

After execution, contains result, sign-extended to 32 bits.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Set with value of ALU CARRY
Overflow	1 if an ALU overflow
PC	PC + 2
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

LNADD, LWADD, XWADD
Add memory contents to accumulator.

Exceptions

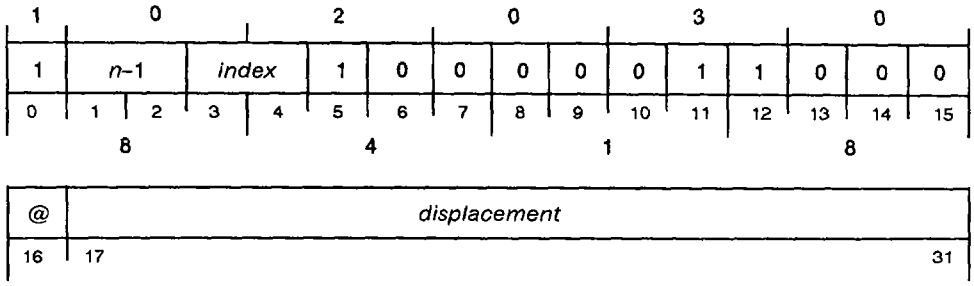
If the result of the add produces a value greater than 32,767, PSR(OVR) is set to 1.

Example

```
XNLDA 0,FIRST     ;Get one value (only 16 bits).
XNADD 0,SECOND    ;Add the second value (16-bit arithmetic).
XNSTA 0,RESULT    ;Store the single word result.
```

Narrow Add Immediate (Extended Displacement) XNADI

XNADI *n*,[@]*displacement*[,*index*]



Function: *n* + (E) -> (E)
ALU CRY -> CRY

Parameters: None

XNADI adds an integer in the range of 1 to 4 to the signed 16-bit integer at the specified memory location.

Arguments

n Integer in range 1 to 4.

Assembler takes coded value of *n* and subtracts one from it before placing it in immediate field. Thus, programmer should code exact value to be added.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Unused
CARRY	Set with value of ALU CARRY (16-bit operation)
Overflow	1 if ALU overflow (16-bit operation)
PC	PC + 2
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

XWADI, LNADI, LWADI
Add 2-bit immediate value to memory.

Exceptions

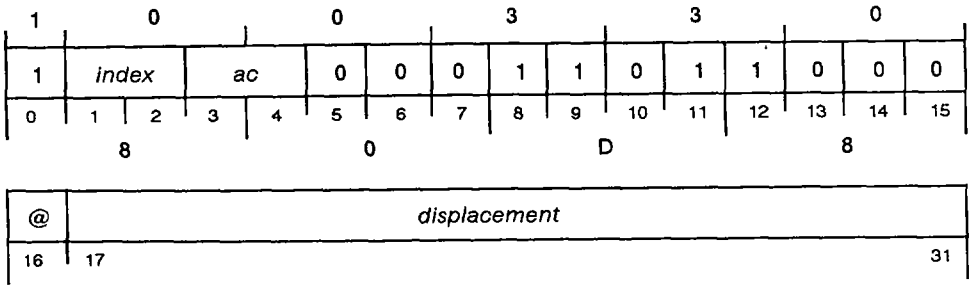
If the add produces a result greater than 32,767, PSR(OVR) is set to 1.

Example

```
    XNADI  4,COUNTER     ;Increment by 4 a counter in memory.
     . . .
COUNTER:  .WORD 0        ;16-bit counter.
```

Narrow Divide Memory Word (Extended Displacement) XNDIV

XNDIV *ac*,[@]*displacement*[,*index*]



Function: *ac* / (E) -> *ac*

Parameters: None

NOTE: If (E) = 0 or result overflows; *overflow* = 1

XNDIV sign extends the signed 16-bit integer contained in the specified accumulator to 32 bits and divides it by the signed 16-bit integer contained in the memory location. It then sign extends the result to 32 bits and loads it into the specified accumulator.

Arguments

ac(16-31)
Before execution, contains signed 16-bit integer (processor sign-extends to 32 bits).
After execution, contains result sign-extended to 32 bits.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	1 if quotient outside specified range or memory word zero; otherwise 0.
PC	PC + 2
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

XWDIV, LNDIV, LWDIV
Divide an accumulator by the contents of memory.

Exceptions

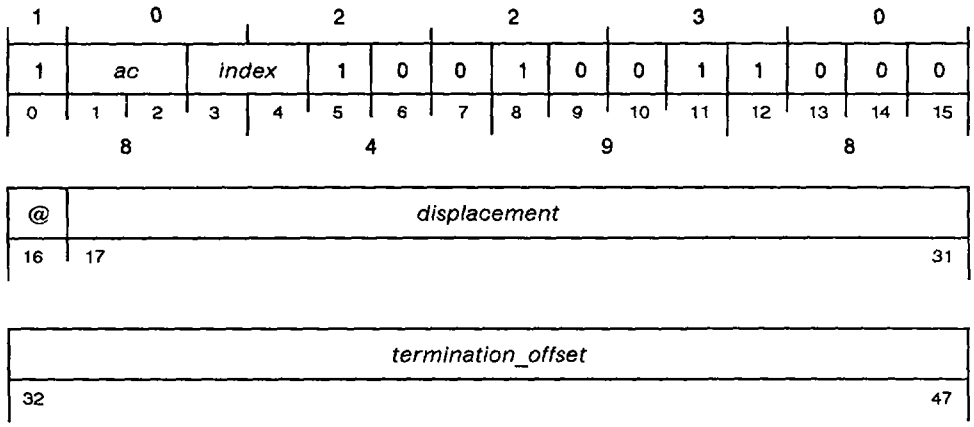
If the quotient is outside the range -32,768 to +32,767 inclusive, or if the memory location contains zero, an overflow occurs, and OVR is set to 1.

Example

```
XNLDA 0,DIVIDEND   ;Get the dividend (16 bits wide).
XNDIV 0,DIVISOR    ;Divide by the divisor.
XNSTA 0,RESULT     ;Store the single word result.
```

Narrow Do Until Greater than (Extended Displacement) XNDO

XNDO *ac,termination_offset,*[@]*displacement*[*,index*]
. ;begin DO-loop
. ;
. ;
WBR ;return to beginning of DO-loop
(normal return)



Function: (E) + 1 -> (E)
If (E) > *ac* then PC + 1 + *termination_offset* -> PC
ALU CRY -> CRY
(E) -> *ac*

Parameters: (E) = 2# -> 2# + 1

XNDO directs the processor to repeat a sequence of instructions until an incremented variable is greater than a loop count.

For each pass the through DO-loop, the processor increments the variable in memory and compares it to the loop count in *ac*. If the contents of memory location is greater than the loop count, the loop ends, the processor moves the incremented value to *ac*, and adds the *termination_offset* plus one to the program counter. If the memory location is equal to or less than the loop count, the processor moves the incremented value to *ac* and continues the DO-loop.

Instructions within the DO-loop (between **XNDO** and **WBR**) can use the loop count in *ac* for indexed addressing. With accumulator-relative indexed addressing, instructions must use absolute displacements.

Arguments

ac Before execution, contains containing signed 32-bit integer specifying loop count. Processor increments value in memory and moves it to *ac*. Value can then be used for ac-relative addressing in DO-loop.

Although value in *ac* can be constant, DO-loop sequence can modify value in memory before restoring it to *ac*. Thus DO-loop sequence can test for condition and then prematurely terminate by modifying either variable or loop-count in memory.

Ac must be reloaded with loop count before processor returns to **XNDO**.

[*termination_offset*]
Specifies PC-relative address for normal return. Argument ranges from 0 to 64 Kwords.

[@]*displacement*[,*index*]
Specifies effective address of a double word in memory to be incremented during each pass of DO-loop. Word contains signed 16-bit integer which is sign-extended to 32-bits.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Set to value of CARRY after each DO-loop increment.
Overflow	1, if fixed-point overflow fault occurs while incrementing DO-loop variable.
PC	PC + 3 (Begin DO-loop) PC + 1 + termination_offset (Normal return)
PSR	OVR set to 1 if incrementing *ac* causes overflow.
Stack	Unchanged

Related Instructions

LWLDA, XWLDA, WLDAI
Use a load accumulator instruction to load *ac* with one more than the actual loop count.

WBR
Use the Wide Banch instruction to end the DO-loop (to loop back to the **XNDO** instruction).

Exceptions

If a fixed-point overflow fault occurs while incrementing DO-loop variable, contents of specified memory location and PC word in narrow return block undefined. (AC0 in fixed-point fault handler properly references address of DO-loop instruction.)

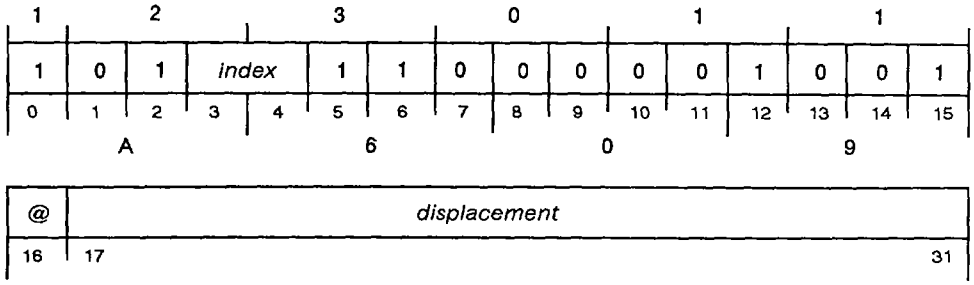
Example

```
    WSUB   0,0 ;Get a 0.
    XNSTA  0,INDEX      ;Initialize the counter in memory.
LOOP:     NLDAI 5,0     ;Maximum index value.
          XNDO 0,END,INDEX          ;Start of the DO-loop.
          .
          .             ;New index value is in AC0 and may be
          .             ;used by computations in the loop.
          .
          WBR LOOP
END:      . . .         ;Loop was executed 5 times.
          .
          .
          .
   INDEX: .WORD 0       ;Index value.
```

Narrow Decrement & Skip if Zero XNDSZ

(Extended Displacement)

XNDSZ [@]*displacement*[,*index*]
(result ≠ 0 return)
(result = 0 return)



Function: (E) - 1 -> (E)
If resulting (E) = 0 then skip

Parameters: None

XNDSZ decrements the unsigned 16-bit integer in a specified memory location by 1, writes the result back into the location, and skips the next sequential instruction if the result is 0. This instruction is indivisible.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1 (result ≠0) PC + 2 (result is 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

DSZ, EDSZ, XWDSZ, LNDSZ, LWDSZ
Decrement contents of memory and skip if result is zero.

Exceptions

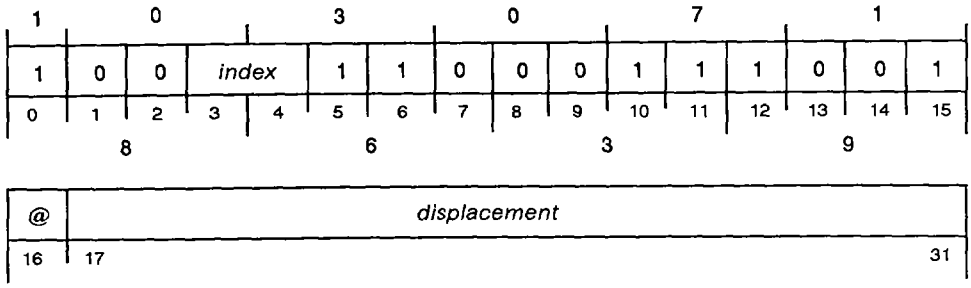
None

Example

```
    NLDAI  5,0          ;Get a constant 5.
    XNSTA  0,COUNTER    ;Initialize the loop counter.
LOOP:     . . .         ;Beginning of loop....
    XNDSZ  COUNTER      ;Decrement counter and skip if zero.
    WBR    LOOP         ;We're not done yet.
    . . .               ;We did the loop 5 times.
    ...
COUNTER:  .WORD 0       ;Counter variable.
```

Narrow Increment & Skip if Zero (Extended Displacement) XNISZ

XNISZ [@]*displacement*[,*index*]
(result ≠ 0 return)
(result = 0 return)



Function: (E) + 1 -> (E)
If resulting (E) = 0 then skip

Parameters: None

XNISZ increments the unsigned 16-bit integer in the memory location by 1, writes the result back into the location, and skips the next sequential instruction if the result is zero. This instruction is indivisible.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 1 (result ≠0) PC + 2 (result = 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

ISZ, EISZ, XWISZ, LNISZ, LWISZ
Increment contents of memory and skip if result is zero.

Exceptions

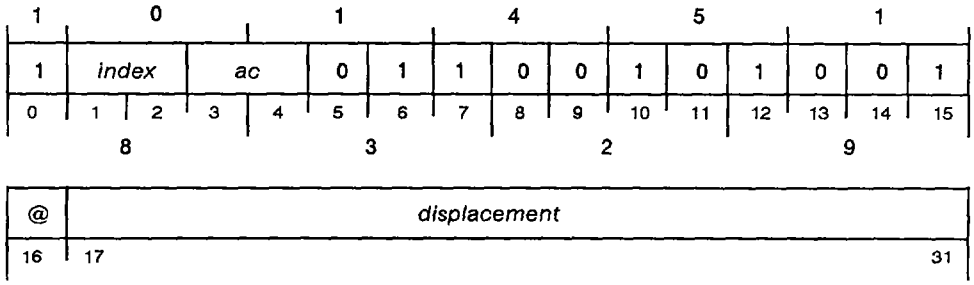
None

Example

```
    NLDAI  -5,0          ;Get a constant -5.
    XNSTA  0,COUNTER     ;Initialize the loop counter.
LOOP:     . . .                    ;Beginning of loop....
          XNISZ COUNTER            ;Increment counter and skip if zero.
          WBR LOOP                 ;We're not done yet.
          . . .                    ;We did the loop 5 times....
COUNTER:  .WORD 0                  ;Counter variable.
```

Narrow Load Accumulator (Extended Displacement) XNLDA

XNLDA *ac*,[@]*displacement*[,*index*]



Function: (E) -> *ac*

Parameters: None

XNLDA calculates the effective address and fetches the signed 16-bit integer contained in this location. Then it sign extends this integer to 32 bits and loads it into the specified accumulator.

Arguments

ac After execution, contains result.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

LDA, ELDA, XWLDA, LNLDA, LWLDA
Load the contents of memory into an accumulator.

Exceptions

None

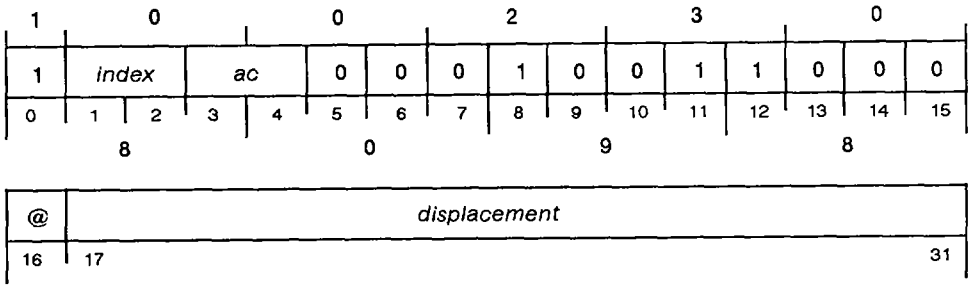
Example

```
XNLDA 0,SINGLE_WORD          ;Get 16 bit value and sign extend.
XWSTA 0,DOUBLE_WORD          ;Store the value as a double word.
```

Narrow Multiply Memory Word XNMUL

(Extended Displacement)

XNMUL *ac*,[@]*displacement*[,*index*]



Function: (E) * *ac* -> *ac*

Parameters: None

XNMUL multiplies the signed 16-bit integer contained in the location specified by E by the signed 16-bit integer contained in the specified accumulator. It then sign extends the result to 32 bits and places the result in the specified accumulator.

Arguments

ac(16-31) Before execution, contains signed 16-bit integer.

After execution, contains result sign-extended to 32 bits.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	1 if result outside specified range; otherwise 0.
PC	PC + 2
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

XWMUL, LNMUL, LWMUL
Multiply an accumulator by the contents of memory.

Exceptions

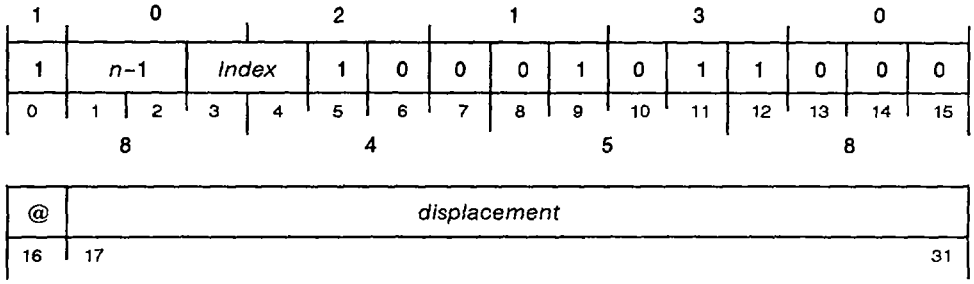
If the result is outside the range of -32,768 to +32,767 inclusive, an overflow occurs, and PSR(OVR) is set to 1.

Example

```
XNLDA 0,FIRST      ;Get one value (only 16 bits).
XNMUL 0,SECOND     ;Multiply by the second value (16-bit arith.).
XNSTA 0,RESULT     ;Store the single word result.
```

Narrow Subtract Immediate (Extended Displacement) XNSBI

XNSBI *n*,[@]*displacement*[,*index*]



Function: (E) - *n* -> (E)
ALU CRY -> CRY

Parameters: None

XNSBI subtracts an integer in the range of 1 to 4 from the signed 16-bit integer contained in the specified memory location.

Arguments

n Integer in range 1 to 4.

Assembler takes coded value of *n* and subtracts one from it before placing it in immediate field. Thus, programmer should code exact value to be subtracted.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Unused
CARRY	Set with value of ALU CARRY (16-bit operation)
Overflow	1 if ALU overflow (16-bit operation)
PC	PC + 2
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

XWSBI, LNSBI, LWSBI
Subtract a 2-bit immediate value from the contents of a memory location.

Exceptions

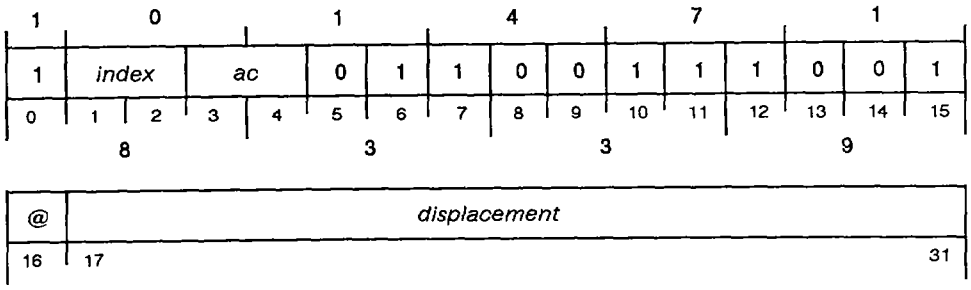
None

Example

```
    XNSBI 2,COUNTER     ;Decrement by 2 a counter in memory.
     . . .
COUNTER: .WORD 0        ;16-bit counter.
```

Narrow Store Accumulator (Extended Displacement) XNSTA

XNSTA *ac*,[@]*displacement*[,*index*]



Function: *ac* -> (E)

Parameters: None

XNSTA calculates the effective address and stores a copy of the 16-bit contents of the specified accumulator into this location.

Arguments

ac(16-31) Before execution, contains 16-bit data.

After execution, contents unchanged.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

STA, ESTA, XWSTA, LNSTA, LWSTA
Store the contents of an accumulator to memory.

Exceptions

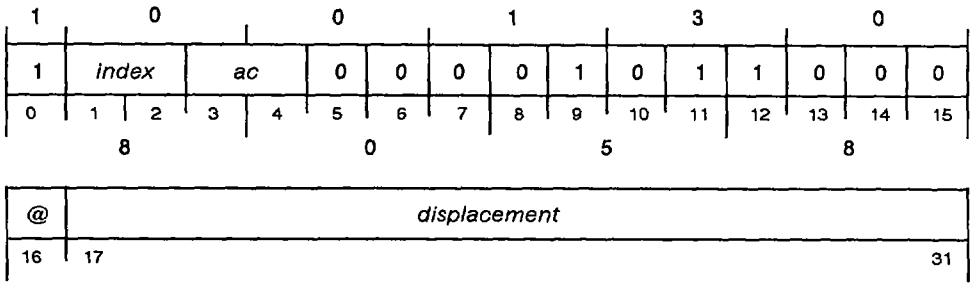
None

Example

```
XNLDA 0,FIRST     ;Get one value (only 16 bits).
XNADD 0,SECOND    ;Add the second value (16-bit arithmetic).
XNSTA 0,RESULT    ;Store the single word result.
```

Narrow Subtract Memory Word (Extended Displacement) XNSUB

XNSUB *ac*,[@]*displacement*[,*index*]



Function: *ac* - (E) -> *ac*
ALU CRY -> CRY

Parameters: None

XNSUB subtracts the signed 16-bit integer contained in the memory location from the signed 16-bit integer contained in the specified accumulator. Then it sign extends the result to 32 bits and stores it in the specified accumulator.

Arguments

ac(16-31) Before execution, contains signed 16-bit integer.

After execution, contains result sign-extended to 32 bits.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Set with value of ALU CARRY
Overflow	1 if ALU overflow
PC	PC + 2
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

XWSUB, LNSUB, LWSUB
Subtract the contents of memory from an accumulator.

Exceptions

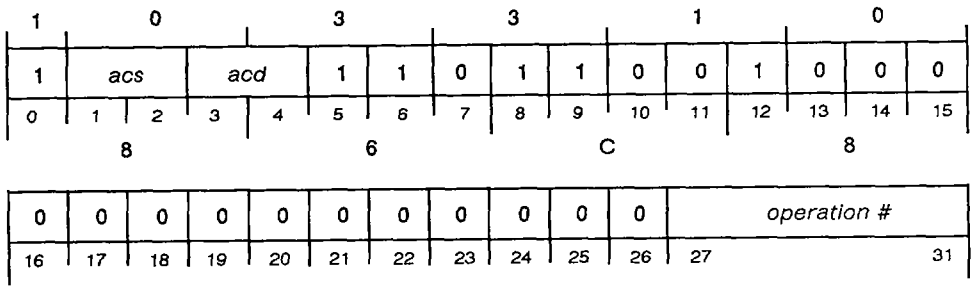
None

Example

```
XNLDA 0,FIRST      ;Get one value (only 16 bits).
XNSUB 0,SECOND     ;Subtract the second value (16-bit arithmetic).
XNSTA 0,RESULT     ;Store the single word result.
```

Extended Operation — XOP0

XOP0 *acs,acd,operation #*



Function: 5 words -> stack (narrow return block)
(E) -> PC
address of *acs* in stack -> AC2
address of *acd* in stack -> AC3

Parameters: (44)page zero = (table) -> unch
E = *operation #* + (44)page zero -> unch

NOTE: Pushed PC = **XOP0** + 2

XOP0 pushes a return block onto the narrow stack and transfers control to a procedure pointed to by the selected address in an extended operations table (XOP table). It is an efficient way to transfer control from one procedure to another.

The return block has the following contents:

Word Pushed	Contents
1	AC0
2	AC1
3	AC2
4	AC3
5	CARRY in bit 0; (XOP0 address + 2) in bits 1-15

After the return block is pushed, the stack address of the stored accumulator designated as *acs* is loaded into AC2, and the stack address of the stored accumulator designated as *acd* is loaded into AC3.

XOP0 then uses the specified operation number as an offset to the starting address of the XOP table from which it fetches the intermediate address of the extended operations procedure. The resulting effective address is loaded into the PC as the starting address of the new procedure.

The XOP table can contain up to 40_8 procedure entry points (intermediate addresses). Its starting address is stored in location 44_8 of page zero reserved memory.

The contents of the **XOP0** starting address are unaffected. All addresses must be in the current segment.

Arguments

acs	Specifies an accumulator whose pushed stack address will be placed into AC2.
acd	Specifies an accumulator whose pushed stack address will be placed into AC3.
operation #	Contains unsigned integer specifying offset from XOP table starting address.

Registers, Flags, and Stacks

AC0	First word pushed.
AC1	Second word pushed.
AC2	Third word pushed. After execution, contains stack address of pushed contents of *acs*.
AC3	Fourth word pushed. After execution, contains stack address of pushed contents of *acd*.
CARRY	Unchanged
Overflow	0
PC	Effective address derived from address fetched from table.
PSR	Unchanged
Stack	Narrow stack pointer incremented by five words.

Related Instructions

POPB	Use the Pop Block instruction to restore the pushed values and return.
WXOP	Wide Extended Operation

Exceptions

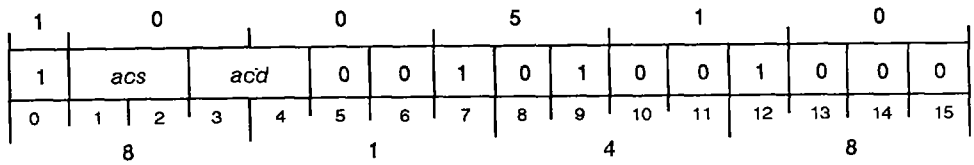
None

Example

```
XOP0
```

Exclusive OR XOR

XOR *acs,acd*



Function: *acs* XOR *acd* -> *acd*

Parameters: None

XOR forms the logical Exclusive OR of *acs* and *acd*, placing the result in *acd*. It sets a bit position in the result to 1 if the corresponding bit positions in the two operands are unlike; otherwise, the instruction sets the result bit to 0.

Arguments

acs(16–31)	Before execution, contains 16–bit value.
	After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd(16–31)	Before execution, contains 16–bit value.
	After execution, contains result.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

WXOR	Wide Exclusive OR
IOR	Inclusive OR
WIOR	Wide Inclusive OR

Exceptions

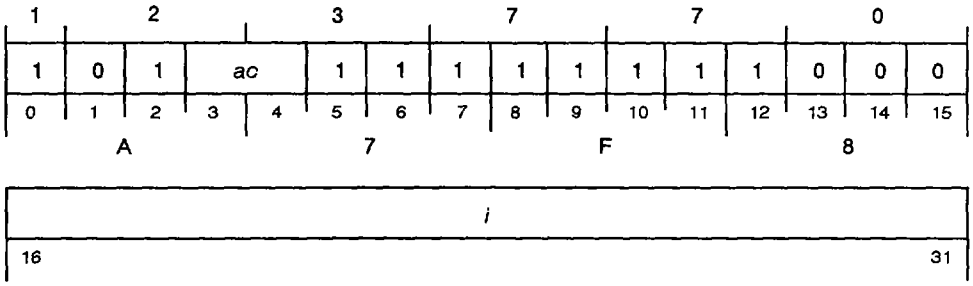
None

Example

```
XOR   0,1          ;Exclusive OR AC0[16-31] and AC1[16-31].
```

Exclusive OR Immediate XORI

XORI *i,ac*



Function: *i* XOR *ac* -> *ac*

Parameters: None

XORI forms logical exclusive OR of the contents of the 16-bit immediate field and the 16-bit value of the specified accumulator, placing the result in the specified accumulator.

Arguments

i	16-bit value
ac(16-31)	Before execution, contains 16-bit value.
	After execution, contains result.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

WXORI	Wide Exclusive OR Immediate

Exceptions

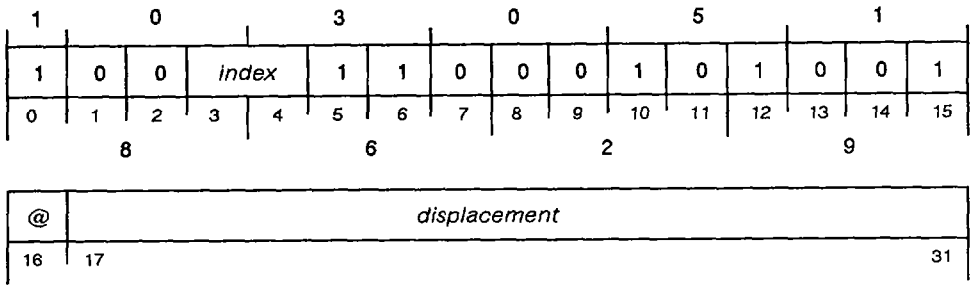
None

Example

```
XORI  377,0        ;Form the 1's complement of the low order byte
                   ;of AC0.
```

Push Address (Extended Displacement) XPEF

XPEF [@]*displacement*[,*index*]



Function: E --> wide stack

Parameters: None

XPEF calculates the effective address and pushes it onto the wide stack. Then it checks for stack overflow.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	After execution, top double word of wide stack contains effective address with bit 0 guaranteed to be 0.

Related Instructions

XPEFB Push Byte Address (Extended Displacement)

Exceptions

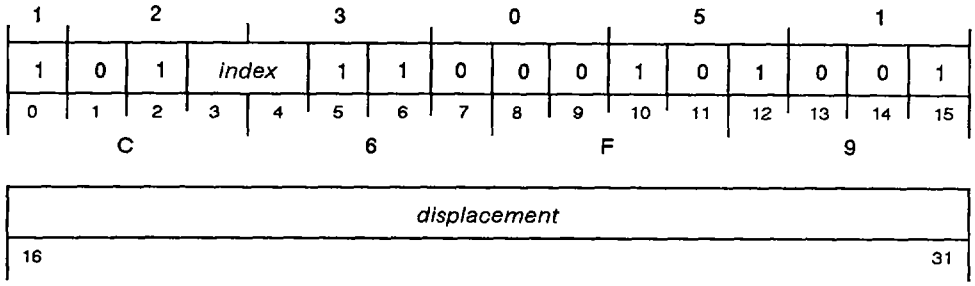
None

Example

```
XPEF   ARG_2          ;Push address of argument 2 onto the stack.
XPEF   ARG_1          ;Push address of argument 1 onto the stack.
XPEF   ARG_0          ;Push address of argument 0 onto the stack.
LCALL  SUBROUT,0,3    ;Call a subroutine with 3 arguments.
```

Push Byte Address (Extended Displacement) XPEFB

XPEFB *displacement*[,*index*]



Function: E(byte) --> wide stack

Parameters: None

XPEFB calculates a byte address and pushes it onto the wide stack. Then it checks for stack overflow.

Arguments

displacement[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	After execution, top double word of wide stack contains 32-bit byte address.

Related Instructions

XPEF	Push Address (Extended Displacement)

Exceptions

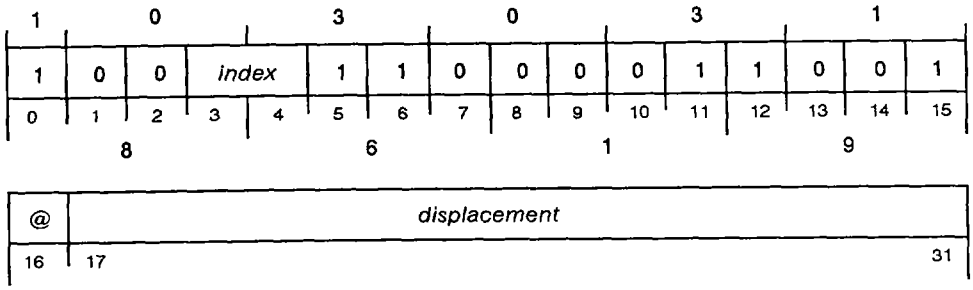
None

Example

```
XPEFB ARG_1*2       ;Push byte address of argument 1 onto the stack.
XPEF  ARG_0         ;Push address of argument 0 onto the stack.
LCALL SUBROUT,0,2   ;Call a subroutine with 2 arguments.
                    ;Subroutine must be expecting a byte
                    ;address to ARG_1 and a word address to ARG_2.
```

Push Jump (Extended Displacement) XPSHJ

XPSHJ [@]*displacement*[,*index*]



Function: PC + 2 -> wide stack
E -> PC

Parameters: None

XPSHJ pushes the current contents of the PC plus 2 onto the wide stack and reloads the program counter with the specified address. Sequential operation continues with the instruction addressed by the updated value of the program counter. Stack overflow is checked after the push operation finishes. The pushed address always references the current segment.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction confined to current segment.

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	Effective address
PSR	Unchanged
Stack	Wide stack pointer incremented by one; wide frame pointer unchanged.

Related Instructions

PSHJ, LPSHJ Push the program counter onto a stack and jump to a subroutine.

Exceptions

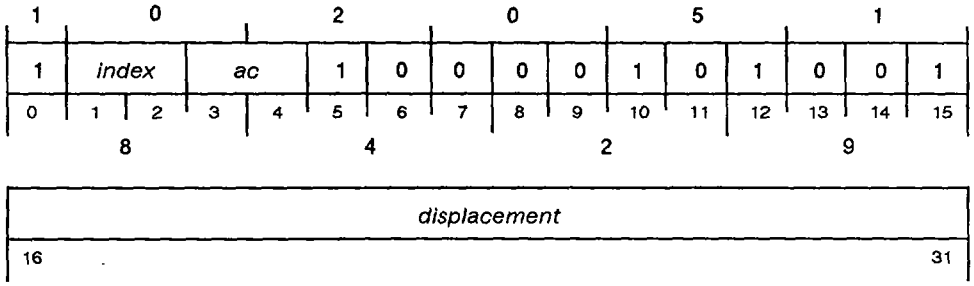
None

Example

```
    XPSHJ  SUBROUT      ;Call a subroutine. Return PC is on the stack.
    ...
SUBROUT:
                        ;Subroutine is implemented here.
    ...
    WPOPJ               ;Pop return address and return to caller.
                        ;ACs modified in the subroutine are not
                        ;restored.
```

Store Byte (Extended Displacement) XSTB

XSTB *ac,displacement*[,*index*]



Function: *ac*[right byte] --> (E)byte

Parameters: None

XSTB calculates the effective byte address. Then it moves a copy of the contents of bits 24–31 of the specified accumulator into memory at the location specified by the byte address.

Arguments

ac(24–31) Before execution, contains 8-bit data.

After execution, contents unchanged.

displacement[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

ESTB, LSTB Store a byte in an accumulator into memory.

Exceptions

None

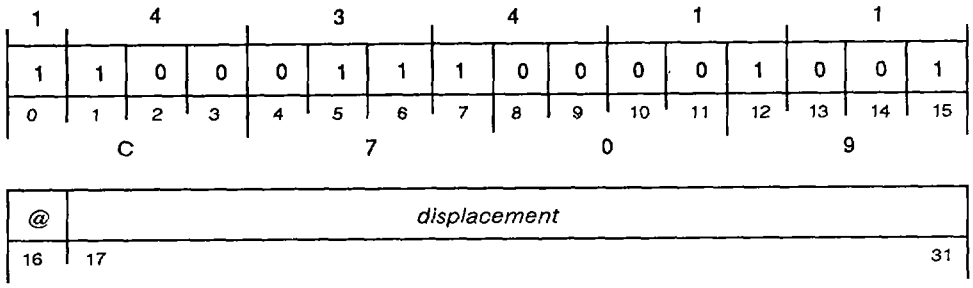
Example

```
    XSTB   2,(BYTE_PAIR*2)+1        ;Store the byte in bits 24-31 of AC2
           .                        ;into the low-order byte of the word
           .                        ;in memory.
           .
BYTE_PAIR:.WORD 0                   ;Location containing a pair of bytes.
```

Vector on Interrupting Device (Extended Displacement) XVCT

Privileged Instruction

XVCT [@]*displacement*



Function: Jumps to interrupt handler using vector table and device control table (DCT) for interrupting device.
Initializes wide stack registers, vector stack and fault handler address.
Old wide stack registers, old mask, old fault handler, wide return block -> new stack
Perform mask out

Parameters:
E = entry 0 (vector table in segment 0)
Interrupting device # = double-word offset to vector table entry
Vector table entry(bits 1-31) = E(DCT entry 0)
Wide stack registers = ? -> Vector stack
Wide stack fault handler addr. = ? -> Vector stack handler addr.
AC0 = ? -> Revised priority mask
AC1 = ? -> I/O channel & device code(23-31[zero extended])
AC2 = ? -> DCT(entry 0 address)
PSR = ? -> DCT(word 4)
PC = ? -> DCT(word 0 & 1 [bits 4-31])

XVCT must be the first instruction that the processor fetches for a type 3 interrupt handler. The processor executes **XVCT** before honoring further interrupts.

The effective address refers to the vector table in segment 0. The interrupting device number becomes a double-word offset that points to a table entry containing the address of the device control table (DCT) for the interrupting device.

The processor saves the current wide stack parameters and intializes a vector stack. The processor then pushes the old stack parameters and a wide return block onto the new stack and initializes the accumulators, PSR, and PC using the contents of the DCT. The processor then transfers control to the word addressed by the PC.

Refer to Interrupt Servicing in the chapter "Device Management" for further information.

Arguments

[@]*displacement* Effective address (E) refers to entry zero of vector table in segment 0. Indirection chain, if any, is narrow.

Interrupting device number becomes double-word offset pointing to appropriate entry in vector table.

Vector table entry bits 1-31 contains address of entry zero of DCT for interrupting device.

Registers, Flags, and Stacks

AC0	Initialized by processor to contain revised priority mask to perform maskout.
AC1(23–31)	Initialized by processor to contain I/O channel and device code; zero extended.
AC2	Initialized by processor to contain entry zero, address of DCT.
AC3	Unused
PSR	Initialized by processor to contain word four of DCT.
Overflow	Unaffected
PC	Initialized by processor to contain address of device interrupt routine from first double word, bits 4–31, of DCT.
Stack	New wide (vector) stack contains: old wide stack registers old wide stack fault handler address standard wide return block old mask Wide stack registers and wide stack fault handler initialized for new vector stack.

Related Instructions

WRSTR	Wide Restore should be the last instruction in the vectored interrupt handler. **WRSTR** pops the wide return block from the vector stack, returning control from a base-level interrupt.

Exceptions

None

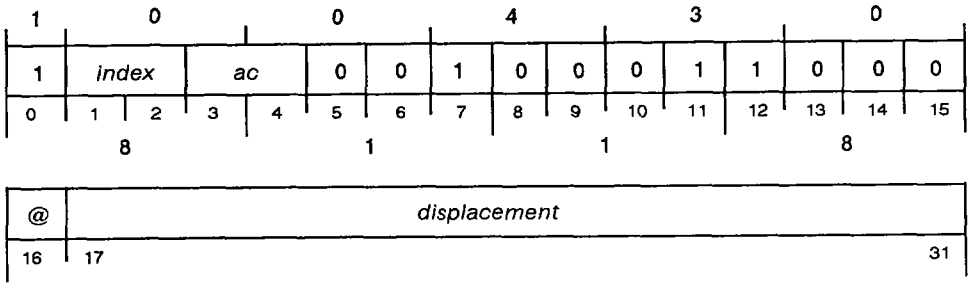
Example

```
XVCT
```

Wide Add Memory Word to Accumulator XWADD

(Extended Displacement)

XWADD *ac*,[@]*displacement*[,*index*]



Function: (E) + *ac* -> *ac*
ALU CRY -> CRY

Parameters: None

XWADD calculates the effective address and adds the signed 32-bit integer contained in this location to the signed 32-bit integer contained in the specified accumulator, placing the result in the accumulator.

Arguments

ac Before execution, contains signed 32-bit integer.

After execution, contains result.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Set with value of ALU CARRY
Overflow	1 if an ALU overflow
PC	PC + 2
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

LNADD, LWADD, XNADD
Add memory contents to an accumulator.

Exceptions

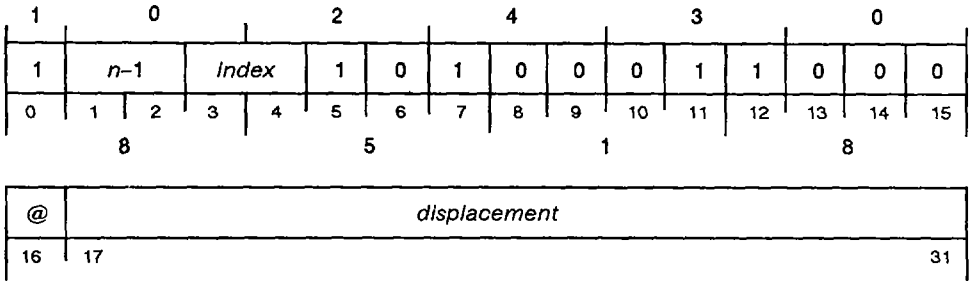
If the result of the add produces a result outside the range, -2,147,483,648 to +2,147,483,647, PSR(OVR) is set to 1.

Example

```
XWLDA 0,FIRST     ;Get one value (32 bits).
XWADD 0,SECOND    ;Add the second value (32-bit arithmetic).
XWSTA 0,RESULT    ;Store the double word result.
```

Wide Add Immediate (Extended Displacement) XWADI

XWADI *n*,[@]*displacement*[,*index*]



Function: *n* + (E) -> (E)
ALU CRY -> CRY

Parameters: None

XWADI adds an integer in the range of 1 to 4 to the signed 32-bit integer contained in a memory location.

Arguments

n Integer in range 1 to 4.

Assembler takes coded value of *n* and subtracts one from it before placing it in immediate field. Thus, programmer should code exact value to be added.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Set with value of ALU CARRY
Overflow	1 if ALU overflow
PC	PC + 2
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

LNADI, LWADI, XNADI
Add 2-bit immediate value to memory.

Exceptions

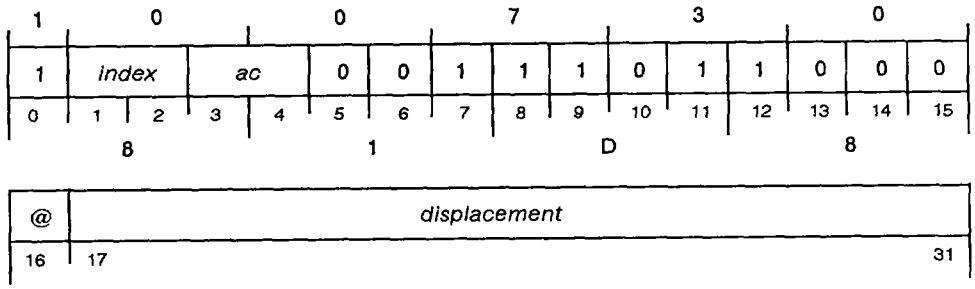
None

Example

```
    XWADI  4,COUNTER    ;Increment by 4 a counter in memory.
     . . .
COUNTER:
          .DWORD       0            ;32-bit counter.
```

Wide Divide Memory Word (Extended Displacement) XWDIV

XWDIV *ac*,[@]*displacement*[,*index*]



Function: *ac* / (E) -> *ac*

Parameters: None

NOTE: If (E) = 0 or result overflows; overflow = 1; *ac* = unchanged

XWDIV sign extends the signed 32-bit integer contained in the specified accumulator to 64 bits and divides it by the signed 32-bit integer contained in the memory location, placing the quotient into the specifed accumulator.

Arguments

ac Before execution, contains signed 32-bit integer.
After execution, contains result.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	1 if quotient outside specified range or memory word 0; otherwise 0.
PC	PC + 2
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

XNDIV, LNDIV, LWDIV
Divide an accumulator by the contents of memory.

Exceptions

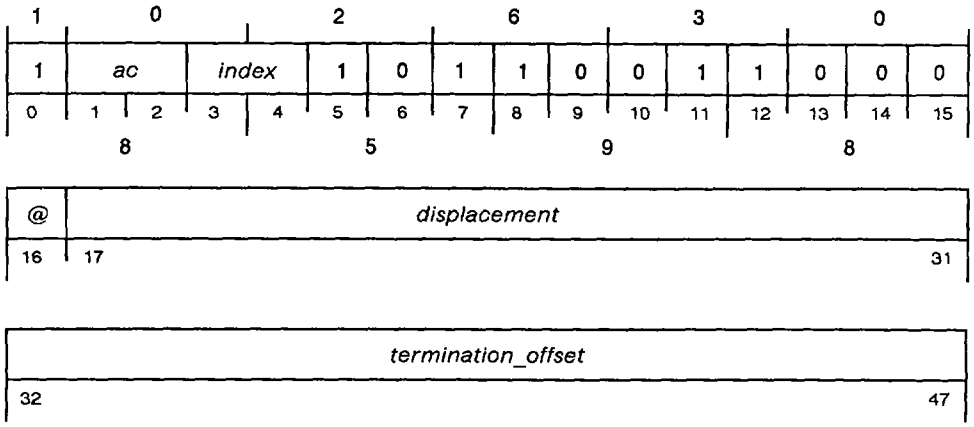
If the quotient is outside the range -2,147,483,648 to +2,147,483,647 inclusive, or if the memory word contains zero, an overflow occurs, PSR(OVR) is set to 1, and *ac* is unchanged.

Example

```
XWLDA 0,DIVIDEND   ;Get the dividend (32 bits wide).
XWDIV 0,DIVISOR    ;Divide by the divisor.
XWSTA 0,RESULT     ;Store the double word result.
```

Wide Do Until Greater Than (Extended Displacement) XWDO

```
XWDO  ac,termination_offset,[@]displacement[,index]
 .          ;begin DO-loop
 .          ;
 .          ;
WBR         ;return to beginning of DO-loop
normal return
```



Function: (E) + 1 -> (E)
If (E) > *ac* then PC + 1 + *termination_offset* -> PC
ALU CRY -> CRY
(E) -> *ac*

Parameters: (E) = 2# -> 2# + 1

XWDO directs the processor to repeat a sequence of instructions until an incremented variable is greater than a loop count.

For each pass through the DO-loop, the processor increments the variable in memory and compares it to the loop count in *ac*. If the content of the memory location is greater than the loop count in *ac*, the loop ends, the processor moves the incremented value to *ac*, and adds the *termination_offset* plus one to the program counter.

If the memory location is less than or equal to the loop count in *ac*, the processor moves the incremented value to the *ac* and continues the DO-loop. The instructions within the DO-loop (i.e., between **XWDO** and **WBR**) can use the loop count in *ac* for index addressing.

When using accumulator relative-indexed addressing, the instructions within the DO-loop must use absolute displacements.

Arguments

ac Before execution, contains signed 32-bit integer specifying loop count. After incrementing value in memory, processor moves it to *ac*, which you can then use for *ac*-relative addressing in DO-loop.

Although you can consider value in *ac* contents as constant, DO-loop sequence can modify value in memory before restoring it to *ac*. For instance, DO-loop sequence can test for condition and then prematurely terminate DO-loop by modifying either variable or loop-count in memory.

NOTE: *You must reload ac with loop count before processor returns to* **XWDO** *instruction.*

[*termination_offset*]
Specifies PC-relative address for normal return. Argument ranges between 0- to 64- Kwords.

[@]*displacement*[,*index*]
Effective address specifies double word in memory that processor increments for each pass of DO-loop. Double word contains signed 32-bit integer.

Registers, Flags, and Stacks

AC0-AC3	Can be specified as *ac*; otherwise unused.
CARRY	Set to value of CARRY after each DO-loop increment.
Overflow	1 if incrementing *ac* causes overflow.
PC	PC + 3 (begin DO-loop) PC + 1 + *termination_offset* (normal return)
PSR	OVR set to 1 if incrementing *ac* causes overflow.
Stack	Unchanged

Related Instructions

LWLDA, XWLDA, WLDAI
Use these instructions to load *ac* with one more than the actual loop count.

WBR
Use the Wide Branch instruction to end the DO-loop (loop back to the **XWDO** instruction).

Exceptions

The contents of the specified memory location and the program counter value in any return block are both undefined.

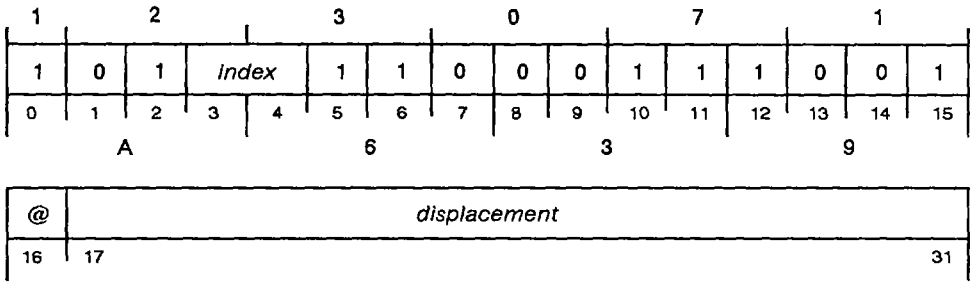
If a fixed-point overflow fault occurs while incrementing the DO-loop variable, contents of memory location and PC in return block are undefined. (AC0 in fixed-point fault handler properly references address of DO-loop instruction.)

Example

```
    WSUB   0,0            ;Get a 0.
    XWSTA  0,INDEX        ;Initialize the counter in memory.
LOOP:       NLDAI 5,0     ;Maximum index value.
            XWDO 0,END,INDEX          ;Start of the DO-loop.
            .
            .             ;New index value is in AC0 and may be
            .             ;used by computations in the loop.
            .
            WBR LOOP
END:        . . .         ;Loop was executed 5 times.
            .
            .
            .
INDEX:      .DWORD 0      ;Index value.
```

Wide Decrement & Skip if Zero (Extended Displacement) XWDSZ

XWDSZ [@]*displacement*[,*index*]
(result ≠ 0 return)
(result = 0 return)



Function: (E) - 1 -> (E)
If resulting (E) = 0 then skip

Parameters: None

XWDSZ calculates the effective address and decrements by one the unsigned 32-bit integer in this location. If the result is equal to zero, **XWDSZ** skips the next sequential word.

XWDSZ executes in one indivisible memory cycle if the word to be decremented is located on a double-word boundary.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0–AC3	Unused
CARRY	Unchanged
Overflow	0
PC	PC + 2 (result ≠0) PC + 3 (result = 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

DSZ, EDSZ, XNDSZ, LNDSZ, LWDSZ
Decrement the contents of memory and skip if result equals zero.

Exceptions

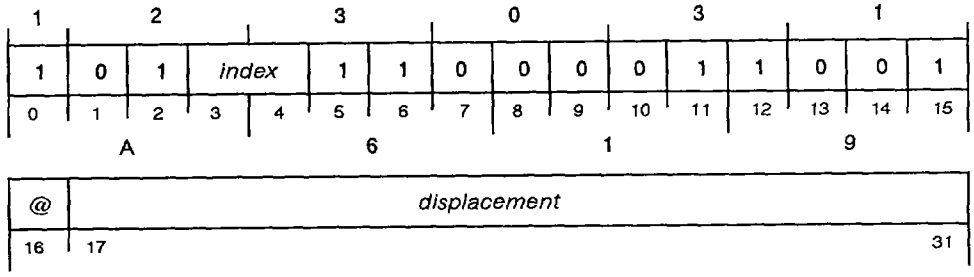
None

Example

```
    NLDAI  5,0          ;Get a constant 5.
    XWSTA  0,COUNTER    ;Initialize the loop counter.
LOOP:      . . .        ;Beginning of loop....
    XWDSZ  COUNTER      ;Decrement counter and skip if zero.
           WBR LOOP     ;We're not done yet.
           . . .        ;We did the loop 5 times....
COUNTER:   .DWORD 0     ;Counter variable.
```

Wide Increment and Skip if Zero (Extended Displacement) XWISZ

XWISZ [@]*displacement*[,*index*]
(result ≠ 0 return)
(result = 0 return)



Function: (E) + 1 -> (E)
If resulting (E) = 0 then skip

Parameters: None

XWISZ calculates the effective address and increments by one the unsigned 32-bit integer in this location. If the result is equal to zero, **XWISZ** skips the next sequential word.

XWISZ executes in one indivisible memory cycle if the word to be incremented is located on a double-word boundary.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Unused	
CARRY	Unchanged	
Overflow	0	
PC	PC + 2	(result ≠ 0)
	PC + 3	(result = 0)
PSR	Unchanged	
Stack	Unchanged	

Related Instructions

ISZ, EISZ, XNISZ, LNISZ, LWISZ
Increment the contents of memory and skip if result is zero.

Exceptions

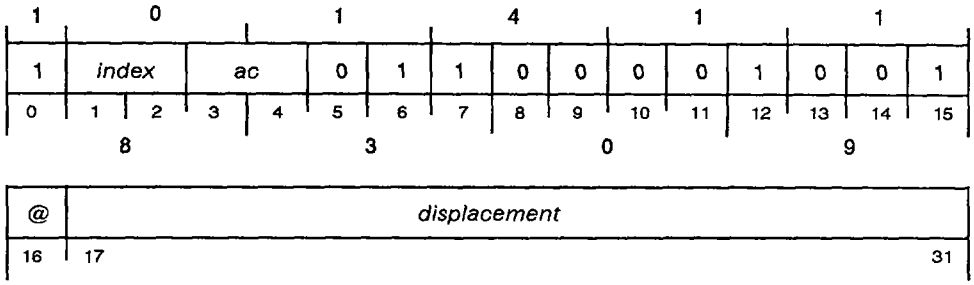
None

Example

```
    NLDAI  -5,0          ;Get a constant -5.
    XWSTA  0,COUNTER     ;Initialize the loop counter.
LOOP:      . . .         ;Beginning of loop.
    . . .
    XWISZ  COUNTER       ;Increment counter and skip if zero.
    WBR    LOOP          ;We're not done yet.
    . . .                ;We did the loop 5 times.
COUNTER:  .DWORD 0       ;Counter variable.
```

Wide Load Accumulator (Extended Displacement) XWLDA

XWLDA *ac*,[@]*displacement*[,*index*]



Function: (E) -> *ac*

Parameters: None

XWLDA calculates the effective address and fetches the 32-bit value contained in this location. Then it loads a copy of this value into the specified accumulator.

Arguments

ac After execution, contains result.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0–AC3 Can be specified as *ac*; otherwise unused.

CARRY Unchanged

Overflow 0

PC PC + 2

PSR Unchanged

Stack Unchanged

Related Instructions

LDA, ELDA, XNLDA, XWLDA, LNLDA
Load an accumulator with the contents of memory.

Exceptions

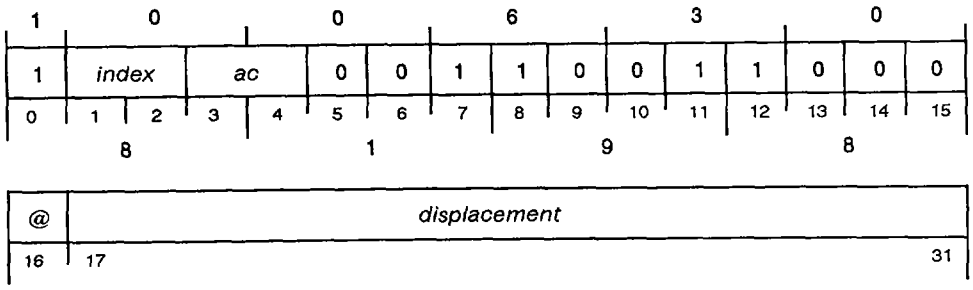
None

Example

```
XWLDA 0,DOUBLE_SOURCE          ;Get 32-bit value.
XWSTA 0,DOUBLE_DEST            ;Store 32-bit value.
```

Wide Multiply Memory Word (Extended Displacement) XWMUL

XWMUL *ac*,[@]*displacement*[,*index*]



Function: (E) * *ac* -> *ac*

Parameters: None

XWMUL multiplies the signed 32-bit integer contained in memory by the signed 32-bit integer contained in the specified accumulator. It then loads the least significant 32 bits of the result into the specified accumulator.

Arguments

ac Before execution, contains signed 32-bit integer.

After execution, contains least significant 32 bits of result.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	1 if result is outside specified range; otherwise 0.
PC	PC + 2
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

XNMUL, LNMUL, LWMUL
Multiply an accumulator by the contents of memory.

Exceptions

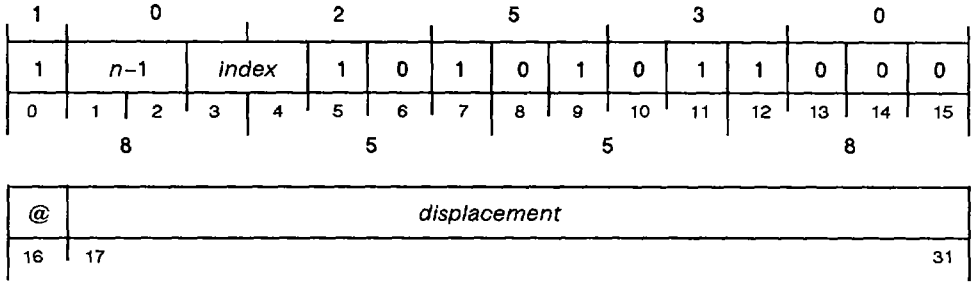
If the result is outside the range -2,147,483,648 to +2,147,483,647 inclusive, an overflow occurs, and PSR(OVK) is set to 1.

Example

```
XWLDA 0,FIRST      ;Get one value (32 bits).
XWMUL 0,SECOND     ;Multiply by the second value (32-bit arith.).
XWSTA 0,RESULT     ;Store the double word result.
```

Wide Subtract Immediate (Extended Displacement) XWSBI

XWSBI *n*,[@]*displacement*[,*index*]



Function: (E) - *n* -> (E)
ALU CRY -> CRY

Parameters: None

XWSBI subtracts an integer in the range of 1 to 4 from the signed 32-bit integer contained in the specified memory location.

Arguments

n Integer in range 1 to 4.

Assembler takes coded value of *n* and subtracts one from it before placing it in immediate field. Thus, programmer should code exact value to be subtracted.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Unused
CARRY	Set with value of ALU CARRY
Overflow	1 if ALU overflow
PC	PC + 2
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

XNSBI, LNSBI, LWSBI
Subtract a 2-bit immediate value from the contents of memory.

Exceptions

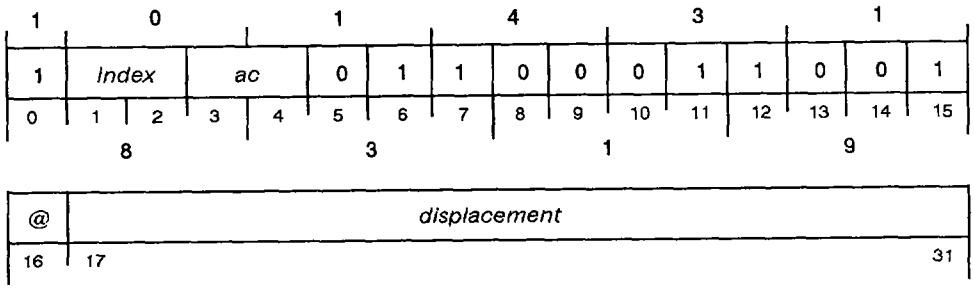
None

Example

```
    XWSBI  2,COUNTER    ;Decrement by 2 a counter in memory.
    . . .
COUNTER:  .DWORD 0      ;32-bit counter.
```

Wide Store Accumulator (Extended Displacement) XWSTA

XWSTA *ac*,[@]*displacement*[,*index*]



Function: *ac* -> (E)

Parameters: None

XWSTA calculates the effective address and stores a copy of the 32-bit contents of the specified accumulator into this location.

Arguments

ac Before execution, contains 32-bit data.

After execution, contents unchanged.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0–AC3	Can be specified as *ac*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 2
PSR	Unchanged
Stack	Unchanged

Related Instructions

STA, ESTA, LNSTA, LWSTA, XNSTA
Store the contents of an accumulator into memory.

Exceptions

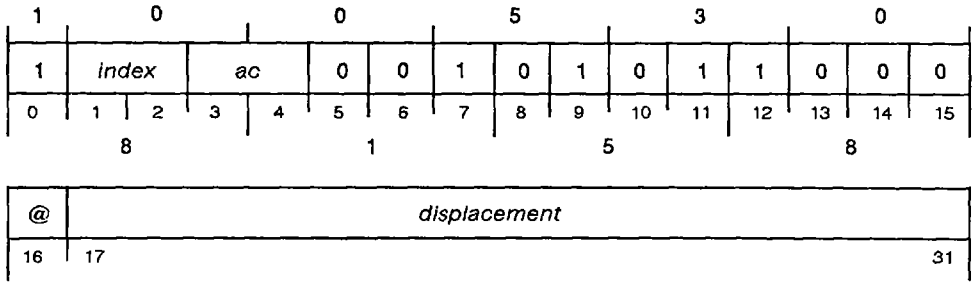
None

Example

```
XWLDA 0,DOUBLE_SOURCE          ;Get 32-bit value.
XWSTA 0,DOUBLE_DEST            ;Store 32-bit value.
```

Wide Subtract Memory Word (Extended Displacement) XWSUB

XWSUB *ac*,[@]*displacement*[,*index*]



Function: *ac* - (E) -> *ac*
ALU CRY -> CRY

Parameters: None

XWSUB subtracts the signed 32-bit integer contained in memory from the signed 32-bit integer contained in the specified accumulator. Then it loads the result into the specified accumulator.

Arguments

ac Before execution, contains signed 32-bit integer.

After execution, contains result.

[@]*displacement*[,*index*]
Effective address generated by instruction can access any word in 4-Gbyte range.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *ac*; otherwise unused.
CARRY	Set with value of ALU CARRY
Overflow	1 if ALU overflow
PC	PC + 2
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

LNSUB, LWSUB, XNSUB
Subtract the contents of memory from an accumulator.

Exceptions

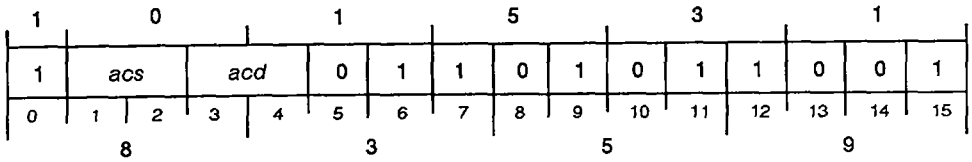
None

Example

```
XWLDA 0,FIRST     ;Get one value (32 bits).
XWSUB 0,SECOND    ;Subtract the second value (32-bit arith.).
XWSTA 0,RESULT    ;Store the double word result.
```

Zero Extend ZEX

ZEX *acs,acd*



Function: *acs*[16 bit #] -> *acd*[32 bit #] (zero extended)

Parameters: None

ZEX zero-extends the 16-bit integer contained in *acs* to 32 bits and loads the result into *acd*.

Arguments

acs(16-31)	Before execution, contains 16-bit integer.
	After execution, contents unchanged unless *acs* and *acd* are same accumulator.
acd	After execution, contains *acs* zero-extended to 32 bits.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
CARRY	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

SEX	Sign Extend

Exceptions

None

Example

```
ADC 3,3    ;Set AC3[16-31] to ones. AC3[0-15] undefined.
ZEX 3,1    ;AC3 unchanged. AC1[0-15] is now all zeros;
           ;AC1[16-31] is now all ones.
```

A

Register Fields

This appendix describes register formats which programmers can access on the ECLIPSE MV/Family computers.

The general information presented in this appendix applies to all 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
CPU Identification	Accumulators with information pertaining to CPU
DCH/BMC Status	Information about data channel and burst multiplexor channel maps
Floating-Point	Information about floating-point computations status
Processor Status	Information about fixed-point computations

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.

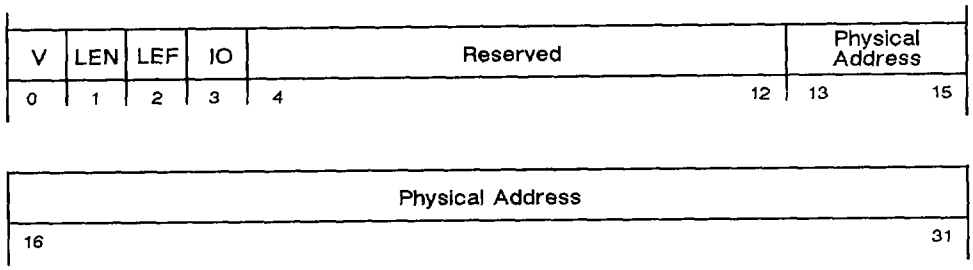


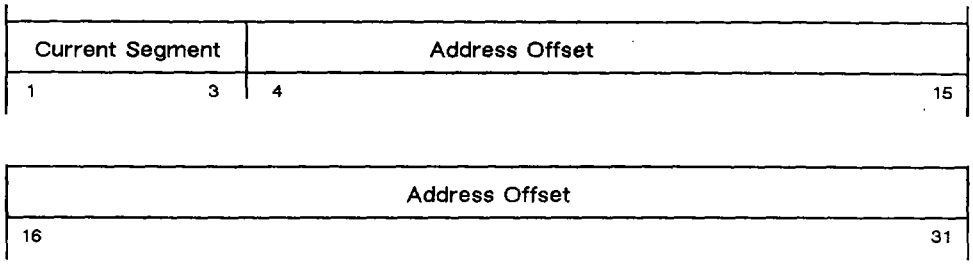
Table A-2 *SBR contents*

Bits	Name	Contents or Function
0	V	Segment validity bit: Indicates ability of processor to reference a segment 0 Invalid SBR 1 Valid SBR
1	LEN	Length bit: Indicates maximum range of logical memory address 0 One-level page table 1 Two-level page table
2	LEF	LEF enable: Indicates whether processor operates in LEF or I/O mode 0 I/O mode 1 LEF mode
3	IO	I/O enable: Indicates if I/O protection violation occurs when I/O instruction is executed 0 Protection violation occurs 1 I/O instruction executes
4-12	Reserved	Reserved for future use
13-31	Physical Address	Identifies physical page address in memory of indicated page table

Program Counter

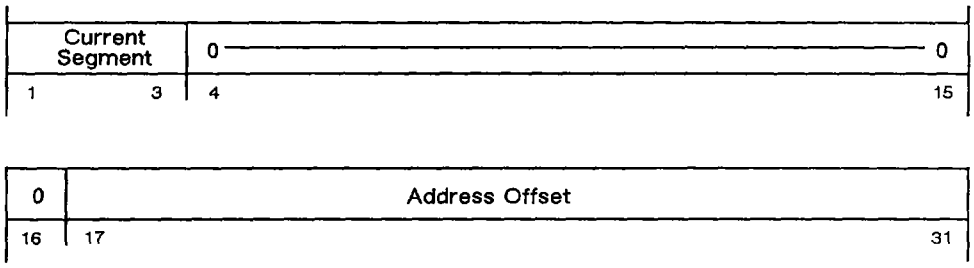
The 31-bit program counter (PC) contains the logical address of the currently executing instruction. PC formats are diagrammed and described below.

PC Format for Execution of ECLIPSE MV Programs



Bits	Name	Contents or Function
1-3	Current Segment	Current segment of program execution
4-31	Address Offset	28-bit address of currently executing instruction

PC Format Altered by ECLIPSE 16-Bit Program Flow Instructions



Bits	Name	Contents or Function
1-3	Current Segment	Current segment of program execution
4-16	0 - 0	Set to 0 by instruction
17-31	Address Offset	15-bit address formed by program flow instruction

Processor Status Register

Only MV-specific instructions affect the 16-bit processor status register (PSR). The format of the PSR is diagrammed below and described in Table A-3.

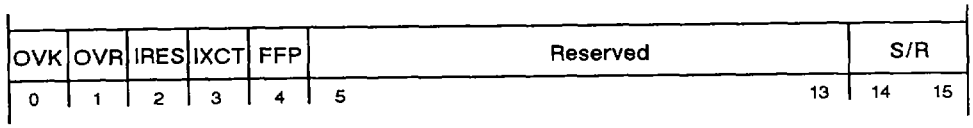


Table A-3 *PSR contents*

Bits	Name	Contents or Function
0	OVK	Overflow mask 0 No fixed-point overflow trap 1 Trap on OVR set to 1
1	OVR	Fixed-point overflow indicator Set to 1 when calculating two's-complement number that does not fit in specified location or register, or when attempting to divide by 0 If OVK equals 1, setting OVR to 1 results in fixed-point overflow fault
2	IRES	Micro-interrupt resume flag Set to 1 when processor receives I/O interrupt request while executing resumable instruction such as WEDIT
3	IXCT	Interrupt execute flag Set to 1 when processor receives I/O interrupt request while executing instruction that was inserted into instruction stream - for example, a PBX instruction
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 the OVK and OVR bits 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 MV system-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-4.

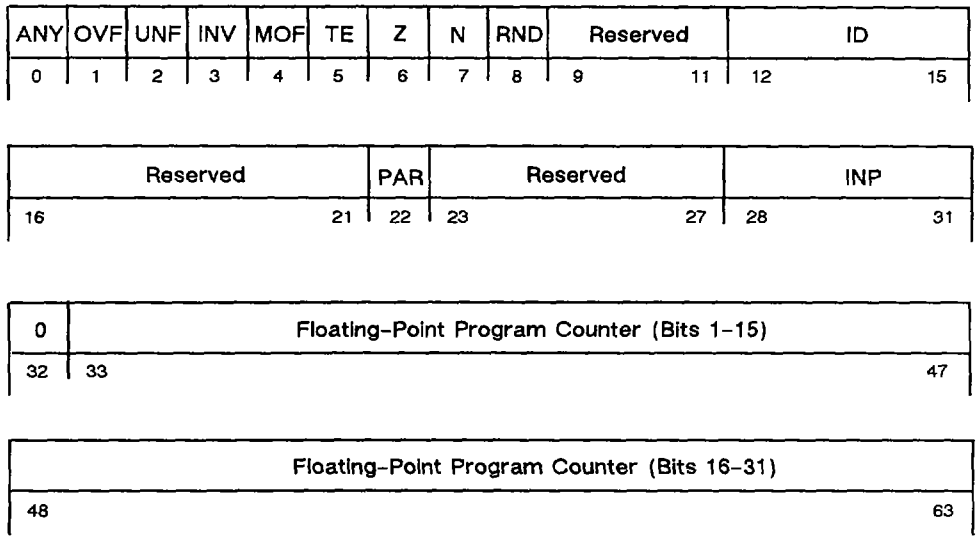


Table A-4 *FPSR contents*

Bits	Name	Contents or Function
0	ANY	Indicates any bits 1 through 4 is set to 1
1	OVF	Exponent overflow indicator
2	UNF	Exponent underflow indicator
3	INV	Invalid input argument
4	MOF	Mantissa overflow
5	TE	Trap enable; if set to 1, setting any bit 1 through 4 results in floating-point fault fault
6	Z	Zero bit
7	N	Negative bit
8	RND	Floating-point rounding mode
9-11	Reserved	Reserved for future use; must be set to 0
12-15	ID	Floating-point model; should be set to 0111
16-21	Reserved	Reserved for future use; should be set to 0
22	PAR	Floating-point operation flag (serial or parallel)
23-27	Reserved	Reserved for future use; should be set to 0
28-31	INP	Invalid input argument indicator
32	0	Should be set to 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

Three registers are described in this section: I/O channel definition register, I/O channel status register, and I/O channel mask register.

I/O Channel Definition Register Format

The I/O channel definition register (6000_8) provides status information. The format for this register is diagrammed as shown and described in Table A-5.

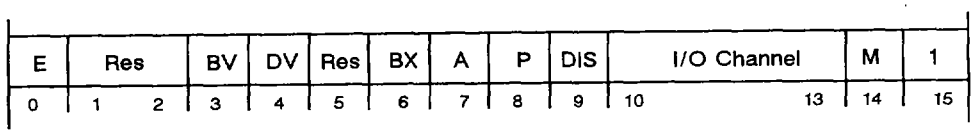


Table A-5 *I/O channel definition register contents*

Bits	Name	Contents or Function
0	E	Error flag (1) 1 Error occurred on I/O channel 0 Only when all other error bits are 0
1,2	Res	Reserved for future use; should be written with zeros
3	BV	BMC validity error flag (1) (2) 1 BMC validity protect error occurred
4	DV	DCH validity error flag (1) (2) 1 DCH validity protect error occurred
5	Res	Reserved for future use; should be written with zero
6	BX	BMC transfer flag 1 BMC transfer in progress (read only bit) Should be written with zero
7	A	BMC address error (1) (2) 1 Channel has detected address parity error
8	P	BMC data error (1) (2) 1 Channel has detected data parity error
9	DIS	Disable block transfer (1) 1 Disables BMC block transfers to/from I/O memory port (read/write bit)
10-13	I/O Channel	I/O channel type; always set to 0 Channel
14	M	DCH mode (1) 1 DCH mapping enabled
15	1	Always set to 1 (read only bit)

(1) *The ECLIPSE 16-bit* **IORST** *instruction clears these bits*

(2) *Writing to these bits with a 1 complements them*

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 and is described in Table A-6.

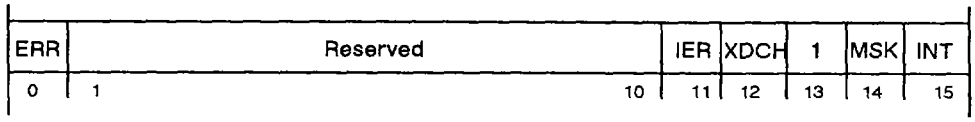


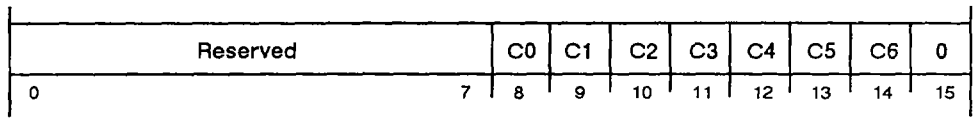
Table A-6 *I/O channel status register contents*

Bits	Name	Contents or Function
0	ERR	If 1, I/O channel has detected error indicated by IOC status register or memory parity error
1-10	Reserved	Reserved for future use
11	IER	If 1, memory parity error detected on read-memory operation
12	XDCH	If 1, extended DCH map slots and operations are supported.
13	1	Always set to 1
1	MSK	If 1, prevents all devices connected to channel from interrupting CPU However, INTA returns device code of any device whose DONE flag is set
15	INT	Interrupt pending; If 1, channel is attempting to interrupt 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 format of the register is diagrammed and described as follows.

WARNING: A CIO read to the I/O channel mask register produces undefined results.



Bits	Name	Contents of 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	Reserved	Set to 0

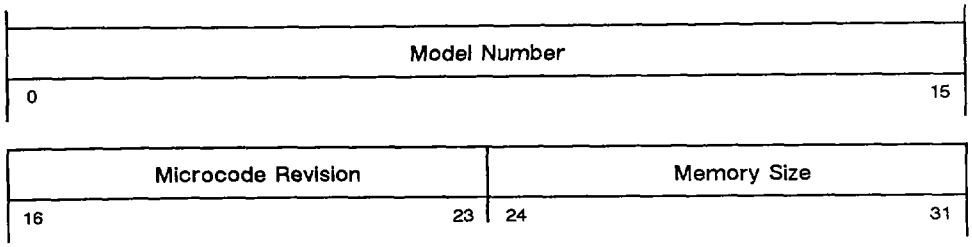
* *If 1, prevents all devices connected to this channel from interrupting the CPU. A system reset sets C0 to zero and C1 through C6 to one.*

CPU Identification

The three Load CPU Identification instructions -- **LCPID**, **ECLID**, and **NCLID** -- return the information to specified accumulators. Accumulator formats are diagrammed and described as follows.

LCPID and ECLID Instructions

The **LCPID** and **ECLID** instructions load a double word into AC0.

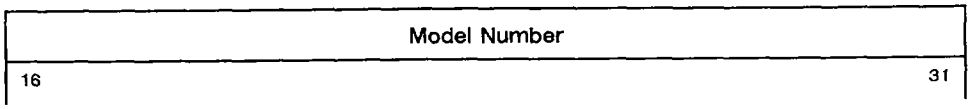


Bits	Name	Contents or Function
0–15	Model Number	Binary value of model number allocated to processor
16–23	Microcode Revision	Current microcode revision
24–31	Memory Size	Amount of physical memory available (in increments of 256 Kbytes) Ex.: 3 indicates 1 Mbyte; 7 indicates 2 Mbytes

NCLID Instruction

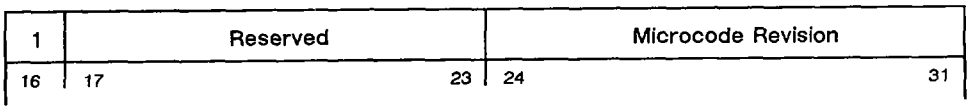
The **NCLID** instruction loads its result into the low-order 16 bits of accumulators AC0 through AC2. Bits 0 through 15 of each accumulator are undefined.

AC0



Bits	Name	Contents or Function
16–31	Model Number	Binary value of model number allocated to processor

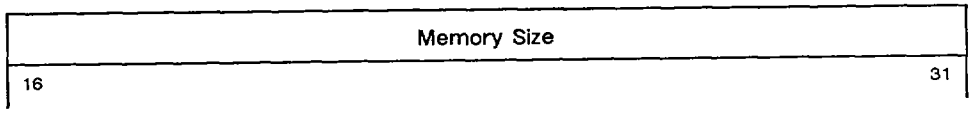
AC1



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
24–31	Microcode Revision	Current microcode revision

AC2



Bits	Name	Contents or Function
16–31	Memory Size	Amount of physical memory available in increments of 32 Kbytes.

ECLIPSE MV/20000 Series Supplement

This section supplements the "Register Fields" appendix by providing machine-specific information on ECLIPSE MV/20000 series computer systems. ECLIPSE MV/20000 series systems support all bits previously described and diagrammed in the "Register Fields" appendix with the exception of those discussed in this supplement.

DCH/BMC Status Registers

The ECLIPSE MV/20000 series systems differ in the implementation of some bits in the following registers:

I/O channel definition register—bits 9 through 13 are reserved.
I/O channel status register—bit 12 is always set to 1.
I/O channel mask register—bits 8 through 10 (MK0, MK1, and MK2) are defined.

CPU Identification

LCPID and **ECLID** instructions are formatted as described earlier. Listed below are the bits implemented on ECLIPSE MV/20000 series systems.

Bit	Name
0-15	Model Number 022046_8 (2326_{16}) ECLIPSE MV/20000 022047_8 (2327_{16}) ECLIPSE MV/20000 with FPU
16-23	Microcode Revision
24-31	Memory Size (physical memory available in increments of 256 Kbytes up to a maximum of 255, which equals 64 Mbytes)

The **NCLID** instruction loads AC1 and AC2 as formatted and described earlier. ECLIPSE MV/20000 series systems implement the following bits in AC0.

Bit	Name
0-15	Undefined
16-31	Model Number 022046_8 (2326_{16}) ECLIPSE MV/20000 022047_8 (2327_{16}) ECLIPSE MV/20000 with FPU

B

Fault Codes

This appendix contains fault codes (and some status codes) for the ECLIPSE MV/20000 series computers. Tables B–1 through B–5 interpret the codes for the following status and fault types:

- Protection faults
- Page faults
- Stack faults
- Power supply controller (PSC) status and faults
- Decimal and ASCII faults

Protection Faults

Table B–1 lists the meanings of codes returned in AC1 when an address translator protection fault occurs.

Table B–1 *Protection fault codes*

Octal Code in AC1	Meaning
0	Read violation
1	Write violation
2	Execute violation
3	Validity bit protection (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
14	Invalid microinterrupt return block
15	Unimplemented instruction fault

Page Faults

Table B-2 lists page fault codes that the processor stores in AC1.

Table B-2 *Page fault codes*

Octal Code in AC1	Meaning
0	Multiple ERCC fault
1	Reserved
2	Page table 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*

Octal Code in AC1	Meaning
000000	Overflow on every stack operation except SAVE and WMSP or ring crossing
000001	Underflow or overflow would occur if WMSP, WSSVR, WSSVS, WSAVR, and WSAVS instructions were executed (PC in return block refers to instruction that caused stack fault)
000002	Too many arguments on a cross ring call
000003	Stack underflow
000004	Overflow—return block has been pushed because of microinterrupt or fault

PSC Status and Faults

Table B-4 contains eight categories of code numbers. The PSC status code numbers are listed under category 0, and the PSC fault code numbers are listed under categories 1 through 7. (Notice that the least significant digit of each octal code number determines the code number's category.)

The leftmost column of Table B-4 lists the code numbers in hexadecimal, and the column beside it lists the code numbers 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 register AC1.

Table B-4 *PSC status and fault codes*

Hex Code	Octal Code	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 on	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/VNR 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 - VNR faults			
0B	013	Battery back-up fault indicated	Status fault
13	023	AC undervoltage (from VSR)	Status fault
1B	033	AC overvoltage (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

Table B-4 *PSC status and fault codes (cont.)*

Hex Code	Octal Code	Meaning	Result (see notes)
Category 4 - Power supply faults (includes undervoltage)			
04	004	+5V logic undervoltage	Fatal fault/retry
24	044	+5V memory 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	Fatal fault
25	045	+5V memory	Fatal fault
3D	075	+12V	Fatal fault
55	125	-5V	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 powerdown and stay down.

Fatal fault/retry—Causes the system to powerdown; PSC tries 3 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 and ASCII Faults

Table B-5 describes decimal and ASCII fault codes. The first and second columns list codes that appear in AC1; the third and fourth columns list instructions and conditions that cause faults.

Table B-5 *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 is too large to convert to data type |number|> $(10^{16}) - 1$ Number is 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	

C

Reserved Memory Locations

The information provided in this appendix applies to all MV/Family computers. Machine-specific details on the MV/8000's C/350 MAP is presented in the appropriate 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 fault handler addresses. The processor's translation of these locations is described in Tables C-1 and C-2. Specified addresses 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.

Page Zero Locations for Segment 0

When an interrupt specific to the system occurs, segment 0 locations 0 through 47_8 contain the information listed in Table C-1. The processor interprets all locations as logical with the address translator enabled.

Table C-1 *Page zero location for segment 0*

Word	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 Word Contents 2 high-order bits 3 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 (indirectable)
14	MV-specific stack fault	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	MV specific 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	ECLIPSE narrow stack fault address	Address of 16-bit narrow stack fault handler (indirectable)
44	XOP0 origin address	Address of beginning of narrow extended operations table
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 error/trap handler

Page Zero Locations for Segments 1 through 7

Table C-2 lists the page zero locations for segments 1 through 7 with the address translator enabled.

Table C-2 *Page zero locations for segments 1 through 7*

Word	Name	Contents or Function
0-7	Reserved	Reserved
10-11	MV-specific breakpoint address	Address of MV-specific breakpoint handler (indirectable)
12-13	WXOP origin address	Address of beginning of WXOP operations table (indirectable)
14	MV-specific stack fault address	Address of wide stack fault 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 gate array
36	Protection fault handler	Address of protection fault address handler (indirectable)
37	Fixed-point fault handler address	Address of fixed-point fault handler (indirectable)
40	Stack pointer	Address of top of 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 address	Address of narrow stack fault handler (indirectable)
44	XOP0 origin address	Address of beginning of narrow extended operations table
45	Floating-point fault address	Address of floating-point fault handler (indirectable)
46	Decimal/ASCII fault address	Address of Decimal/ASCII fault handler (indirectable)
47	DERR error handler	Address of DERR error/trap handler

D

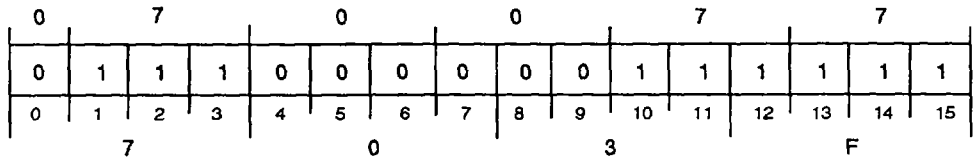
Load Control Store Instruction

This appendix describes the Load Control Store instruction and its associated microcode file.

WARNING: *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)

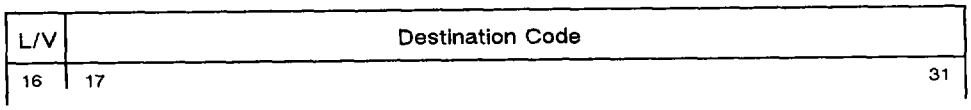


The Load Control Store instruction loads and verifies the soft internal states of the machine (e.g., the micro-store, decode rams, and scratch pad). In conjunction with bits 16 through 31 of the four accumulators, the **LCS** instruction loads and verifies, or verifies only, using the contents of a microcode file. The assembler recognizes **LCS** to be equivalent to **NIO,CPU**.

Since the **LCS** instruction loads a maximum of 16K words per instruction, it may be necessary to issue the instruction many times. This instruction is noninterruptible. Note that multi-processor systems insure that microcode blocks will be no greater than 1K words in length.

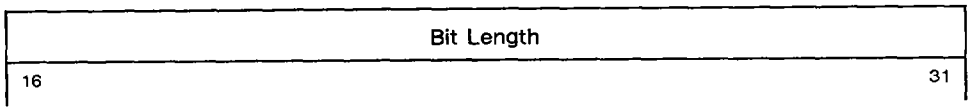
The formats of the accumulators are diagrammed and described next (bits 0 through 15 of each accumulator are undefined and unused).

AC0



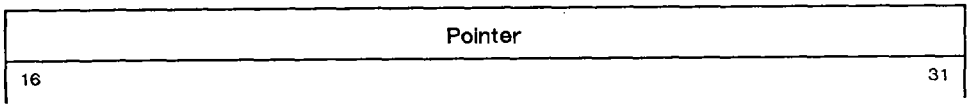
Bits	Name	Contents or Function
16	L/V	Load/verify option 0 Load and verify 1 Verify only
17–31	Destination Code	Indicates where data is to be loaded

AC1



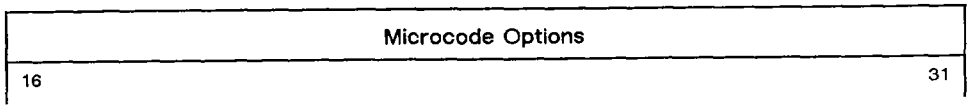
Bits	Name	Contents or Function
16–31	Bit Length	Bit length of code data

AC2



Bits	Name	Contents or Function
16–31	Pointer	Pointer to first block of data (indirectable)

AC3 (multiprocessor systems only)



NOTE: *Single processor systems 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)** 16–30 Reserved Unused 31 FPU Do NOT load microcode support for a hardware floating-point unit (FPU). This 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:

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 merely verify, only complete 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, the Title block may be preceded by Revision and/or Comment blocks and the End block may be followed by Comment blocks.
- Fill and Code blocks must be placed between the Title/End block pair.
- The Revision block 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 *Blocks of the microcode file format*

Block	Contents or Function
Title	Data about code word's bit length, and destination code. The program issuing LCS instruction uses 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 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 microcode file. Program issuing LCS instruction determines whether Revision blocks are ignored or output to system console

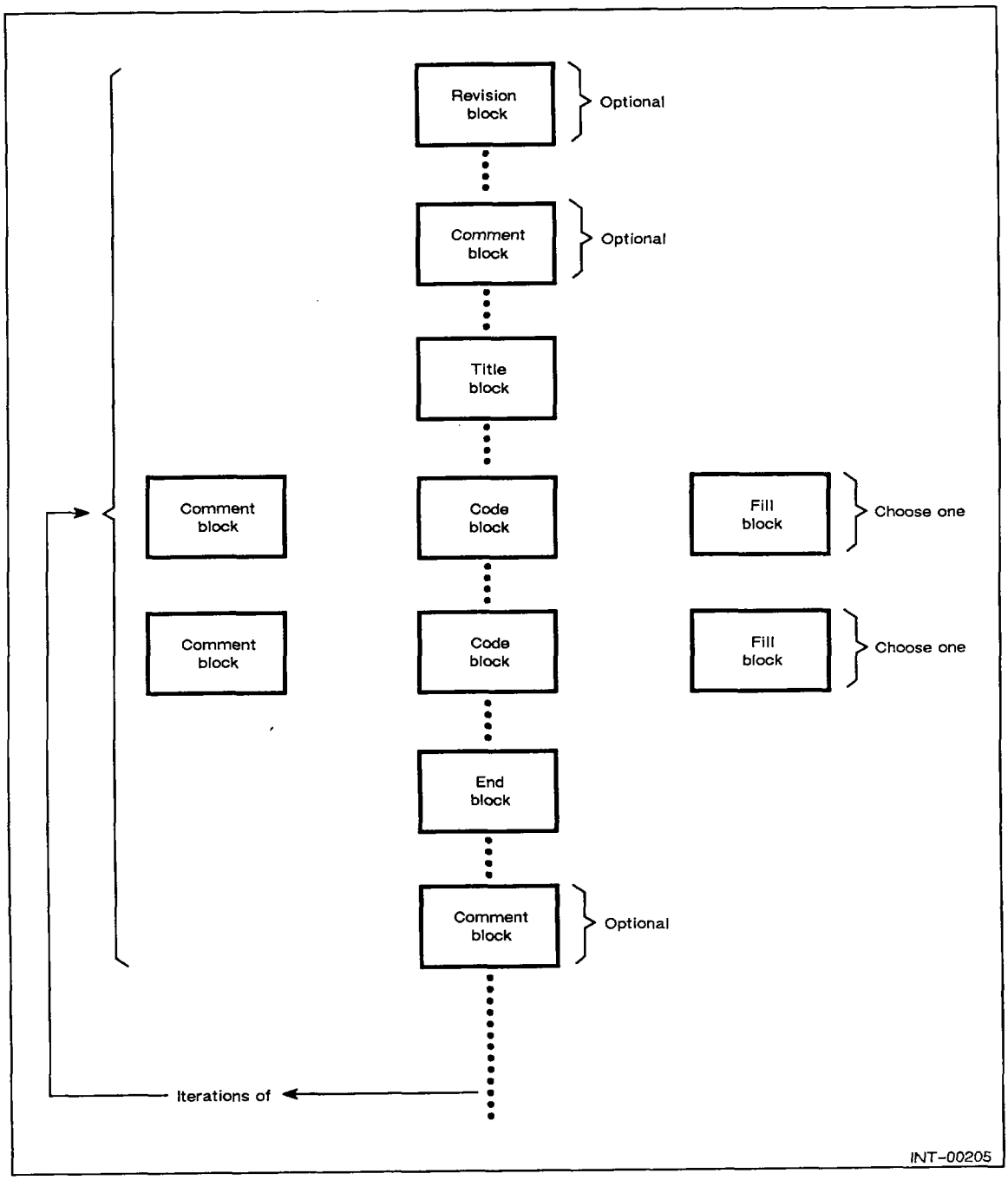


Figure D-1 *Microcode file format*

LCS Implementation

The effect of the **LCS** instruction on Code, Comment, Fill, and End blocks is described below. 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 contained 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 the five error conditions described in 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, and Table D-2 summarizes the words used in this format. Tables D-3, D-4, and D-6 through D-9 explain these words in more detail.

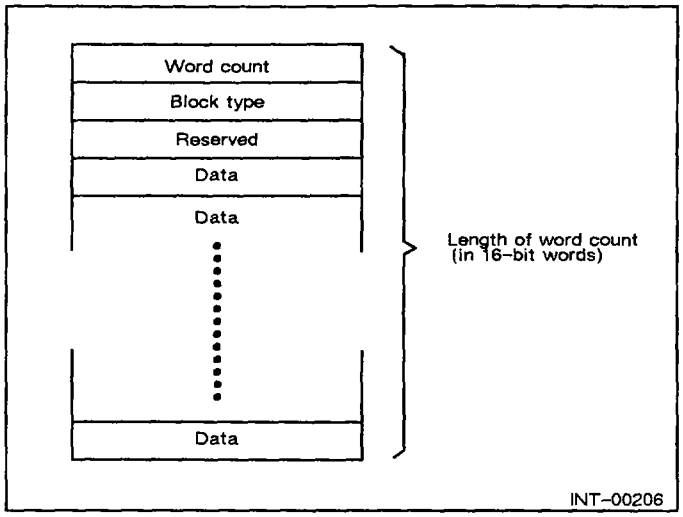


Figure D-2 *Microcode block format*

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	Indicates type of data contained in block
3	Reserved	Reserved for future use
4 + n	Data	Remaining words contain 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. (Only positive non-zero 16-bit integers in range 1 through 77777_8 are accepted by processor.)

The data from the first Title block is used by the program issuing the LCS instruction. For example:

ACO <--- Data word 4 (destination)
AC1 <--- Data word 1 (code word's bit length)

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 master (SCP); data word 2 must be an address (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 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 master.	Start master 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 accumulator (AC0, AC1, AC2) contents indicate 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 unexpected block type error will be returned.*
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 -- e.g., 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 (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 limited to 16 Kwords. 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.

E

Standard I/O Device Codes

Table E-1 *Standard I/O device codes*

Octal Code	Device Mnemonic	Priority Bit Mask	Description
00	--	--	Reserved
01	--	--	Available for use
02	--		Available for use
03	--		Reserved
* 04	PSC	13	Power supply controller
05	--	--	Reserved
06	MCAT	12	Multiprocessor adapter transmitter
07	MCAR	12	Multiprocessor adapter receiver
* 10	TTI	14	TTY input
* 11	TTO	15	TTY output
12	--	--	Reserved
13	--	--	Reserved
* 14	RTC	13	Real-time clock
15	--	--	Reserved
16	--	--	Reserved
17	LPT	12	Line printer
20	--	--	Reserved
21	--	--	Reserved
22	MTB	10	Magnetic tape
23	MTJ	10	Magnetic tape
24	--	--	Reserved
25	--	--	Reserved
26	DKB	9	Fixed-head DG/disk
27	DPF	7	DG/disk storage subsystem
30	IAC13	11	Intelligent asynchronous controller 13
31	IAC14	11	Intelligent asynchronous controller 14
32	IAC15	11	Intelligent asynchronous controller 15
33	DKP	7	Moving-head disk
34	ISC	4	Intelligent synchronous controller
35	--	--	Reserved
36	--	--	Reserved

Octal Code	Device Mnemonic	Priority Bit Mask	Description
37	--	--	Reserved
40	DCU	4	Data control unit
41	DCU1	4	Second data control unit
42	--	--	Reserved
* 43	PIT	11	Programmable interval timer
44	--	--	Reserved
* 45	SCP	15	Diagnostic remote processor
46	MCAT1	12	Second multiprocessor transmitter
47	MCAR1	12	Second multiprocessor receiver
50	IAC1	11	Intelligent asynchronous controller 1
51	IAC2	11	Intelligent asynchronous controller 2
52	IAC3	11	Intelligent asynchronous controller 3
53	IAC4	11	Intelligent asynchronous controller 4
54	IAC5	11	Intelligent asynchronous controller 5
55	IAC6	11	Intelligent asynchronous controller 6
56	IAC7	11	Intelligent asynchronous controller 7
57	LPT1	12	Second line printer
60	--	--	Reserved
61	--	--	Reserved
62	MTB1	10	Second magnetic tape
63	MTJ1	10	Second magnetic tape
64	--	--	Reserved
65	IAC	11/5	Host-to-IAC interface
66	DKB1	9	Second fixed-head DG/Disk
67	DPF1	7	Second DG/Disk storage subsystem
70	IAC8	11	Intelligent asynchronous controller 8
71	IAC9	11	Intelligent asynchronous controller 9
72	IAC10	11	Intelligent asynchronous controller 10
73	IAC11	11	Intelligent asynchronous controller 11
74	IAC12	11	Intelligent asynchronous controller 12
75	--	--	Reserved
76	--	--	Reserved
* 77	CPU	--	CPU and console functions

* *These devices are supported on the primary IOC only. The device codes are reserved on any additional IOCs.*

F

Context Block Formats

This appendix describes the formats of the two types of context blocks in ECLIPSE MV/20000 series computers.

- Type 1 uses words 0 through 17
- Type 2 uses words 0 through 50

All words in the context block contain information used by the microcode and other internal systems.

NOTES: *The context block does not save the floating-point state. To save this information, use a Push Floating-Point State instruction.*

The processor requires that the context block save area, the pointer to the save area, and all indirect chains accessed align on double-word boundaries.

Table F-1 shows the format of the context blocks.

Table F-1 *Context block format*

Words in Block	Contents or Function
0-1	PSR (word 0), 0 (word 1)
2-3	AC0
4-5	AC1
6-7	AC2
8-9	AC3
10-11	CARRY. PC of offending (executing) instruction
12-13	Next PC. Double word containing segment number in bits 1-3 of next instruction to execute. Processor uses this word to resolve on microcycle basis segment in which instruction is actually executing. Because most instructions cannot cross segment boundaries, this double word reflects same segment as program counter of executing instruction.
14-15	LAR, address that caused page fault
16	XIR—If the IXCT bit of the PSR is set, this word contains the opcode of the instruction. Otherwise, this word is undefined.
17	PCW (Processor Control Word) **Bits Contents** 0-23 Zeros 24-25 Source accumulator number 26-27 Destination accumulator number 28 IM[0] ALU width (0 = wide, 1 = narrow) 29 IM[1] LA width (0 = wide, 1 = narrow) 30 ION flag—If ION is modified, and it goes from 0 to 1—interrupts become enabled, however, an interrupt will not be taken until the next dispatch. 1 to 0—interrupts become disabled, and interrupts will not be taken at the next dispatch. 31 Context block length (0 = short, 1 = long)
18-19	NPC
20-21	AGR
22-23	GR0
24-25	GR1
26-27	GR2
28-29	GR3
30-31	GR4
32-33	GR5
34-35	WR
36-37	Correct LAR—LAR of the original page fault.
38-39	Scratch pad register 0
40-41	Scratch pad register 1
42	Microstack level count (times 2)
43	Microstack level 0 bottom of stack
44	Microstack level 1
45	Microstack level 2
46	Microstack level 3
47, 48, 49, or 50	Loop counter

G

Instruction Execution Times

This appendix includes a discussion of instruction execution times for the ECLIPSE MV/20000 series computers as well as Table G-3, which lists the average execution time of each instruction.

The discussion of the ECLIPSE MV/20000 series instruction execution times assumes the following:

- Physical memory modules consist of 4, 8, 16, or 32 Mbytes of memory.
- All logical-to-physical address translations are resident in the address translation unit (ATU), the data cache, and the instruction cache.
- The **EDIT** and **WEDIT** subopcodes—which process commercial numeric data—process data type 4. The source pointer into the data (*j*) is never moved out of the bounds of the data.
- No indirection is used in any instruction that can specify indirection.
- Bit 8 in the FPSR (the RND bit) is zero, disabling rounding in floating-point instructions.
- No store instruction writes into the same 2 Kbyte page as specified by the program counter (PC).

If these conditions do not apply, adjust the execution times following the instructions given in Tables G-1 and G-2.

Table G-1 *Adjusting memory references*

To Every Memory Reference	Add (microseconds)
If logical-to-physical address translation is not in the address translation unit (ATU) cache	
For one-level page table	0.340
For two-level page table	0.510
If indirection is specified	0.170/level of indirection
If a double-word reference address ends in 7_8	0.170
If data is not in data cache	0.170

Table G-2 *Adjusting any instruction*

To Any Instruction	Add (microseconds)
If any of the following faults occur	
Stack overflow/underflow	1.70
Fixed-point fault	2.55, +1.70 if stack fault
Floating-point fault	2.98, +1.70 if stack fault
Protection fault	2.55 (min), 3.83 (max)
If instruction is not in instruction cache	0.170
Store operation into same 2 Kbyte page as PC of currently executing instruction	0.340
Floating-point instruction with FPSR bit 8 (RND) bit set	0.425 without FPU 0.170 with FPU

Hardware FPU Considerations

This section describes the use of the parallel floating-point unit (FPU) on the ECLIPSE MV/20000. If your system does not have the hardware FPU option, you can skip this section.

Each ECLIPSE MV/20000 series central processing unit (CPU) optionally supports a hardware FPU that operates in parallel with the CPU. This FPU accelerates execution of the following types of instructions:

- Floating-point instructions such as **FAS**, **FLDD**, and **FSTS**
- Integer multiplication and division instructions such as **LWMUL** and **LNDIV**
- Commercial instructions such as Load Integer (**LDI**) and Store Integer (**STI**)
- IIS instructions (Intrinsic Instruction Set) such as **WFCOSD** and **WFEXPS**

The execution time of an instruction that uses the FPU can vary considerably according to the input values to the instruction. Thus, the quoted FPU execution times are average values only.

Because of the parallel nature of the FPU, many instructions that use the FPU are specified by two execution times—CPU and FPU. The total execution time depends on the amount of overlap that can be achieved in the instruction stream. If the floating-point instruction is interleaved between nonfloating-point instructions, the FPU execution time is hidden, since the CPU begins processing the nonfloating-point instruction immediately after dispatching the floating-point instruction to the FPU. Back-to-back floating-point instructions lose this overlap, so the FPU time must be considered.

This concept is best illustrated with an example, using the Floating Add Single instruction (**FAS**) that executes in 0.170 microseconds on the CPU and 0.255 microseconds on the FPU. Compare the two instruction sequences below, noting that **WMOV** is a nonfloating-point instruction with an execution time of 0.085 microseconds.

```
FAS   0,1          FAS   0,1
WMOV  1,2          FMOV  1,2
   :                  :
   :                  :
   :                  :
```

First, looking at the instruction sequence in the left column, **FAS** completes in 0.255 microseconds. In this sequence, the **WMOV** instruction does not use the FPU so that

0.085 microseconds of FPU execution time is hidden. The total execution time from the beginning of **FAS** to the completion of the **WMOV** is 0.255 microseconds. The visible execution time of the **FAS** instruction is therefore 0.170 microseconds. Now in the right column, **FAS** still executes in 0.255 microseconds, but the FPU must complete the **FAS** instruction before **FMOV** begins executing. The total execution time of the instruction sequence on the right is 0.340 microseconds. The visible execution time of the **FAS** instruction is therefore 0.255 microseconds.

For instructions in which the CPU and FPU are both involved (such as a floating-point load or store), it is not possible to achieve any overlap with nonfloating-point instructions, so only the total execution time is given.

The intrinsic instruction set (IIS) presents a special case of the CPU and FPU working interactively. In order to reduce interrupt latency, the CPU execution time for an IIS instruction (such as **WFCOSD**) will increase as the FPU execution time increases. The CPU can proceed to the next instruction in the instruction stream only after the FPU determines that the currently executing IIS instruction will complete within 12.5 microseconds. Because of this interaction between CPU and FPU, an average combined execution time for each IIS instruction is given in Table G-3.

Integer multiplication, integer division, and some commercial instructions also use the FPU interactively and cannot begin execution until the FPU has completed any previous floating-point instruction. It is not possible, therefore, to achieve any overlap between these instructions and actual floating-point instructions.

Finally, there are several CPU instructions that wait for the FPU to complete any previous floating-point instructions before proceeding. Because CPU instructions wait, their apparent execution time may increase when there is a previous floating-point instruction.

NOTE: *FPSR bit 22 (the PAR bit) controls the amount of overlap that can be achieved between floating-point and nonfloating-point instructions. When this bit is zero, the FPU operates in parallel mode as described above. When this bit is one, the FPU operates in serial mode. In serial mode, all floating-point instructions are allowed to complete before the CPU moves on to the next instruction in the instruction stream. Running the FPU in serial mode can seriously degrade system performance in a floating-point intensive environment.*

Non-Hardware FPU

The intrinsic instruction set is only implemented on the ECLIPSE MV/20000 series computer with the hardware FPU option. An IIS instruction on an ECLIPSE MV/20000 without the hardware FPU option is functionally equivalent to an **LPSHJ** instruction using the 32-bit displacement specified by the IIS instruction.

Interpreting Table G-3

Table G-3 lists the average execution time for all of the instructions implemented on the ECLIPSE MV/20000 series computer. The left column lists the instruction execution times of the ECLIPSE MV/20000 *without* the hardware FPU option. The right column lists the instruction execution times of the ECLIPSE MV/20000 *with* the hardware FPU option.

In Table G-3, the following symbols identify special instructions:

- # Instructions that wait for the FPU to complete processing before they begin to compute, even though they do not need to use the FPU.
- * ECLIPSE C/350-compatible instructions.
- - Instructions capable of specifying indirection.
- @ Intrinsic instruction set (IIS) that is only included with the hardware FPU option. (Actual execution time can vary considerably according to the input values.)
- ‡ Integer multiplication, integer division, and commercial instructions that use the FPU interactively.

Table G-3 *Instruction execution times*

ECLIPSE MV/20000 series computer		ECLIPSE MV/20000 series computer with FPU	
Mnemonic	Timing (microseconds)	Mnemonic	Timing (microseconds)
ADC *	.085 + .085 if skip	ADC *	.085 +. 085 if skip
ADD *	.085 + .085 if skip	ADD *	.085 +. 085 if skip
ADDI *	0.128	ADDI *	0.128
ADI *	.085	ADI *	.085
ANC *	.085 + .085 if skip	ANC *	.085 + .085 if skip
AND *	.085 + .085 if skip	AND *	.085 + .085 if skip
ANDI *	.085	ANDI *	.085
BAM *	0.553 + 0.298 (number of words moved)	BAM *	0.553 + 0.298 (number of words moved)
BKPT	1.70	BKPT	1.70
BLM *	1.02 + .043/word moved (min) 1.02 + 0.425/word moved (max)	BLM *	1.02 + .043/word moved (min) 1.02 + 0.425/word moved (max)
BTO *	0.468	BTO *	0.468
BTZ *	0.468	BTZ *	0.468
CLM *	0.340	CLM *	0.340 #
CMP *	0.850 + 0.298/byte (min) 0.850 + 0.468/byte (max)	CMP *	0.850 + 0.298/byte (min) 0.850 + 0.468/byte (max)
CMT *	0.850 + 0.850/byte	CMT *	0.850 + 0.850/byte
CMV *	0.850 +.021/byte (min) 0.850 + 0.425/byte (max)	CMV *	0.850 + .021/byte (min) 0.850 + 0.425/byte (max)
COB *	0.170 (min), 4.17 (max)	COB *	0.170 (min), 4.17 (max)
COM *	.085 + .085 if skip	COM *	.085 + .085 if skip
CRYTC	.085	CRYTC	.085
CRYTO	.085	CRYTO	.085
CRYTZ	.085	CRYTZ	.085
CTR *	Translate and move: 0.425 + 0.468/byte; Translate and compare: 0.425 + 0.765/byte	CTR *	Translate and move: 0.425 + 0.468/byte; Translate and compare: 0.425 + 0.765/byte
CVWN	0.170	CVWN	0.170
DAD *	0.255	DAD *	0.255
DEQUE	Queue not empty: 1.70 + .085 if AC1 = -1 + 0.255 if final element Queue empty: 0.425	DEQUE	Queue not empty: 1.70 + .085 if AC1 = -1 + 0.255 if final element Queue empty: 0.425
DERR	0.595	DERR	0.595 #
DHXL *	0.510	DHXL *	0.510
DHXR *	0.510	DHXR *	0.510
DIV *	1.76	DIV *	1.36 ‡
DIVS *	2.04	DIVS *	1.53 ‡
DIVX *	2.21	DIVX *	1.53 ‡
DLSH *	0.425	DLSH *	0.425
DSB *	0.255	DSB *	0.255

Table G-3 *Instruction execution times (cont.)*

ECLIPSE MV/20000 series computer		ECLIPSE MV/20000 series computer with FPU	
Mnemonic	Timing (microseconds)	Mnemonic	Timing (microseconds)
DSPA *	0.935 +.085 if entry = -1	DSPA *	0.935 +.085 if entry = -1 #
DSZ *	0.383 +.085 if skip	DSZ *	0.383 +.085 if skip
DSZTS	0.213 +.085 if skip	DSZTS	0.213 +.085 if skip
ECLID	0.170	ECLID	0.170
EDIT *	0.765 + sum of sub-op execution times processed	EDIT *	0.765 + sum of sub-op execution times processed
DADI	0.765	DADI	0.765
DAPS	0.680 (w/o add) 0.935 (with add)	DAPS	0.680 (w/o add) 0.935 (with add)
DAPT	0.680 (w/o add) 0.935 (with add)	DAPT	0.680 (w/o add) 0.935 (with add)
DAPU	0.850	DAPU	0.850
DASI	1.02 (type 4) 1.36 (type 5)	DASI	1.02 (type 4) 1.36 (type 5)
DDTK	1.28	DDTK	1.28
DEND	0.893	DEND	0.893
DICI	0.680 + 0.340/character insert	DICI	0.680 + 0.340/character insert
DIMC	0.935 + 0.340/character insert + 0.425 if j in either stack	DIMC	0.935 + 0.340/character insert + 0.425 if j in either stack
DINC	0.765	DINC	0.765
DINS	0.765	DINS	0.765
DINT	0.850	DINT	0.850
DMVA	0.935 + 0.255/character moved + 0.425 if j in either stack	DMVA	0.935 + 0.255/character moved + 0.425 if j in either stack
DMVC	0.935 + .255/character moved + 0.425 if j in either stack	DMVC	0.935 + .255/character moved + 0.425 if j in either stack
DMVF	1.02 0.595/digit moved + 0.425 if j in either stack	DMVF	1.02 + 0.595/digit moved + 0.425 if j in either stack
DMVN	0.850 + 0.680/digit moved + 0.425 if j in either stack	DMVN	0.850 + 0.680/digit moved + 0.425 if j in either stack
DMVO	1.36	DMVO	1.36
DMVS	1.19 + 0.595/digit moved + 0.425 if j in either stack	DMVS	1.19 + 0.595/digit moved + 0.425 if j in either stack
DNDF	0.765	DNDF	0.765
DSSO	0.510	DSSO	0.510
DSSZ	0.510	DSSZ	0.510
DSTK	1.11	DSTK	1.11
DSTO	0.510	DSTO	0.510
DSTZ	0.510	DSTZ	0.510
EDSZ *	0.383 +.085 if skip ~	EDSZ *	0.383 +.085 if skip ~
EISZ *	0.383 +.085 if skip ~	EISZ *	0.383 +.085 if skip ~

Table G-3 *Instruction execution times (cont.)*

ECLIPSE MV/20000 series computer Mnemonic	Timing (microseconds)	ECLIPSE MV/20000 series computer with FPU Mnemonic	Timing (microseconds)
EJMP *	0.255 ~	EJMP *	0.255 ~ #
EJSR *	0.255 ~	EJSR *	0.255 ~ #
ELDA *	.085 ~	ELDA *	.085 ~
ELDB *	.085	ELDB *	.085
ELEF *	.085 ~	ELEF *	.085 ~
ENQH	0.935 + 0.255 if queue not empty	ENQH	0.935 + 0.255 if queue not empty
ENQT	0.935 + .255 if queue not empty	ENQT	0.935 + .255 if queue not empty
ESTA *	.085	ESTA *	.085
ESTB *	.085	ESTB *	.085
FAB *	0.340	FAB *	0.170 CPU; 0.170 FPU
FAD *	1.53	FAD *	0.170 CPU; 0.255 FPU .085 CPU; 0.255 FPU
FAMD *	1.70	FAMD *	0.255 CPU; 0.340 FPU ~
FAMS *	1.23	FAMS *	0.170 CPU; 0.255 FPU ~
FAS *	1.15	FAS *	0.170 CPU; 0.255 FPU
FCLE *	0.765	FCLE *	0.170 CPU; 0.170 FPU
FCMP *	1.02	FCMP *	0.170 CPU; 0.340 FPU
FDD *	11.9	FDD *	0.170 CPU; 2.89 FPU
FDMD *	11.9	FDMD *	0.255 CPU; 2.98 FPU ~
FDMS *	4.25	FDMS *	0.170 CPU; 1.53 FPU ~
FDS *	4.25	FDS *	0.170 CPU; 1.53 FPU
FEXP *	0.935	FEXP *	0.170 CPU; 0.170 FPU
FFAS *	1.23	FFAS *	0.510 ‡
FFMD *	1.28 ~	FFMD *	0.510 ~ ‡
FHLV *	0.850	FHLV *	0.170 CPU; 0.255 FPU
FINT *	1.36	FINT *	0.170 CPU; 0.170 FPU
FLAS *	1.79	FLAS *	0.425
FLDD *	0.425 ~	FLDD *	0.170 CPU ~
FLDS *	0.340 ~	FLDS *	0.170 CPU ~
FLMD *	1.79 ~	FLMD *	0.425 CPU ~
FLST *	0.765	FLST *	0.383 CPU
FMD *	9.35	FMD *	0.170 CPU; 0.765 FPU ~
FMMD *	9.35 ~	FMMD *	0.255 CPU; 0.850 FPU ~
FMMS *	2.55 ~	FMMS *	0.170 CPU; 0.425 FPU ~
FMOV *	0.510	FMOV *	0.170 CPU; 0.170 FPU
FMS *	2.55	FMS *	0.170 CPU; 0.425 FPU
FNEG *	0.595	FNEG *	0.170 CPU; 0.170 FPU
FNOM *	2.04	FNOM *	0.170 CPU; 0.170 FPU
FNS *	0.170	FNS *	0.170

Table G-3 *Instruction execution times (cont.)*

ECLIPSE MV/20000 series computer Mnemonic	Timing (microseconds)	ECLIPSE MV/20000 series computer with FPU Mnemonic	Timing (microseconds)
FPOP *	2.64	FPOP *	1.74 ‡
FPSH *	2.38	FPSH *	2.38 ‡
FRDS *	0.510	FRDS *	0.170 CPU; 0.170 FPU
FRH *	0.170	FRH *	0.170
FSA *	0.170	FSA *	0.170
FSCAL*	1.79	FSCAL*	0.255 CPU; 0.340 FPU
FSD *	1.53	FSD *	0.170 CPU; 0.255 FPU
FSEQ *	0.170 + .085 if skip	FSEQ *	0.298 + .085 if skip
FSGE *	0.170 + .085 if skip	FSGE *	0.298 + .085 if skip
FSGT *	0.170 + .085 if skip	FSGT *	0.298 + .085 if skip
FSLE *	0.170 + .085 if skip	FSLE *	0.298 + .085 if skip
FSLT *	0.170 + .085 if skip	FSLT *	0.298 + .085 if skip
FSMD *	1.70 ~	FSMD *	0.255 CPU; 0.340 FPU ~
FSMS *	1.23 ~	FSMS *	0.170 CPU; 0.255 FPU ~
FSND *	0.170 + .085 if skip	FSND *	0.298 + .085 if skip
FSNE *	0.170 + .085 if skip	FSNE *	0.298 + .085 if skip
FSNER*	0.170 + .085 if skip	FSNER*	0.298 + .085 if skip
FSNM *	0.170 + .085 if skip	FSNM *	0.298 + .085 if skip
FSNO *	0.170 + .085 if skip	FSNO *	0.298 + .085 if skip
FSNOD*	0.170 + .085 if skip	FSNOD*	0.298 + .085 if skip
FSNU *	0.170 + .085 if skip	FSNU *	0.298 + .085 if skip
FSNUD*	0.170 + .085 if skip	FSNUD*	0.298 + .085 if skip
FSNUO*	0.170 + .085 if skip	FSNUO*	0.298 + .085 if skip
FSS *	1.15	FSS *	0.170 CPU; 0.255 FPU
FSST *	0.680 ~	FSST *	0.340 ‡ ~
FSTD *	0.340 ~	FSTD *	0.340 ‡ ~
FSTS *	0.170 ~	FSTS *	0.170 ‡ ~
FTD *	0.170	FTD *	0.170
FTE *	0.298	FTE *	0.255
FXTD	.085	FXTD	.085
FXTE	0.213	FXTE	0.213
HLV *	.085 + 0.340 if ac is negative	HLV *	.085 + 0.340 if ac is negative
HXL *	0.255	HXL *	0.255
HXR *	0.255	HXR *	0.255
INC *	.085 +.085 if skip	INC *	.085 +.085 if skip
IOR *	.085	IOR *	.085
IORI *	.085	IORI *	.085
ISZ *	0.383 +.085 if skip ~	ISZ *	0.383 +.085 if skip ~
ISZTS	0.213 +.085 if skip ~	ISZTS	0.213 +.085 if skip ~
JMP *	0.255 ~	JMP *	0.255 ~ #
JSR *	0.255 ~	JSR *	0.255 ~ #
LCALL	Intra-ring: 0.425~ Cross-ring: 2.98 +0.255/argument	LCALL	Intra-ring: 0.425~ Cross-ring: 2.98 +0.255/argument #
LCPID	0.170	LCPID	0.170
LDA *	.085 ~	LDA *	.085 ~

Table G-3 *Instruction execution times (cont.)*

ECLIPSE MV/20000 series computer		ECLIPSE MV/20000 series computer with FPU	
Mnemonic	Timing (microseconds)	Mnemonic	Timing (microseconds)
LDAFP	.085	LDAFP	.085
LDASB	.085	LDASB	.085
LDASL	.085	LDASL	.085
LDASP	.085	LDASP	.085
LDATS	.085	LDATS	.085
LDB *	.085	LDB *	.085
LDI *	7.31 (Type 4, length 7)	LDI *	4.84 (Type 4, length 7) ‡
LDIX *	30.52 (Type 4, length 31)	LDIX *	13.86 (Type 4, length 31) ‡
LDSP	1.02 ~	LDSP	1.02 ~ #
LEF *	.085 ~	LEF *	.085 ~
LFAMD	1.70 ~	LFAMD	0.255 CPU; 0.340 FPU
LFAMS	1.23 ~	LFAMS	0.170 CPU; 0.255 FPU ~
LFDMD	11.9	LFDMD	0.255 CPU; 2.98 FPU ~
LFDMS	4.25	LFDMS	0.170 CPU; 1.53 FPU ~
LFLDD	0.425 ~	LFLDD	0.170 CPU ~
LFLDS	0.340 ~	LFLDS	0.170 CPU ~
LFLST	1.28 ~	LFLST	0.638 CPU ~
LFMMD	9.35 ~	LFMMD	0.255 CPU; 0.850 FPU ~
LFMMS	2.55 ~	LFMMS	0.170 CPU; 0.425 FPU ~
LFSMD	1.70 ~	LFSMD	0.255 CPU; 0.340 FPU ~
LFSMS	1.23 ~	LFSMS	0.170 CPU; 0.255 FPU ~
LFSST	1.02 ~	LFSST	0.510 ‡ ~
LFSTD	0.340 ~	LFSTD	0.340 ‡ ~
LFSTS	0.170 ~	LFSTS	0.170 ‡ ~
LJMP	0.255 ~	LJMP	0.255 # ~
LJSR	0.255 ~	LJSR	0.255 # ~
LLDB	.085	LLDB	.085
LLEF	.085 ~	LLEF	.085 ~
LLEFB	0.128	LLEFB	0.128
LMRF	0.595	LMRF	0.595
LNADD	0.170 ~	LNADD	0.170 ~
LNADI	0.213	LNADI	0.213
LNDIV	1.79 ~	LNDIV	1.36 ‡ ~
LNDO	0.255 (no termination) 0.510 (for termination)	LNDO	0.255 (no termination) 0.510 (for termination) #
LNDSZ	0.383 +.085 if skip	LNDSZ	0.383 +.085 if skip
LNISZ	0.383 +.085 if skip	LNISZ	0.383 +.085 if skip
LNLDA	.085 ~	LNLDA	.085 ~
LNMUL	1.36 ~	LNMUL	0.850 ‡ ~
LNSBI	0.213	LNSBI	0.213
LNSTA	.085 ~	LNSTA	.085 ~
LNSUB	0.170 ~	LNSUB	0.170 ~
LOB *	0.510 (min), 0.765 (max)	LOB *	0.510 (min), 0.765 (max)
LPEF	0.170 ~	LPEF	0.170 ~

Table G-3 *Instruction execution times (cont.)*

ECLIPSE MV/20000 series computer		ECLIPSE MV/20000 series computer with FPU	
Mnemonic	Timing (microseconds)	Mnemonic	Timing (microseconds)
LPEFB	0.340	LPEFB	0.340
LPHY	1.53 (1 or 2 level, valid) 1.19 (invalid)	LPHY	1.53 (1 or 2 level, valid) 1.19 (invalid)
LPSHJ	.255 ~	LPSHJ	.255 # ~
LPSR	.085	LPSR	.085
LPTE	1.70	LPTE	1.70
LRB *	0.808 (min), 1.15 (max)	LRB *	0.808 (min), 1.15 (max)
LSBRA	3.36	LSBRA	3.36
LSBRS	3.08	LSBRS	3.08
LSH *	.085	LSH *	.085
LSN *	0.850 + 0.510/leading zero digit	LSN *	0.850 + 0.510/leading zero digit
LSTB	.085	LSTB	.085
LWADD	0.170 ~	LWADD	0.170 ~
LWADI	.213 ~	LWADI	.213 ~
LWDIV	3.49 ~	LWDIV	1.79 ‡ ~
LWDO	.255 (no termination) 0.510 (for terminination)	LWDO	.255 (no termination) 0.510 (for termination) #
LWDSZ	0.383 + .085 if skip ~	LWDSZ	0.383 +.085 if skip ~
LWISZ	0.383 + .085 if skip ~	LWISZ	0.383 +.085 if skip ~
LWLDA	.085 ~	LWLDA	.085 ~
LWMUL	3.14 ~	LWMUL	1.11 ‡ ~
LWSBI	0.213 ~	LWSBI	0.213 ~
LWSTA	.085 ~	LWSTA	.085 ~
LWSUB	0.170 ~	LWSUB	0.170 ~
MOV *	.085 + .085 if skip	MOV *	.085 + .085 if skip
MSP *	0.510	MSP *	0.510
MUL *	1.19	MUL *	0.978 ‡
MULS *	1.275	MULS *	1.06 ‡
NADD	.085	NADD	.085
NADDI	0.128	NADDI	0.128
NADI	.085	NADI	.085
NBStc	1.86 + 0.255 (n-1) + 0.240 if end is encountered	NBStc	1.86 + 0.255 (n-1) + 0.240 if end is encountered
NCLID	0.980	NCLID	0.980
NDIV	1.87	NDIV	1.36 ‡
NEG *	.085 + .085 if skip	NEG *	.085 + .085 if skip
NFStc	1.86 + 0.255 (n-1) +0.340 if end is encountered	NFStc	1.86 + 0.255 (n-1) +0.340 if end is encounterd
NLDAI	.085	NLDAI	.085
NMUL	1.68	NMUL	0.850 ‡
NNEG	.085	NNEG	.085
NSALA	0.128 + .085 if skip	NSALA	0.128 + .085 if skip
NSALM	0.255 + .085 if skip	NSALM	0.255 + .085 if skip
NSANA	0.128 + .085 if skip	NSANA	0.128 + .085 if skip
NSANM	0.255 + .085 if skip	NSANM	0.255 + .085 if skip
NSBI	.085	NSBI	.085
NSUB	.085	NSUB	.085
ORFB	0.340 + 1.57 (count), count = AC0 + 1	ORFB	0.340 + 1.57 (count), count = AC0 + 1

Table G-3 *Instruction execution times (cont.)*

ECLIPSE MV/20000 series computer		ECLIPSE MV/20000 series computer with FPU	
Mnemonic	Timing (microseconds)	Mnemonic	Timing (microseconds)
PATU	0.383	PATU	0.383
PBX	1.79 + executed instruction	PBX	1.79 + executed instruction #
POP *	0.383 + .085/ac	POP *	0.383 + .085/ac
POPB *	0.978	POPB *	0.978 #
POPJ *	0.510	POPJ *	0.510 #
PSH *	0.383 + .085/ac	PSH *	0.383 + .085/ac
PSHJ *	0.510	PSHJ *	0.510 #
PSHR *	0.510	PSHR *	0.510
RRFB	0.510 + 0.510 (count), count = AC0 +1	RRFB	0.510 + 0.510 (count), count = AC0 +1
RSTR *	1.40	RSTR *	1.40 #
RTN *	0.978	RTN *	0.978 #
SAVE *	1.955	SAVE *	1.955 #
SAVZ	1.743	SAVZ	1.743 #
SBI *	.085	SBI *	.085
SEX	.085	SEX	.085
SGE *	.085 + .085 if skip	SGE *	.085 + .085 if skip
SGT *	.085 + .085 if skip	SGT *	.085 + .085 if skip
SMRF	.0.553	SMRF	0.553
SNB *	0.383 + .085 if skip	SNB *	0.383 + .085 if skip
SPTE	0.638	SPTE	0.638
SNOVR	0.170 + .085 if skip	SNOVR	0.170 + .085 if skip
SPSR	.085	SPSR	.085
STA *	.085 -	STA *	.085 -
STAFP	0.170	STAFP	0.170 #
STASB	0.170	STASB	0.170
STASL	0.170	STASL	0.170
STASP	.085	STASP	.085
STATS	.085	STATS	.085
STB *	.085	STB *	.085
STI *	9.01 (Type 4, length 7)	STI *	7.56 (Type 4, length 7) ‡
STIX *	37.74 (Type 4, length 31)	STIX *	25.16 (Type 4, length 31) ‡
SUB *	.085 + .085 if skip	SUB *	.085 + .085 if skip
SZB *	0.383 + .085 if skip	SZB *	0.383 + .085 if skip
SZBO *	0.723 + .085 if skip	SZBO *	0.723 + .085 if skip
VWP	0.383	VWP	0.383
VBP	0.255	VBP	0.255
WADC	.085	WADC	.085
WADD	.085	WADD	.085
WADDI	0.128	WADDI	0.128
WADI	.085	WADI	.085
WANC	.085	WANC	.085
WAND	.085	WAND	.085
WANDI	.085	WANDI	.085
WASH	0.425 (min), 0.850 (max)	WASH	0.425 (min), 0.850 (max)

Table G-3 *Instruction execution times (cont.)*

ECLIPSE MV/20000 series computer		ECLIPSE MV/20000 series computer with FPU	
Mnemonic	Timing (microseconds)	Mnemonic	Timing (microseconds)
WASHI	0.510 (min), 0.935 (max)	WASHI	0.510 (min), 0.935 (max)
WBLM	1.02 + .043/word (min) 1.02 + 0.425/word (max)	WBLM	1.02 + .043/word (min) 1.02 + 0.425/word (max)
WBR	0.255	WBR	0.255 #
WBStc	1.72 + 0.255 (n-1) + 0.340 if end is encountered	WBStc	1.72 + 0.255 (n-1) + 0.340 if end is encountered
WBTO	0.468	WBTO	0.468
WBTZ	0.468	WBTZ	0.468
WCLM	if acs< >acd 0.340 if no skip 0.425 if skip if acs = acd 0.468 if no skip 0.383 if skip	WCLM	if acs< >acd 0.340 if no skip # 0.425 if skip # if acs = acd 0.468 if no skip # 0.383 if skip #
WCMP	0.850 + 0.298/byte (min) 0.850 + 0.468/byte (max)	WCMP	0.850 + 0.298/byte (min) 0.850 + 0.468/byte (max)
WCMT	0.850 + 0.850/byte	WCMT	0.850 + 0.850/byte
WCMV	0.850 + .021/byte (min) 0.850 + 0.425/byte (max)	WCMV	0.850 + .021/byte (min) 0.850 + 0.425/byte (max)
WCOB	0.170 (min), 8.25 (max)	WCOB	0.170 (min), 8.25 (max)
WCOM	.085	WCOM	.085
WCST	0.850 + 0.510/byte	WCST	0.850 + 0.510/byte
WCTR	Translate & move: 0.425 + 0.468/byte; Translate & compare: 0.425 + 0.765/byte	WCTR	Translate & move: 0.425 + 0.468/byte: Translate & compare: 0.425 + 0.765/byte
WDCMP	Both strings same length: 1.28 + 1.19/digit Strings different lengths: 1.28 + 0.680/digit	WDCMP	Both strings same length: 1.28 + 1.19/digit Strings different lengths: 1.28 + 0.680/digit
WDDEC	3.74 (Type 4, length 7) (min) 12.07 (Type 4, length 7) (max)	WDDEC	3.74 (Type 4, length 7) (min) 12.07 (Type 4, length 7) (max)
WDINC	3.74 (Type 4, length 7) (min) 12.07 (Type 4, length 7) (max)	WDINC	3.74 (Type 4, length 7) (min) 12.07 (Type 4, length 7) (max)
WDIV	3.49	WDIV	1.79 ‡
WDIVS	4.76	WDIVS	2.47 ‡
WDMOV	7.82 (Type 4, length 7)	WDMOV	7.82 (Type 4, length 7)
WDPOP	3.27 (restartable) 9.35 (resumable)	WDPOP	3.27 (restartable) 9.35 (resumable)

Table G-3 *Instruction execution times (cont.)*

ECLIPSE MV/20000 series computer Mnemonic	Timing (microseconds)	ECLIPSE MV/20000 series computer with FPU Mnemonic	Timing (microseconds)
WEDIT	0.765 + sum of sub-op execution times processed	WEDIT	0.765 + sum of sub-op execution times processed
DADI	0.765	DADI	0.765
DAPS	0.680 (w/o add) 0.935 (with add)	DAPS	0.680 (w/o add) 0.935 (with add)
DAPT	0.680 (w/o add) 0.935 (with add)	DAPT	0.680 (w/o add) 0.935 (with add)
DAPU	0.850	DAPU	0.850
DASI	1.02 (type 4) 1.36 (type 5)	DASI	1.02 (type 4) 1.36 (type 5)
DDTK	1.28	DDTK	1.28
DEND	0.893	DEND	0.893
DICI	0.680 + 0.340/character insert	DICI	0.680 + 0.340/character insert
DIMC	0.935 + 0.340/character insert + 0.425 If j in either stack	DIMC	0.935 + 0.340/character insert + 0.425 If j in either stack
DINC	0.765	DINC	0.765
DINS	0.765	DINS	0.765
DINT	0.850	DINT	0.850
DMVA	0.935 + 0.255/character moved + 0.425 if j in either stack	DMVA	0.935 + 0.255/character moved + 0.425 if j in either stack
DMVC	0.935 + .255/character moved + 0.425 If j in either stack	DMVC	0.935 + .255/character moved + 0.425 If j in either stack
DMVF	1.02 0.595/digit moved + 0.425 if j in either stack	DMVF	1.02 + 0.595/digit moved + 0.425 if j in either stack
DMVN	0.850 + 0.680/digit moved + 0.425 if j in either stack	DMVN	0.850 + 0.680/digit moved + 0.425 if j in either stack
DMVO	1.36	DMVO	1.36
DMVS	1.19 + 0.595/digit moved + 0.425 if j in either stack	DMVS	1.19 + 0.595/digit moved + 0.425 if j in either stack
DNDF	0.765	DNDF	0.765
DSSO	0.510	DSSO	0.510
DSSZ .	0.510	DSSZ	0.510
DSTK	1.11	DSTK	1.11
DSTO	0.510	DSTO	0.510
DSTZ	0.510	DSTZ	0.510
WFACOSD	(not implemented)	WFACOSD	14.88 @ ‡
WFACOSS	(not implemented)	WFACOSS	6.38 @ ‡
WFASIND	(not implemented)	WFASIND	14.88 @ ‡

Table G-3 *Instruction execution times (cont.)*

ECLIPSE MV/20000 series computer Mnemonic	Timing (microseconds)	ECLIPSE MV/20000 series computer with FPU Mnemonic	Timing (microseconds)
WFASINS	(not implemented)	WFASINS	6.38 @ ‡
WFATAND	(not implemented)	WFATAND	16.66 @ ‡
WFATANS	(not implemented)	WFATANS	6.63 @
WFATN2D	(not implemented)	WFATN2D	21.76 @ ‡
WFATN2S	(not implemented)	WFATN2S	8.67 @
WFCOSD	(not implemented)	WFCOSD	9.61 @ ‡
WFCOSS	(not implemented)	WFCOSS	3.83 @
WFEXPD	(not implemented)	WFEXPD	11.14 @ ‡
WFEXPS	(not implemented)	WFEXPS	4.17 @
WFFAD	1.28	WFFAD	0.510 @ ‡
WFLAD	1.79	WFLAD	0.425
WFLG2D	(not implemented)	WFLG2D	14.11 @ ‡
WFLG2S	(not implemented)	WFLG2S	5.78 @
WFLNGD	(not implemented)	WFLNGD	14.11 @ ‡
WFLNGS	(not implemented)	WFLNGS	5.78 @
WFLOGD	(not implemented)	WFLOGD	14.11 @ ‡
WFLOGS	(not implemented)	WFLOGS	2.13 ‡
WFPOP	3.49	WFPOP	2.24 ‡
WFPSH	2.98	WFPSH	2.98 @ ‡
WFPWRD	(not implemented)	WFPWRD	25.08 @ ‡
WFPWRS	(not implemented)	WFPWRS	9.86 @ ‡
WFSIND	(not implemented)	WFSIND	9.61 @ ‡
WFSINS	(not implemented)	WFSINS	3.83 @
WFSQRD	(not implemented)	WFSQRD	5.97 @
WFSQRS	(not implemented)	WFSQRS	3.15 @
WFStc	1.72 + 0.255 (n-1) + 0.340 if end is encountered	WFStc	1.72 + 0.255 (n-1) + 0.340 if end is encountered
WFTAND	(not implemented)	WFTAND	12.24 @ ‡
WFTANS	(not implemented)	WFTANS	5.44 @
WHLV	.085 + 0.340 if ac is negative	WHLV	.085 + 0.340 if ac is negative
WINC	.085	WINC	.085
WIOR	.085	WIOR	.085
WIORI	.085	WIORI	.085
WLDAI	.085	WLDAI	.085
WLDB	0.170	WLDB	0.170
WLDI	7.31 (Type 4, length 7)	WLDI	4.84 (Type 4, length 7) ‡
WLDIX	30.52 (Type 4, length 31)	WLDIX	13.86 (Type 4, length 31) ‡
WLMP	1.02 + 1.28 (number of BMC/DCH slots, min.)	WLMP	1.02 + 1.28 (number of BMC/DCH slots, min.)
WLOB	0.510 (min), 0.765 (max)	WLOB	0.510 (min), 0.765 (max)
WLRB	0.808 (min), 1.15 (max)	WLRB	0.808 (min), 1.15 (max)
WLSH	.085	WLSH	.085
WLSHI	0.170	WLSHI	0.170
WLSI	0.170	WLSI	0.170
WLSN	0.850 + 0.510/leading zero digit	WLSN	0.850 + 0.510/leading zero digit

Table G-3 *Instruction execution times (cont.)*

ECLIPSE MV/20000 series computer		ECLIPSE MV/20000 series computer with FPU	
Mnemonic	Timing (microseconds)	Mnemonic	Timing (microseconds)
WMESS	0.723 + .255 if successful	WMESS	0.723 + .255 if successful
WMOV	.085	WMOV	.085
WMOVR	.085	WMOVR	.085
WMSP	0.468	WMSP	0.468
WMUL	3.14	WMUL	1.11 ‡
WMULS	3.14	WMULS	1.19 ‡
WNADI	0.128	WNADI	0.128
WNEG	.085	WNEG	.085
WPOP	0.128 + .085/ac	WPOP	0.128 + .085/ac
WPOPB	0.765 intra-ring	WPOPB	0.765 intra-ring #
	2.00 cross-ring		2.00 cross-ring #
WPOPJ	.255	WPOPJ	.255 #
WPSH	0.170 + .085/ac	WPSH	0.170 + .085/ac
WRSTR	2.08 intra-ring	WRSTR	2.08 intra-ring #
	2.55 cross-ring		2.55 cross-ring #
WRTN	0.765 intra-ring	WRTN	0.765 intra-ring #
	2.08 cross-ring		2.08 cross-ring
WSALA	0.128 + .085 if skip	WSALA	0.128 + .085 if skip
WSALM	0.255 + .085 if skip	WSALM	0.255 + .085 if skip
WSANA	0.128 + .085 if skip	WSANA	0.128 + .085 if skip
WSANM	0.255 + .085 if skip	WSANM	0.255 + .085 if skip
WSAVR	0.638	WSAVR	0.638 #
WSAVS	0.638	WSAVS	0.638 #
WSBI	.085	WSBI	.085
WSEQ	.085 + .085 if skip	WSEQ	.085 + .085 if skip
WSEQI	0.128 + .085 if skip	WSEQI	0.128 + .085 if skip
WSGE	.085 + .085 if skip	WSGE	.085 + .085 if skip
WSGT	.085 + .085 if skip	WSGT	.085 + .085 if skip
WSGTI	0.128 + .085 if skip	WSGTI	0.128 + .085 if skip
WSKBO	0.213 + .085 if skip	WSKBO	0.213 + .085 if skip
WSKBZ	0.213 + .085 if skip	WSKBZ	0.213 + .085 if skip
WSLE	.085 + .085 if skip	WSLE	.085 + .085 if skip
WSLEI	0.128 + .085 if skip	WSLEI	0.128 + .085 if skip
WSLT	.085 + .085 if skip	WSLT	.085 + .085 if skip
WSNB	0.383 + .085 if skip	WSNB	0.383 + .085 if skip
WSNE	.085 + .085 if skip	WSNE	.085 + .085 if skip
WSNEI	0.128 + .085 if skip	WSNEI	0.128 + .085 if skip
WSSVR	0.765	WSSVR	0.765 #
WSSVS	0.765	WSSVS	0.765 #
WSTB	0.170	WSTB	0.170
WSTI	9.01 (Type 4, length 7)	WSTI	7.56 (Type 4, length 7) ‡
WSTIX	37.74 (Type 4, length 31)	WSTIX	25.16 (Type 4, length 31) ‡
WSUB	.085	WSUB	.085

Table G-3 *Instruction execution times (cont.)*

ECLIPSE MV/20000 series computer Mnemonic	Timing (microseconds)	ECLIPSE MV/20000 series computer with FPU Mnemonic	Timing (microseconds)
WSZB	0.383 + .085 if skip	WSZB	0.383 + .085 if skip
WSZBO	0.723	WSZBO	0.723
WUGTI	0.128 + .085 if skip	WUGTI	0.128 + .085 if skip
WULEI	0.128 + .085 if skip	WULEI	0.128 + .085 if skip
WUSGE	.085 + .085 if skip	WUSGE	.085 + .085 if skip
WUSGT	.085 + .085 if skip	WUSGT	.085 + .085 if skip
WXCH	0.170	WXCH	0.170
WXOP	1.83	WXOP	1.83 #
WXOR	.085	WXOR	.085
WXORI	.085	WXORI	.085
XCALL	intra-ring: 0.425 ~ cross-ring: 2.98 + 0.255/argument	XCALL	intra-ring: 0.425 ~ # cross-ring: 2.98 + 0.255/argument #
XCH *	0.170	XCH *	0.170
XCT *	0.765 + executed instruction (# if executed instruction marked with #)	XCT *	0.765 + executed instruction (# if executed instruction marked with #)
XFAMD	1.70 ~	XFAMD	0.255 CPU; 0.340 FPU ~
XFAMS	1.23 ~	XFAMS	0.170 CPU; 0.255 FPU ~
XFDMD	11.9 ~	XFDMD	0.255 CPU; 2.98 FPU ~
XFDMS	4.25 ~	XFDMS	0.170 CPU; 1.53 FPU ~
XFLDD	0.425 ~	XFLDD	0.170 CPU ~
XFLDS	0.340 ~	XFLDS	0.170 CPU ~
XFMMD	9.35 ~	XFMMD	0.255 CPU; 0.850 FPU ~
XFMMS	2.55 ~	XFMMS	0.170 CPU; 0.425 FPU ~
XFSMD	1.70 ~	XFSMD	0.255 CPU; 0.340 FPU ~
XFSMS	1.23 ~	XFSMS	0.170 CPU; 0.255 FPU ~
XFSTD	0.340 ~	XFSTD	0.340 ‡ ~
XFSTS	0.170 ~	XFSTS	0.170 ‡ ~
XJMP	0.255 ~	XJMP	0.255 # ~
XJSR	0.255 ~	XJSR	0.255 # ~
XLDB	.085 ~	XLDB	.085 ~
XLEF	.085 ~	XLEF	.085 ~
XLEFB	0.128	XLEFB	0.128
XNADD	0.170 ~	XNADD	0.170 ~
XNADI	0.213 ~	XNADI	0.213 ~
XNDIV	1.79 ~	XNDIV	1.36 ‡ ~
XNDO	0.255 (no termination) 0.510 (for termination)	XNDO	0.255 (no termination) 0.510 (for termination) #
XNDSZ	0.383 + .085 if skip ~	XNDSZ	0.383 + .085 if skip ~
XNISZ	0.383 + .085 if skip ~	XNISZ	0.383 + .085 if skip ~
XNLDA	.085 ~	XNLDA	.085 ~
XNMUL	1.36 ~	XNMUL	0.850 ‡ ~
XNSBI	0.213 ~	XNSBI	0.213 ~
XNSTA	.085 ~	XNSTA	.085 ~
XNSUB	0.170	XNSUB	0.170 ~
XOP0 *	2.00	XOP0 *	2.00 #

Table G-3 *Instruction execution times (concluded)*

ECLIPSE MV/20000 series computer		ECLIPSE MV/20000 series computer with FPU	
Mnemonic	Timing (microseconds)	Mnemonic	Timing (microseconds)
XOR *	.085	XOR *	.085
XORI *	.085	XORI *	.085
XPEF	0.170 ~	XPEF	0.170 ~
XPEFB	0.340	XPEFB	0.340
XPSHJ	0.255 ~	XPSHJ	0.255 ‡ ~
XSTB	.085	XSTB	.085
XVCT	6.80 base level, in-line 4.85 intermediate level, in-line 6.38 base level interrupt 5.36 intermediate level, interrupt (minimums)	XVCT	6.80 base level, in-line 4.85 intermediate level, in-line 6.38 base level interrupt 5.36 intermediate level interrupt (minimums)
XWADD	0.170 ~	XWADD	0.170 ~
XWADI	0.213 ~	XWADI	0.213 ~
XWDIV	3.49 ~	XWDIV	1.79 ‡ ~
XWDO	0.255 (no termination) 0.510 (for termination)	XWDO	0.255 (no termination) 0.510 (for termination) ‡
XWDSZ	0.383 + .085 if skip ~	XWDSZ	0.383 + .085 if skip ~
XWISZ	0.383 + .085 if skip ~	XWISZ	0.383 + .085 if skip ~
XWLDA	.085 ~	XWLDA	.085 ~
XWMUL	3.14 ~	XWMUL	1.11 ‡ ~
XWSBI	0.213 ~	XWSBI	0.213 ~
XWSTA	.085 ~	XWSTA	.085 ~
XWSUB	0.170	XWSUB	0.170 ~
ZEX	.085	ZEX	.085

Glossary

ADDRESS TRANSLATOR. Used in demand paging to translate the specified logical address to its physical equivalent.

ATTRIBUTE BLOCK. Consists of unsigned 32-bit integers that are 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 Read Attribute instruction 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 reference 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.

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 referenced by an instruction or routine from secondary storage (e.g., disks) to main memory as they are needed.

DEQUEUING. The process of removing a data element from a queue.

DOUBLE-WORD. Two consecutive words of memory (4 bytes or 64 bits).

ENQUEUING. 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 font is 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 double-word table of signed and unsigned 32-bit integers.

FORM MASK. Used to implement palette sharing, a technique that helps programs to share 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. Giga-byte (230 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 ECLIPSE instruction set, referred to as narrow instructions; and the 32-bit instruction set, referred to as wide instructions. The 16-bit instructions supported by the 32-bit processor and also supported by 16-bit ECLIPSE computers (such as the ECLIPSE C/350 computer), are referred to as ECLIPSE 16-bit 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 X 16**y
where 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 acesses 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 16-bit program development and upward program compatibility.

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 zeros produce undefined results except as noted in this manual.

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 ten zeros following it; the logical address denoting the first logical address in a page.

PAGE FAULT. The condition caused by a reference to a page that is not resident in main memory.

PAGEFRAME. 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.

PAGE TABLE. One or more page table pages that completely specify the logical to physical address translation for a segment. A page table may be one- or two-level. A one-level page table consists of a single page table page, and a two-level page table consists of one page table page whose PTEs point to page table pages.

PAGE TABLE ENTRY (PTE). A double-word used by the processor in translating logical addresses to physical addresses. A PTE contains a page number and bits defining the page protocol.

PAGE TABLE PAGE. A page consisting of 512 PTEs.

PAGE ZERO. Denotes the first 2K bytes of a segment, but sometimes used to denote 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 privilege. 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.

RING. Protection mechanisms that safeguard 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, containing the address of the page table 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 and retrieve them from the stack in reverse order. 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. Referefnces a state that may or may not have been altered after an instruction execution.

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 is used to facilitate 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.

WIDE INSTRUCTIONS. These instructions manipulate data with lengths of 8, 16, or 32 bits. The mnemonics of the instructions indicate the size of the data fields referenced. 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 double words 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 are 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).

Index

Within the index, the letter "f" following a page entry indicates "and the following page"; the letters "ff" following a page entry indicate "and the following pages." Instruction mnemonics are printed in boldface type (such as **ADD**); instruction names are printed with initial capital letters (such as Extended Add Immediate).

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Data General

TP_____

TIPS ORDER FORM
Technical Information & Publications Service

BILL TO:
COMPANY NAME_____
ADDRESS _____
CITY _____
STATE_____ ZIP _____
ATTN: _____

SHIP TO: (if different)
COMPANY NAME_____
ADDRESS _____
CITY _____
STATE_____ ZIP _____
ATTN: _____

QTY	MODEL #	DESCRIPTION	UNIT PRICE	LINE DISC	TOTAL PRICE

(Additional items can be included on second order form)

[Minimum order is $50.00]

Tax Exempt #_____
or Sales Tax (if applicable)

TOTAL	
Sales Tax	
Shipping	
TOTAL	

CUT ALONG DOTTED LINE

METHOD OF PAYMENT

☐ Check or money order enclosed
For orders less than $100.00

☐ Charge my ☐ Visa ☐ MasterCard
Acc't No._____ Expiration Date_____

☐ Purchase Order Number:_____

SHIP VIA

☐ DGC will select best way (U.P.S or Postal)

☐ Other:
☐ U.P.S. Blue Label
☐ Air Freight
☐ Other _____

NOTE: ORDERS LESS THAN $100, INCLUDE $5.00 FOR SHIPPING AND HANDLING.

Person to contact about this order _____ Phone _____ Extension _____

Data General Corporation
Educational Services/TIPS
MS G214
2400 Computer Drive
Westboro, MA 01580
(617) 366-8911, Extension 1610

Buyer's Authorized Signature Date
(agrees to terms & conditions on reverse side)

Title

DGC Sales Representative (If Known) Badge #

DISCOUNTS APPLY TO MAIL ORDERS ONLY

012-1780

educational services

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 shown on the reverse hereof which is accepted by DGC.

1. **PRICES**
   Prices for DGC publications will be as stated in the Educational Services Literature Catalog in effect at the time DGC accepts Buyer's order or as specified on an authorized DGC quotation in force at the time of receipt by DGC of the Order Form shown on the reverse hereof. Prices are exclusive of all excise, sales, use or similar taxes and, therefore are subject to an increase equal in amount to any tax DGC may be required to collect or pay on the sale, license or delivery of the materials provided hereunder.

2. **PAYMENT**
   Terms are net cash on or prior to delivery except where satisfactory open account credit is established, in which case terms are net thirty (30) days from date of invoice.

3. **SHIPMENT**
   Shipment will be made F.O.B. Point of Origin. DGC normally ships either by UPS or U.S. Mail or other appropriate method depending upon weight, unless Customer designates a specific method and/or carrier on the Order Form. In any case, DGC assumes no liability with regard to loss, damage or delay during shipment.

4. **TERM**
   Upon execution by Buyer and acceptance by DGC, this agreement shall continue to remain in effect until terminated by either party upon thirty (30) days prior written notice. It is the intent of the parties to leave this Agreement in effect so that all subsequent orders for DGC publications will be governed by the terms and conditions of this Agreement.

5. **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.

6. **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.

7. **DISCLAIMER OF WARRANTY**
   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 SUPPLIED HEREUNDER.

8. **LIMITATIONS OF LIABILITY**
   IN NO EVENT SHALL DGC BE LIABLE FOR (I) ANY COSTS, DAMAGES OR EXPENSES ARISING OUT OF OR IN CONNECTION WITH ANY CLAIM BY ANY PERSON THAT USE OF THE PUBLICATION OF INFORMATION CONTAINED THEREIN INFRINGES ANY COPYRIGHT OR TRADE SECRET RIGHT OR (II) ANY INCIDENTIAL, SPECIAL, DIRECT OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOSS OF DATA, PROGRAMS OR LOST PROFITS.

9. **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. 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.

DISCOUNT SCHEDULES

DISCOUNTS APPLY TO MAIL ORDERS ONLY.

LINE ITEM DISCOUNT

5-14 manuals of the same part number - 20%
15 or more manuals of the same part number - 30%

DISCOUNTS APPLY TO PRICES SHOWN IN THE CURRENT TIPS CATALOG ONLY.

Data General

TIPS ORDERING PROCEDURE:

Technical literature may be ordered through the Customer Education Service's Technical Information and Publications Service (TIPS).

1. Turn to the TIPS Order Form.
2. Fill in the requested information. If you need more space to list the items you are ordering, use an additional form. Transfer the subtotal from any additional sheet to the space marked "subtotal" on the form.
3. Do not forget to include your MAIL ORDER ONLY discount. (See discount schedules on the back of the TIPS Order Form.)
4. Total your order. (MINIMUM ORDER/CHARGE after discounts of $50.00.)

   If your order totals less than 100.00, enclose a certified check or money order for the total (include sales tax, or your tax exempt number, if applicable) plus $5.00 for shipping and handling.
5. Please indicate on the Order Form if you have any special shipping requirements. Unless specified, orders are normally shipped U.P.S.
6. Read carefully the terms and conditions of the TIPS program on the reverse side of the Order Form.
7. Sign on the line provided on the form and enclose with payment. Mail to:

   Data General Corporation
   Educational Services/TIPS
   MS G214
   2400 Computer Drive
   Westboro, MA 01580
   (617) 366-8911, Extension 1610
8. We'll take care of the rest!

educational services

moisten & seal

CUSTOMER DOCUMENTATION COMMENT FORM

Your Name _____ Your Title _____

Company _____

Street _____

City _____ State _____ Zip _____

We wrote this book for you, and we made certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve the manual. Please take a few minutes to respond. Thank you.

Manual Title _____ Manual No. _____

Who are you?
☐EDP/MIS Manager ☐Analyst/Programmer ☐Other _____
☐Senior Systems Analyst ☐Operator _____
☐Engineer ☐End User

How do you use this manual? *(List in order: 1 = Primary Use)*

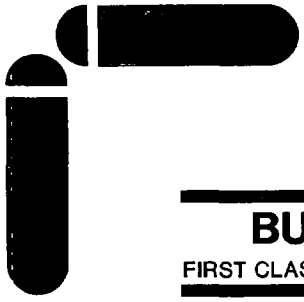
___ Introduction to the product ___ Tutorial Text ___ Other
___ Reference ___ Operating Guide _____

fold

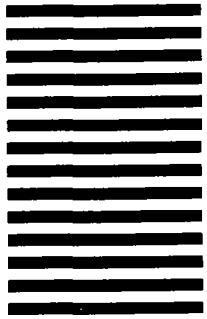
		Yes	No
About the manual:	Is it easy to read?	☐	☐
	Is it easy to understand?	☐	☐
	Are the topics logically organized?	☐	☐
	Is the technical information accurate?	☐	☐
	Can you easily find what you want?	☐	☐
	Does it tell you everything you need to know?	☐	☐
	Do the illustrations help you?	☐	☐

If you wish to order manuals, use the enclosed TIPS Order Form (USA only).

Comments:



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA 01772

Postage will be paid by addressee

DataGeneral

Customer Documentation
MS E-219
4400 Computer Drive
Westboro, MA 01581-9973