Data General

Customer Documentation

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

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

014-001372-01

Ordering No. 014-001372

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

Notice

ECLIPSE MV/Family (32-Bit) Systems Instruction Dictionary
014-001372-01

Revision History:

Original Release - January 1988
First Revision - July 1988

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

Preface

The *ECLIPSE® MV/Family (32-Bit) Systems Instruction Dictionary* describes each instruction in the ECLIPSE MV/Family instruction set alphabetically according to instruction mnemonic. The first volume in this two-volume global set, *ECLIPSE® MV/Family (32-Bit) Systems Principles of Operation* (DGC No. 014-001371), explains the processor-independent concepts and functions to an assembly language programmer. Processor-dependent information, available in machine-specific supplements, complements the two-volume global set.

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

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

NOTE: *All references in this manual to other chapters, sections, or appendixes pertain to chapters, sections, and appendixes in the first volume of this set,* "ECLIPSE MV/Family (32-Bit) Systems Principles of Operation."

Coordinating Machine-Specific Supplements

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

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

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

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

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

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

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

End of Preface

1

Instruction Dictionary

The Instruction Dictionary presents the ECLIPSE® MV/Family instruction set. The instructions appear in alphabetical order by the Data General Assembler instruction mnemonic. Each machine-specific supplement contains an appendix, "Instruction Execution Times," which lists the instructions supported by that particular machine (if an instruction is not listed in the appendix, it is not supported by that machine).

Instructions, which specify signed or unsigned values, assume the inputs are in the appropriate formats; otherwise results may be undefined.

Table 1-1 lists the abbreviations and symbols used in the Instruction Dictionary. The example following the table presents the standard format for each instruction description.

Table 1-1 *Abbreviations and symbols*

Abbreviation or Symbol	Description	Abbreviation or Symbol	Description
[ ]	The square brackets indicate an optional argument or arguments. Omit the square brackets when you include an optional argument with an Assembler statement. Square brackets also enclose a data indicator in the shorthand "Function" and "Parameters" descriptions.	acd	Destination fixed-point accumulator.
		AND	Logical AND.
		BCD	Binary coded decimal.
		CRY	Carry.
		dec#	Decimal number.
		displacement	Signed 8-, 15- or 31-bit integer.
		E	Effective address.
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.	(E)	Contents of E.
		fp#	Floating-point number.
		fp#s	Single-precision floating-point number.
		fp#d	Double-precision floating-point number.
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.	fpac	Floating-point accumulator.
		fpacs	Source floating-point accumulator.
		fpacd	Destination floating-point accumulator.
#	Unsigned value.	FPSR()	Floating-point status register (flags).
2#	Signed integer (two's complement).	I	16- or 32-bit signed or unsigned integer.
$\overline{XXX}$	One's complement.	index	Addressing mode specifier (0, 1, 2, 3).
&	Treat these items as one (concatenate).	n	Unsigned integer from 1 to 4.
(#-#)	Indicates a range of bits (from # to #).	norm	Normalized floating-point number.
*	Multiplication.	OR	Logical inclusive OR.
**	Exponentiation.	*Overflow*	Temporary condition.
+	Addition or positive value.	PC	Program counter.
-	Subtraction or negative value.	PSR()	Processor status register (flags).
/	Division.	PTE	Page table entry.
→	Returns result to.	R/W	Read or write.
≠>	Does not return result.	SBR	Segment base register.
<-->	Swap contents.	skip	If condition is true, skip the next word.
?=?	Comparison	stack	Wide or narrow stack (instruction dependent).
=	Equal to.	wfp/fp	Wide/narrow frame pointer.
≠	Not equal to.	wsb	Wide stack base.
>	Greater than.	wsl/sl	Wide/narrow stack limit
<	Less than.	wsp/sp	Wide/narrow stack pointer.
?	Undefined contents.	x	Contents unknown or undefined.
@	Indirection specifier.	XOR	Logical exclusive OR.
@(AC#)	At address specified in AC#.		
ac	Fixed-point accumulator.		
acs	Source fixed-point accumulator.		

Example

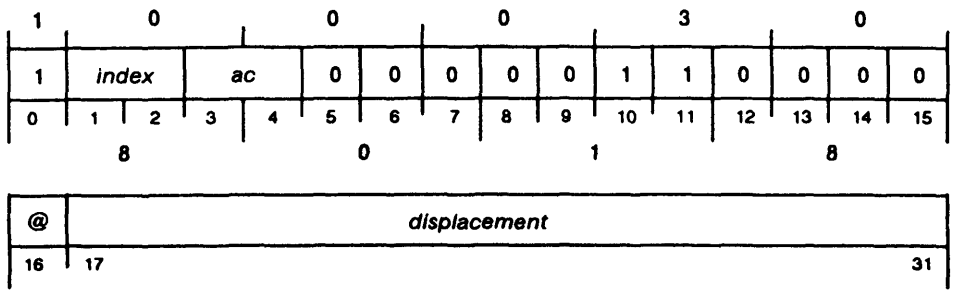
① Do Nothing Twice (Extended Displacement) XDUMME

③ Privileged Instruction ②

④ **XDUMME** *ac,[@]displacement[,index]*

④a (exception return)
(normal return)

⑤ ⑥ ⑦



⑧

Function: *ac* - 1 → (E); (E) → *ac*; *ac* + 1 → *ac*
ALU carry -> CRY

Parameters: None

NOTE: This instruction does not do much of anything.

⑨ **XDUMME** subtracts 1_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 1_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 The Load Accumulator instruction places a word from memory into an accumulator.

⑬ Exceptions

If **XDUMME** does anything, a protection fault occurs.

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 all **MNEMONICs** throughout this manual).

(3) Special category of instruction.

Within this manual there are some special instruction categories for ECLIPSE MV/Family computers:

• Privileged Instruction	ECLIPSE MV/Family (32-bit) instructions executable only in segment 0.
• ECLIPSE Instruction	ECLIPSE (16-bit) compatible instructions (all of these instructions will execute on an ECLIPSE C/350 system).
• Edit Sub-opcode	ECLIPSE (16-bit) and ECLIPSE MV/Family (32-bit) instructions logically executable only within an Edit sub-program initiated by an **EDIT** or **WEDIT** instruction.
• Intrinsic Instruction	ECLIPSE MV/Family (32-bit) instructions belonging to the optional Instrinsic Instruction Set (IIS).
• Graphics Instruction	ECLIPSE MV/Family (32-bit) instructions belonging to the optional Graphics Instruction Set (GIS).
• Multiprocessor Instruction	ECLIPSE MV/Family (32-bit) instructions executable only on systems capable of supporting more than one central processor.

All other instructions are ECLIPSE MV/Family (32-bit) instructions executable in any segment.

(4) Instruction **MNEMONIC** with *arguments* (*arguments* are *lowercase italic* throughout this manual).

(4a) Instructions which may skip the next sequential 16-bit word indicate the conditions that update the program counter. Instructions which either load another value into the program counter or normally 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 actual length of the instruction.*

Additional bit boxes may or may not include the octal/hexadecimal codings, dependent on their contents. For instance, 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 identical to the descriptions in the *ECLIPSE MV/Family Instruction Reference Booklet.*

(9) Paragraph(s) describing the instruction action.

(10) Listing of *arguments* with a description of their contents before and after execution.

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 described otherwise.

Instructions which use only certain portions of an argument use a modified format. For example, an instruction which uses bits 16-31 of an accumulator lists its accumulator argument as *ac*(16-31). This format holds true even if the result returned is greater than the initial value, for instance, *ac*(16-31) initially contains a 16-bit value, yet the final result may occupy all 32 bits of the accumulator (and is described as doing so). Note that all accumulator bits which are unused are considered unchanged following execution of the instruction, unless specified otherwise.

In general, the result retains the initial precision (32-bit floating-point addition produces a 32-bit floating-point result). Exceptions are given in the instruction description.

(10a) The argument, [@]*displacement*[,*index*], is treated as a single entity within the descriptions. Upon instruction completion, memory locations remain unchanged unless explicitly changed by the executing instruction or otherwise noted within the instruction description.

(11) The "Registers, Flags, and Stacks" section 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. 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 term "Unchanged" indicates that the contents of an item are used by the instruction and remain the same after instruction execution.

The term "Unaffected" describes a temporary state (such as *Overflow*) that is not affected by the instruction.

The term "Undefined" (after execution or because of a fault condition) means that no useful information can be inferred from the results.

The four fixed-point accumulators (AC0-AC3) and the four floating-point accumulators (FPAC0-FPAC3) are described individually only if they are used or affected by the instruction.

The results of operations that complement Carry for fixed-point ECLIPSE (16-bit) instructions are

Unsigned:	0 => integer > 65,535
Signed:	-32,768 > integer > 32,767

Overflow is a temporary condition which applies only while the instruction is executing (refer to the chapter, "Fixed-Point Computing"). The results of operations that create an overflow condition for fixed-point, signed ECLIPSE MV/Family 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 processor status register overflow bit (OVR). In general, instructions which operate on unsigned values leave *Overflow* 0 or unaffected; instructions which operate on signed values affect *Overflow* from the arithmetic/logic unit (ALU).

The Program Counter (PC) is always updated (following instruction execution) in one of the following ways:

The previous PC value plus the length of the currently executing instruction (a value of one is added to the PC for each 16-bit word in the instruction).

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; ECLIPSE MV/Family (32-bit) instructions affect the wide stack.

(12) "Related Instructions" lists applicable instructions in the following categories:

Generic — helpful in 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 (such as, the 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 perform almost identical functions).

(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 in each instruction description are

Faults taken upon exception conditions, such as a protection fault for an invalid address.

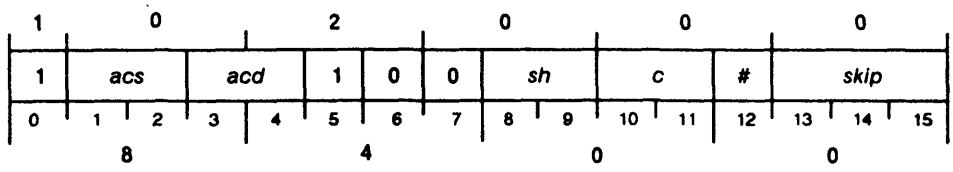
The majority of floating-point instructions do not check for unnormalized, or ill-formed, data. Operations on this data may produce erroneous results. Use the **FNOM** instruction to normalize floating-point data.

(14) Examples for each instruction range from a single line of code through complete subroutines. For further explanation of the assembly language statements, refer to the manual, *AOS/VS Macroassembler (MASM) Reference Manual.*

Add Complement ADC

ECLIPSE Instruction

ADC[*c*][*sh*][*#*] *acs*,*acd*[,*skip*]
(*skip* false return)
(*skip* true return)



Function: $\overline{acs} + acd \rightarrow acd$

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 (*c*) on initial value of Carry before performing operation (opcode). Following table gives values of *c*, as indicated by 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 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*, as indicated by 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, as indicated by 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 indicated 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

A 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).

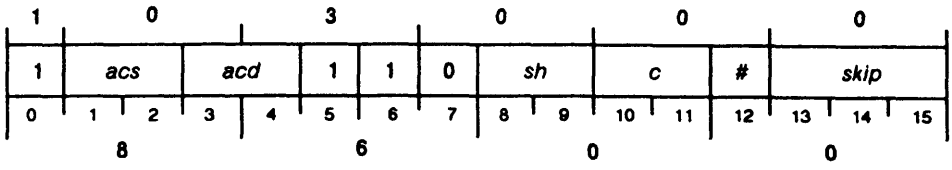
Example

```
ADCZ 1,2,SNR        ;Sets Carry to 0, adds complement
                    ;of AC1 to AC2, placing the result into AC2.
WBR ZEROANS         ;If the result of the addition is not 0,
INC 2,2             ;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*, as indicated by 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*, as indicated by 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, as indicated by 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 indicated 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

A 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)

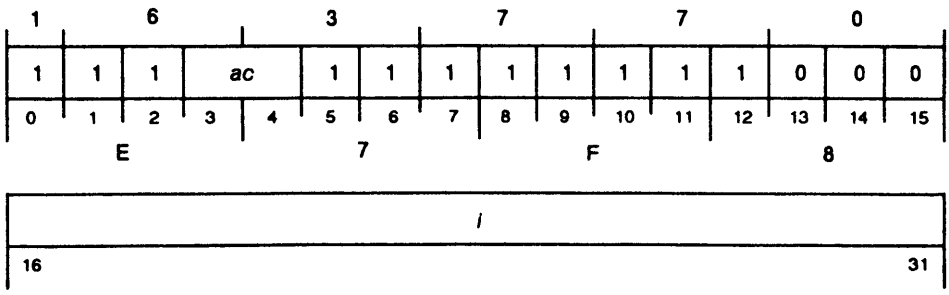
Do not specify **ADD** with no-load option (#) in combination with either never skip or always skip option. Instruction may not end in 1000_2 or 1001_2 (reserved for other instructions).

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 0.
```

Extended Add Immediate — ADDI

ADDI *i,ac*



Function: $i + ac \rightarrow ac$

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 in the range 1 to 4 to an accumulator.

NADDI, WADDI, WNADI Add a signed 16- or 32-bit immediate value to an accumulator.

Exceptions

None

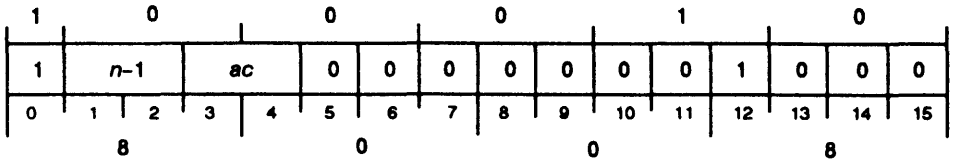
Example

```
    LDA    0,TWELVE     ;Load AC0 with 12.
    ADDI   5,0          ;Add 5 to AC0.
    . . .               ;AC0(16-31) now contains 17. Bits 0-15 undefined.
TWELVE:  .WORD 12.
```

Add Immediate ADI

ECLIPSE Instruction

ADI *n,ac*



Function: $n + ac \rightarrow ac$

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. Since the assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field, you should code exact value 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 in range 1 to 4 to an accumulator.

Exceptions

None

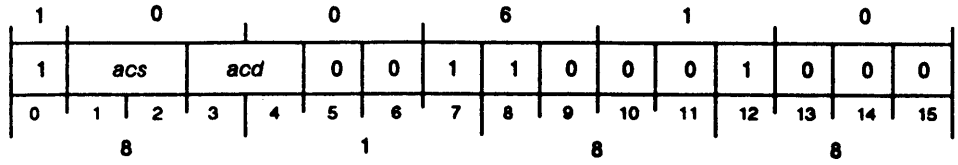
Example

```
ADI 4,2    ;Assume that AC2 contains 177775₈. After this 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 takes the logical complement of the contents of *acs*. ANC then forms the logical AND of this value and the contents of *acd*, placing the result in *acd*. For instance, the instruction sets a bit position in the result to 1 if the corresponding bit in *acs* contains 0 and *acd* contains 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

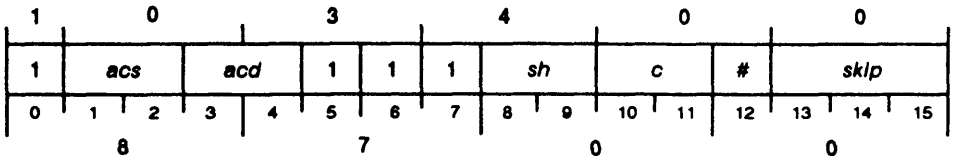
Example

```
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*, as indicated by 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*, as indicated by 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, as indicated by 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 indicated 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

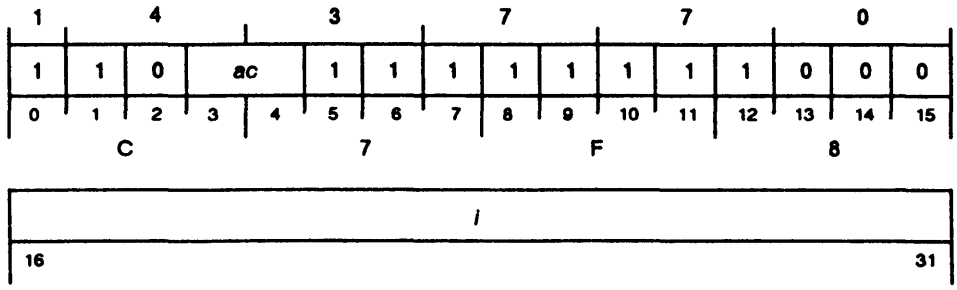
Do not specify **AND** with no-load option (#) in combination with either never skip or always skip option. Instruction may not end in 1000_2 or 1001_2 (reserved for other instructions).

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 forms the logical AND of 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

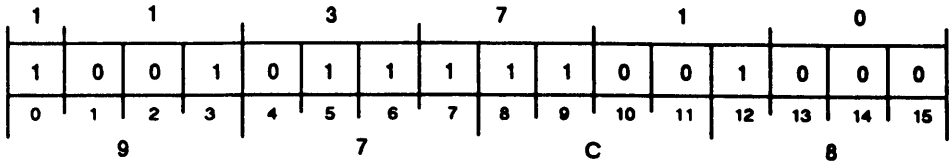
Example

```
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, contents 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, contains 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 or values into AC0 and AC1.
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.

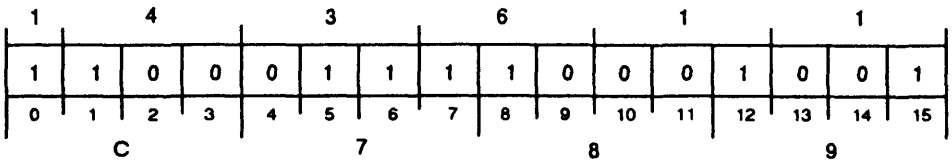
If the result of any add operation is greater than $32,768_8$, no indication is given.

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 doublewords → 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. (The "Subroutine" section in the chapter, "Program Flow Management," provides more information on the Breakpoint instruction.)

Arguments

None

Registers, Flags, and Stacks

AC0-AC3	Unused
Carry	Indeterminate
Overflow	Indeterminate
PC	After return block pushed, 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 return block pushed.
Stack	Wide stack contains wide-return block (value of PC in return block is address of **BKPT**.)

Related Instructions

PBX	The Pop Block and Execute instruction returns from the breakpoint handler if you do not remove **BKPT**.
WPOPB	The Wide Pop Block Block instruction 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.

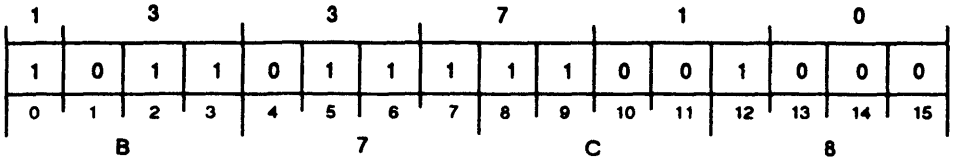
Example

```
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, this 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
WCMV	Wide Character Move. Use this instruction to move a large number of bytes and words or to move pages (if properly aligned).

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 the number is outside these bounds, no data is moved and the contents of the accumulators remain unchanged.

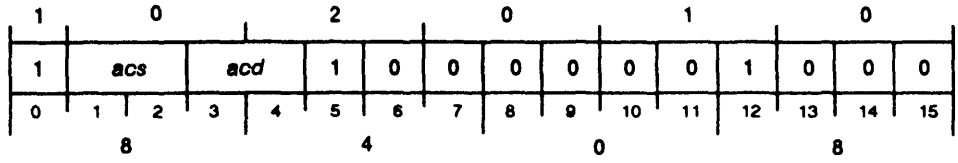
Example

```
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 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)	Contains high-order 16 bits of 32-bit bit pointer. If same accumulator is specified for *acd*, instruction treats accumulator contents as low-order 16 bits of bit pointer and assumes high-order 16 bits are 0. After execution, contents unchanged.
acd(16–31)	Contains 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

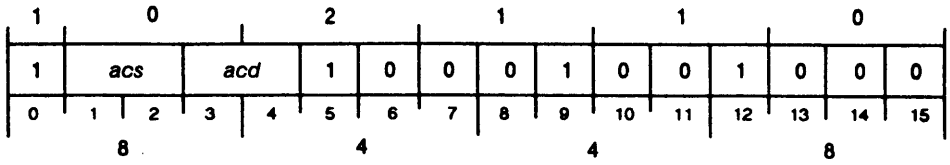
Example

```
ELEF   0,FLAGS      ;Get address of word containing flags.
SUB    1,1          ;Get a 0 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 0).
```

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 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) Contains high-order bits of 32-bit bit pointer. If same accumulator is specified 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) Contains 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

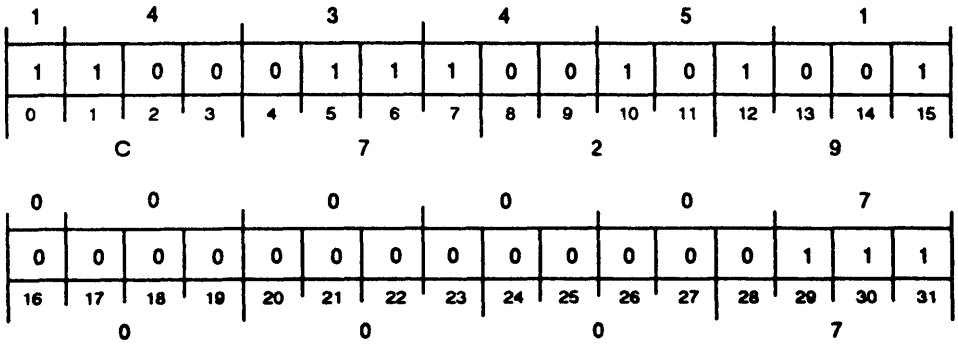
Example

```
    ELEF   0,FLAGS      ;Get address of word containing flags.
    SUB    1,1          ;Get a 0 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).
```

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 if 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:

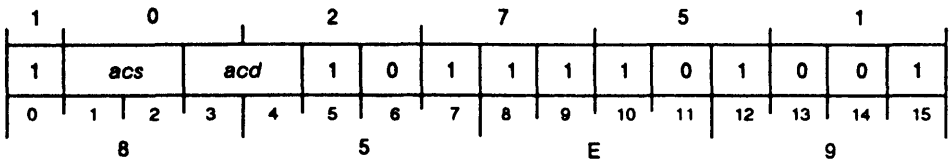
Condition	Error Code
Non-existent processor	2
Processor failure	4
Illegal option	6

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

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

Command I/O CIO

CIO *acs,acd*



Function: R/W ⟶ I/O system bus

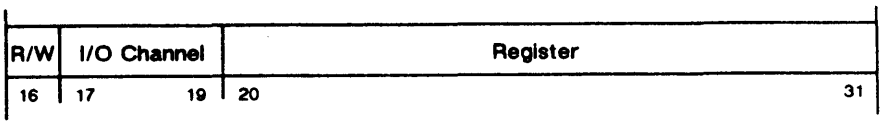
Parameters: *acs*(16-31) = #(command) ⟶ unchanged
If *acs*(16): 1 = write; *acd*(16-31) = data
0 = read; result ⟶ *acd*(16-31)
acs(17-19) = I/O channel number
acs(20-31) = Register address

CIO asserts a read or write data command from an accumulator onto the I/O channel bus. Use this instruction to read Data Channel and Burst Multiplexor Channel (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 I/O 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	R/W	Read or write indicator. A 0 specifies a read operation, a 1 specifies a write operation.
17-19	I/O Channel	I/O channel indicator. I/O channel numbers range from 0 to 7 (default is 0). Refer to the chapter, "Device Management," for definition of I/O channels on a specific system.
20-31	Register	Address of a DCH/BMC map register or slot (in range from 0000 to 7777_8). Refer to the chapter, "Device Management" for definition of DCH/BMC registers.

After execution, contents unchanged.

acd(16-31) Temporary data storage.

On a read command, receives data from specified *acs* address.

On a write command, contains data to be transmitted to specified *acs* 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

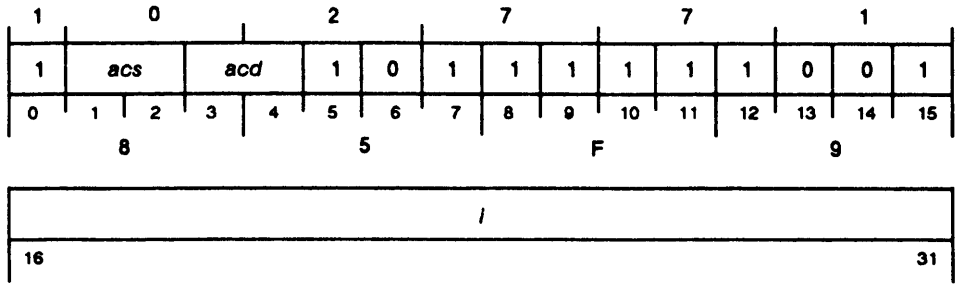
None

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. Use this instruction to read Data Channel and Burst Multiplexor Channel (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 I/O 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 for *acd*, *i* contains command.

If specification is different from *acd*, contents logically ORed with *i* to form command to be transmitted.

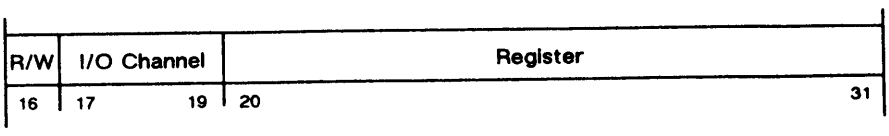
After execution, contents unchanged.

acd(16-31) Temporary data storage.

On a read command, receives data from specified *acs* address.

On a write command, contains data to be transmitted to specified *acs* address.

Immediate definition of the command, or is logically ORed with *acs* to derive the command. Command format is



Bits	Name	Contents or Function
16	R/W	Read or write indicator. A 0 specifies a read operation, a 1 specifies a write operation.
17-19	I/O Channel	I/O channel indicator. I/O channel numbers range from 0 to 7 (default is 0). Refer to the chapter, "Device Management," for definition of I/O channels on a specific system.
20-31	Register	Address of a DCH/BMC map register or slot (in range from 0000 to 7777_8). Refer to the chapter, "Device Management," for definition of DCH/BMC registers.

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

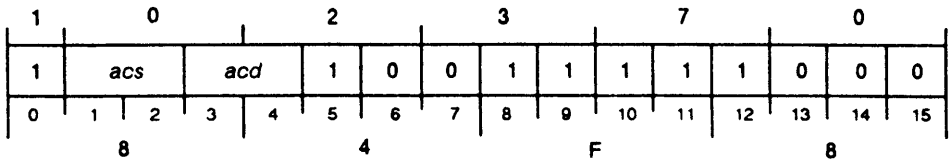
Example

```
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).

- If the integer in *acs* is equal to or between L and H, the processor skips the next sequential instruction.
- If the integer in *acs* is less than L or greater than H, the processor executes the next sequential instruction.

The specification of *acd* determines the location of L and H.

When the location of the **CLM** 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 specification is 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

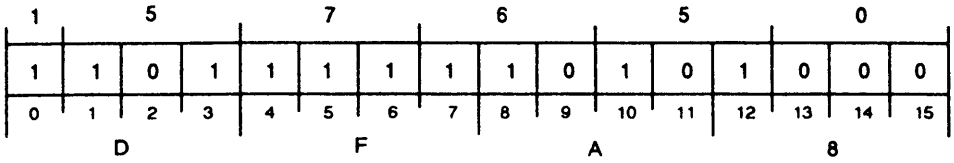
Example

```
    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 +/-1 or failing byte
AC3 = str1 bp → last bp +/-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 either two bytes are not equal, or the specified number of bytes have been compared. **CMP** then 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 can 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, the comparisons continue using space characters $<040_8>$ for comparison with the remaining bytes of the 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 comparison.

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 length of zero, no comparison is made and the result returned to AC1 is 0.

If the specified addresses are outside the user's address space, a protection fault may occur (code 2 returned in AC1), even if no bytes are to be compared. (The protection fault occurs when the offending byte is processed.)

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.

During execution, address wraparound may possibly not occur at 77777_8. In this case, the ECLIPSE program counter can generate logical addresses larger than 64 Kbytes, with undefined results.

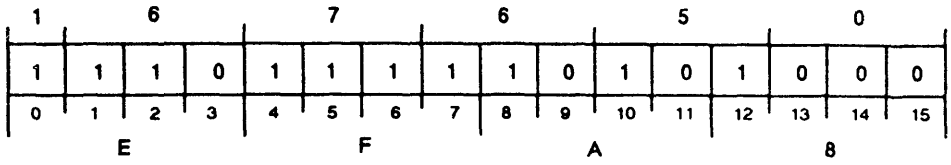
Example

```
    ;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 0 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.." in 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
                JMP     EQ              ;equal.
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 +/-1
AC3 = source byte pointer → last byte pointer +/-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.
- 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 byte 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.

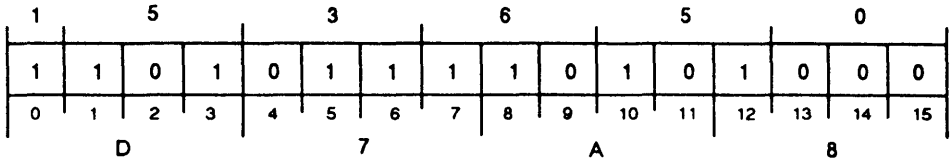
Example

```
    .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
    ;       NL  <012>     NEW LINE
    ;       FF  <014>     FORM FEED
    ;       CR  <015>     CARRIAGE RETURN
    ;       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 destination buffer byte pointer into AC2.
   ELEF 3,2*SRCSTR,0  ;Point to first character of SRCSTR and
   CMT                ;move it until we hit delimiter or 80th character.
   SUB    2,2         ;End of program - Return to caller.
   ?RETURN
    ;       -- DELIMITER TABLE --
            .ENABLE UWORD
            .RDX          2
    ;       NUL       NL FF CR
    ;        /         \ | /
DELIMS:     1000000000101100          ;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 characters!
    ;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. CMV 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, contains 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 byte 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.

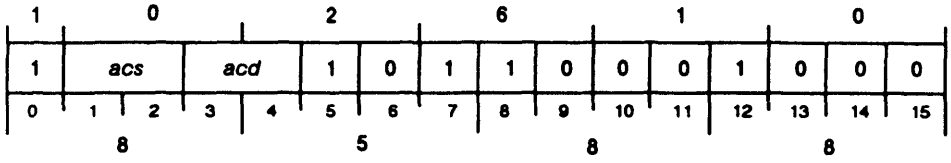
Example

```
    ;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 character 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 *acs* and adds this number to the contents of *acd*.

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

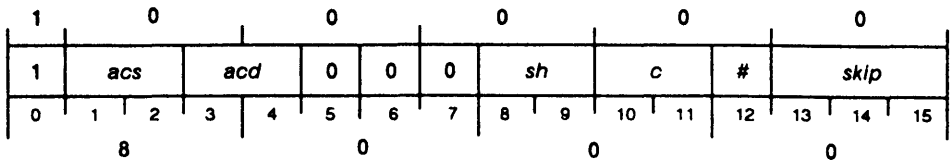
Example

```
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: $\overline{acs} \rightarrow acd$

Parameters: None

COM initializes Carry to the specified value, forms the logical complement of the value in *acs*, and performs any 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*, as indicated by 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*, as indicated by 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, as indicated by 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 indicated 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

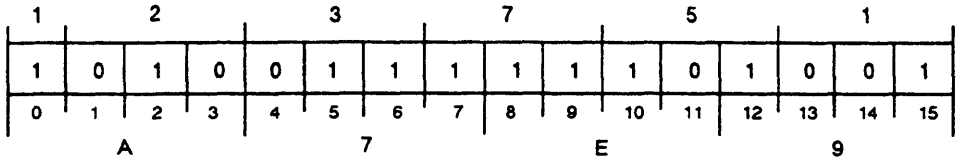
Do not specify **COM** with no-load option (#) in combination with either never skip or always skip option. Instruction may not end in 1000_2 or 1001_2 (reserved for other instructions).

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 0.
```

Complement Carry — CRYTC

CRYTC



Function: $\overline{CRY} \rightarrow CRY$

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

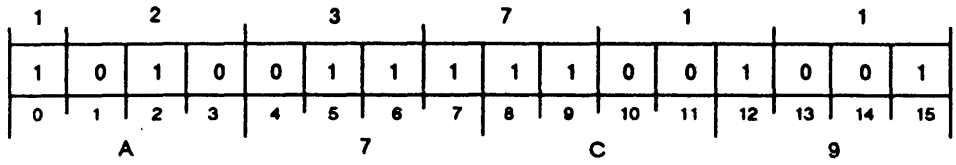
None

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

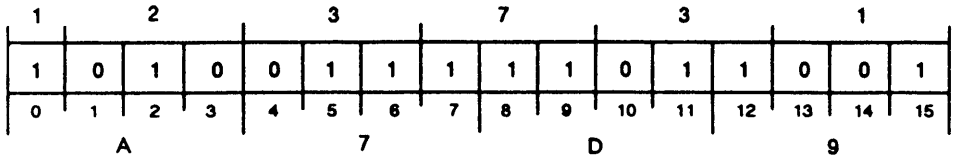
None

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

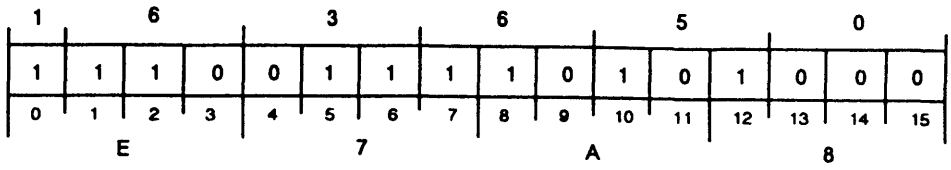
Example

```
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 → resolved address of translation table
AC1 = # bytes [-2#] → 0
AC2 = destination byte pointer → last byte pointer + 1
AC3 = source byte pointer → last byte pointer + 1

-OR-

Function: source1 @(AC3) (translated) ?=? source2 @(AC2)
result code → AC1

Parameters: AC0 = word address of translation table byte pointer → resolved address of translation table
AC1 = # bytes [+#] → result -1 (source1 < source2)
0 (source1 = source2)
+1 (source1 > source2)
AC2 = source2 byte pointer → last byte pointer + 1 or to failing byte
AC3 = source1 byte pointer → last byte pointer + 1 or to failing byte

CTR has two different operating modes: *translate and move*, or *translate and compare*.

For either mode, **CTR** processes one byte at a time from the source string(s), translating the byte from one data representation to another. The instruction performs this translation by using the source byte as an 8-bit index into a 256-byte *translation table*. The byte addressed by the index then becomes the *translated value*.

- In the *translate and move* mode, the translated value is deposited in the next byte position in the destination string. The process continues until the specified number of bytes has been translated and moved.
- In the *translate and compare* mode, the translated value from one source string is compared with the translated value from a second source string. In performing this translation, **CTR** treats the translated values as unsigned 8-bit integers. The process continues until either the specified number of bytes has been translated and compared, or a pair of unequal values is found. The string for which the byte has the smaller numerical value is defined as the lower-valued string. When processing is completed, **CTR** returns a result code in AC1.

In both modes, the strings are processed from the specified starting addresses upward in single-byte increments. The 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.

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 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	source1 byte < source2 byte
0	source1 byte = source2 byte
+1	source1 byte > source2 byte

AC2(16-31) Before execution, contains starting byte address for destination string (or source2). With each byte accessed, address is incremented by 1.

After execution,

for *translate and move* mode, contains address of byte following last byte in string.

for *translate and compare* mode, contains either address of source2 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 (or source1). With each byte accessed, address is incremented by 1.

After execution,

for *translate and move* mode, contains address of byte following last byte in string.

for *translate and compare* mode, contains either address of source1 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 byte 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 performs no operation. (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.

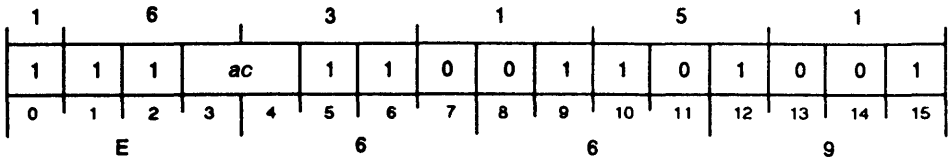
Example

```
    .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 = negative, 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 character 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) ≠ 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 (0–16) 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

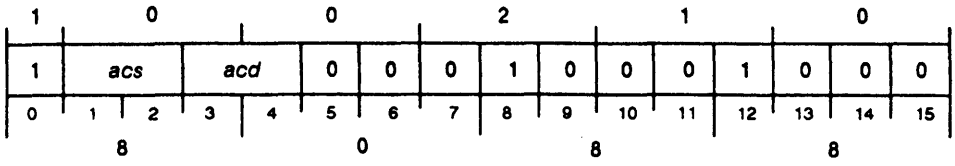
Example

```
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 (bits 0-15 are undefined; bits 16-27 are unchanged).

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.

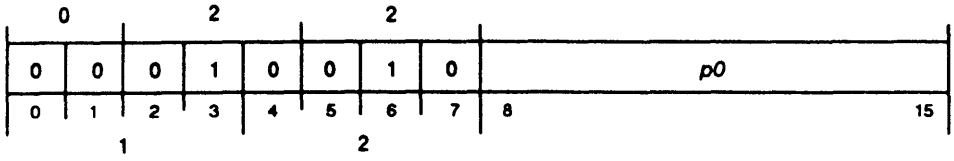
Example

```
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

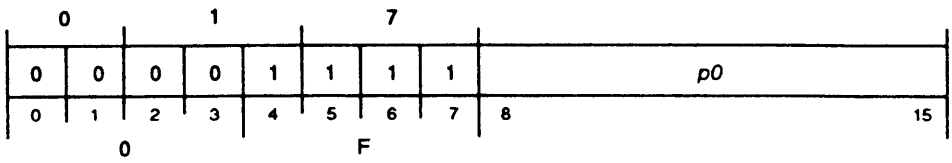
Example

```
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

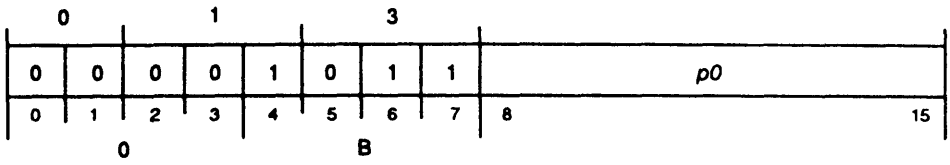
Example

```
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

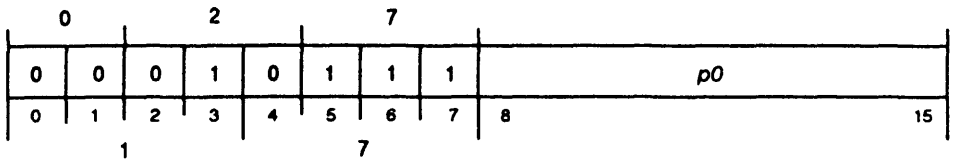
Example

```
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 addition, 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

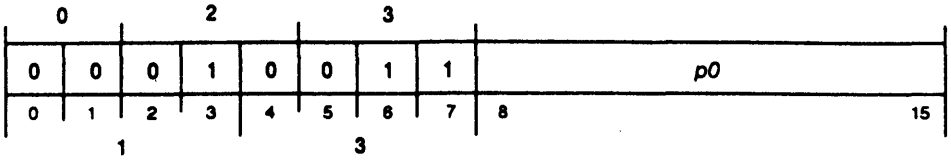
Example

```
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.

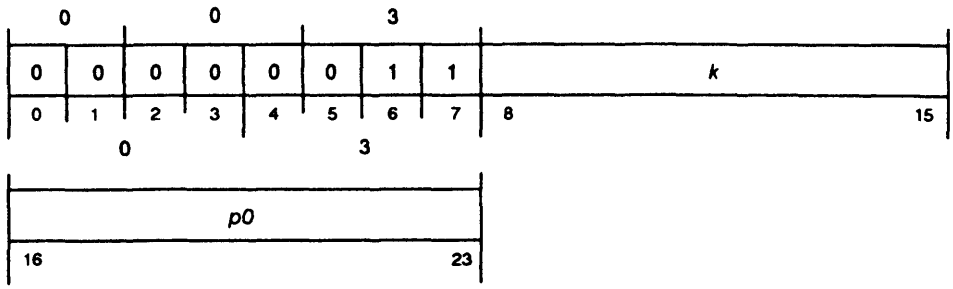
Example

```
DASI -2    ;Decrement SI by 2.
```

Decrement and Jump if Nonzero — DDTK

Edit Subopcode

DDTK *k,p0*



Function: $(k)[\#] - 1 \rightarrow (k)$
If $(k) \neq 0$, 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* is negative, word decremented is at address: narrow stack pointer + 1 + *k*. If *k* is positive, word decremented is at address: narrow frame pointer + 1 + *k*. For **WEDIT**: If integer specified by *k* is negative, doubleword decremented is at address: WSP + 2 + 2**k*. If *k* is positive, doubleword decremented is at address: WFP + 2 + 2**k*.

Related Instructions

DSTK	Store in Stack

Exceptions

None

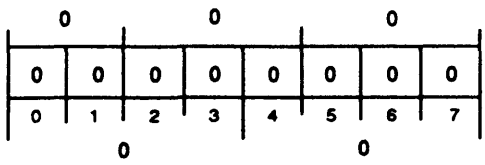
Example

```
WEDIT               ;Call Edit subprogram.
      ...           ;Subprogram.
DDTK -2,2           ;Decrement the doubleword 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 Enter an Edit subprogram.

Exceptions

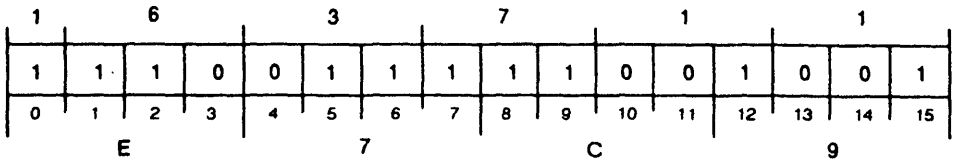
None

Example

```
DEND  ;Terminates Edit subprogram.
```

Dequeue a Queue Data Element — DEQUE

DEQUE
(dequeue last element or from empty queue return)
(dequeue 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 removes a data element from a queue.

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.

The dequeue operation itself is not interruptible. The entire operation completes before any interrupts are enabled. **DEQUE** is indivisible with respect to **ENQT**, **ENQH**, or another **DEQUE** instruction.

The instruction requires up to nine pages to be resident in memory. (The worst case occurs when the element to be removed 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 dequeue 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 remove the data element.

If the **DEQUE** instruction is successful, entry into the next instruction occurs without an intervening interruption.

Arguments

None

Registers, Flags, and Stacks

AC0 — Before execution, contains effective address of queue descriptor.

After execution, contents unchanged.

If removing 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 removing last element in queue, this means both links in the queue descriptor will be set to -1.

AC1 — Before execution, specifies element to be removed.

Instruction removes an element by writing -1 in forward and backward links of the element being removed. The forward link of the element preceding the element being removed is updated to the value in the forward link of the element being removed. Similarly, the backward link

of the element following the element being removed is set to the value of the backward link of the element being removed.

If AC1 does not contain -1, accumulator contains effective address of data element to be removed. If AC1 contains -1, accumulator removes head of queue, obtained from queue descriptor pointed to by effective address in AC0.

After execution, contains address of element removed.

If removing from empty queue, unchanged.

If removing from queue with one or more data elements, AC1 updated with address of removed data element.

AC2-AC3	Unused
Carry	Unchanged
Overflow	0
PC	PC + 1 (removing last data element or deleting from an empty queue)
	PC + 2 (removing from queue with two or more data elements)
PSR	Unchanged
Stack	Unchanged

Related Instructions

ENQH	Use this instruction to add a new data element to a queue before another element.
ENQT	Use this instruction to add a new data element to a queue behind another element.

Exceptions

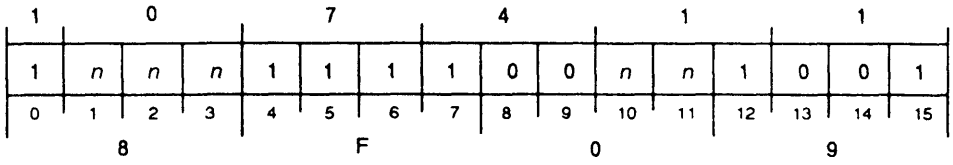
None

Example

```
;This subroutine removes an element from a linked list queue.
;It is the responsibility of the caller to set the transition bit,
;if necessary.
;
;Calling conventions:
        ;    XJSR  PDEQ
        ;    <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 error 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_8	Nonindirectable 16-bit pointer to error handler (in first 64 Kbytes of current segment). If bit 0 of pointer is 1, instruction checks for stack overflow. If bit 0 of pointer is 0, instruction ignores stack overflow.
PC	Address of error handler.
PSR	Unchanged
Stack	Top doubleword of wide stack contains address of **DERR** instruction preceded by doubleword with 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.

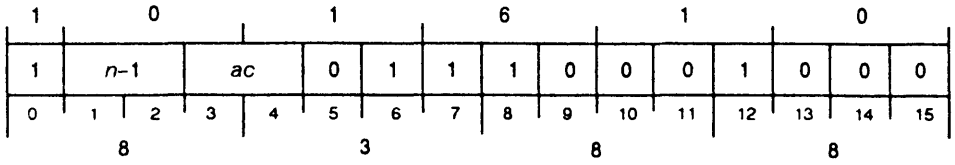
Example

```
...
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. Since Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field, you 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

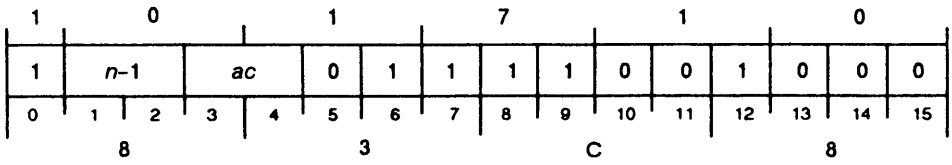
Example

```
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. Since Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field, you 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

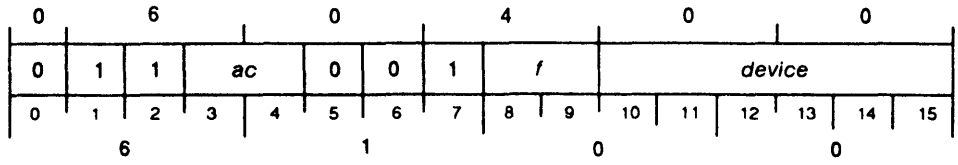
Example

```
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 1	Set to 0
C	Set to 0	Set to 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. Any of bits 16-31 in *ac* 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

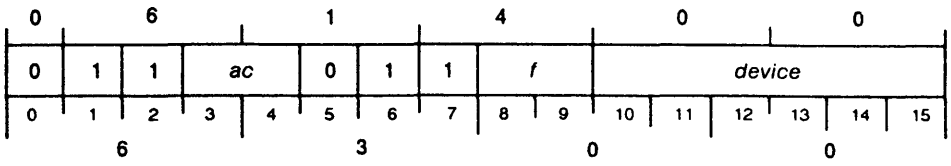
Example

```
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 1	Set to 0
C	Set to 0	Set to 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. Any of bits 16–31 in *ac* 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

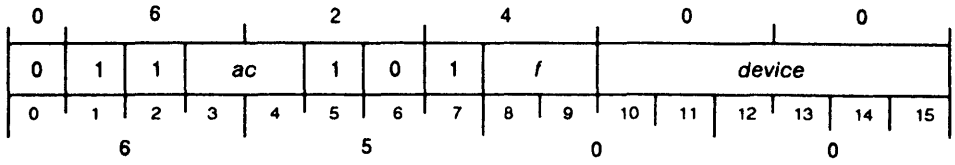
Example

```
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 1	Set to 0
C	Set to 0	Set to 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. Any of bits 16-31 in *ac* 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.

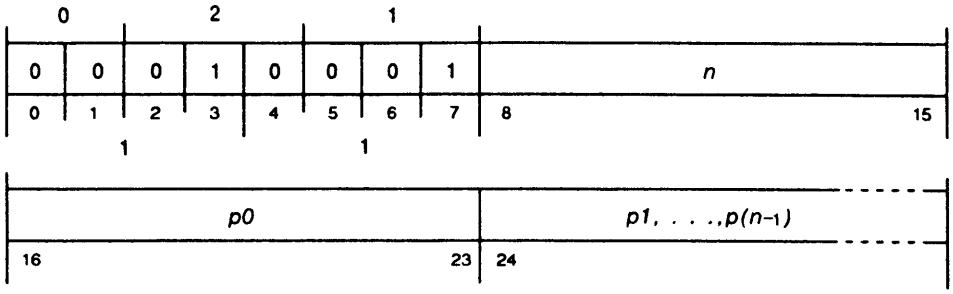
Example

```
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

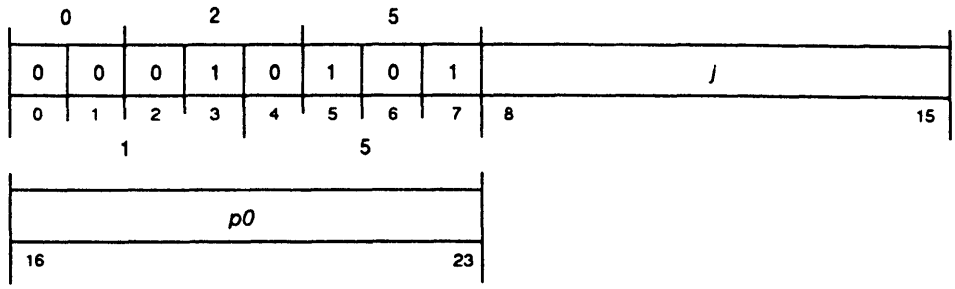
Example

```
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

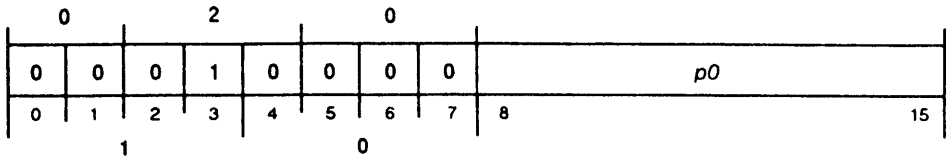
Example

```
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

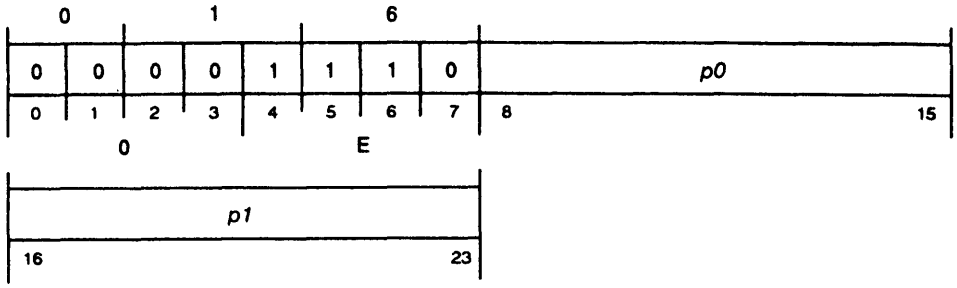
Example

```
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

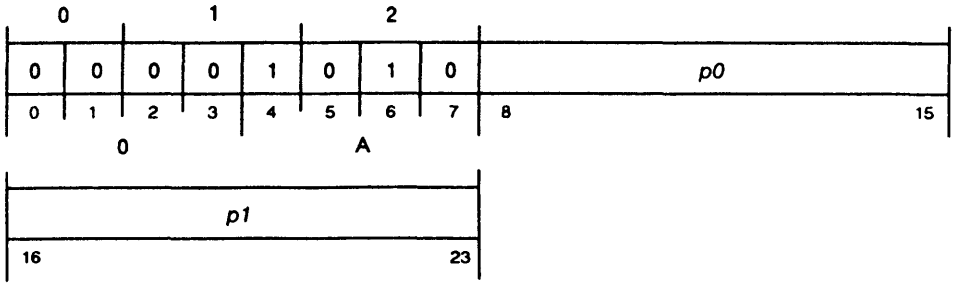
Example

```
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

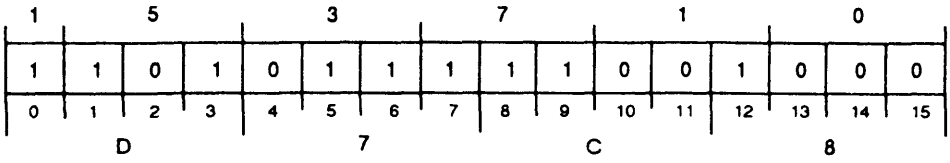
Example

```
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, WDIVS	Perform a signed divide using the contents of accumulators.
DIVX	Sign Extend and Divide
NDIV	Narrow Divide
WDIV	Wide 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.

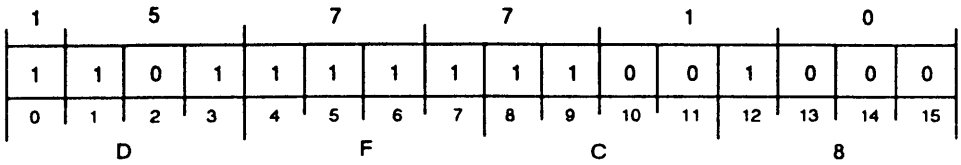
Example

```
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 AC2=0 or quotient overflows:
1 → CRY, AC0&AC1 = undefined, 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 0 quotient or 0 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	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 AC2 initially contains 0 or quotient is outside range of $-32{,}768_{10}$ to $+32{,}767_{10}$, Carry set to 1 and operation terminates. Contents of AC0 and AC1 unpredictable.

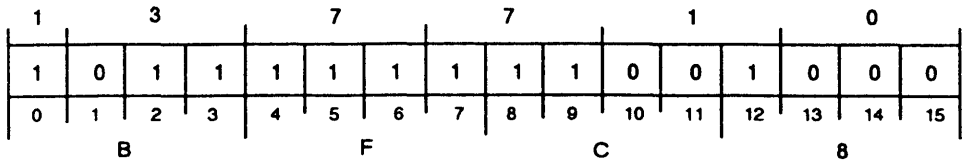
Example

```
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 AC2=0 or quotient overflows:
1 → CRY, AC0&AC1 = undefined, 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 (refer to **DIVS**).

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, WDIVS	Perform a signed divide using the contents of accumulators.
NDIV	Narrow Divide
WDIV	Wide Divide

Exceptions

If AC2 initially contains 0 or the quotient is outside range of $-32{,}768_{10}$ to $+32{,}767_{10}$, Carry is set to 1, the contents of AC0 and AC1 are undefined, and the instruction terminates.

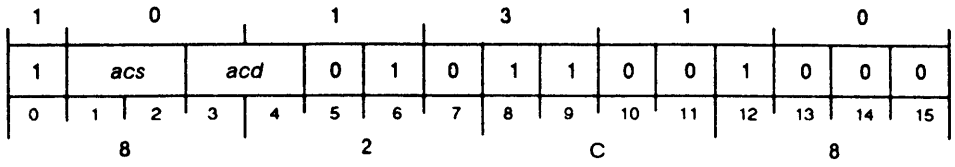
Example

```
DIVX                    ;If the signed integer in AC1 is evenly
ZEX     0,0             ;divisible by the signed integer
WSEQ    0,0             ;in AC2,
        JMP NOTEVEN     ;jump to YESEVEN.
JMP     YESEVEN         ;Otherwise, jump to NOTEVEN.
```

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.

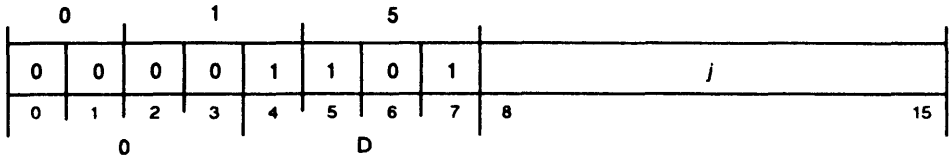
Example

```
DLSH 1,3        ;Shift contents of AC3 and AC0 in the magnitude and
. . .           ;direction indicated by 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 decimal/ASCII fault.

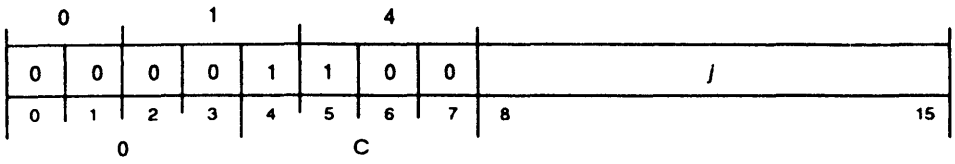
Example

```
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 decimal/ASCII 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 decimal/ASCII fault and performs no other action.

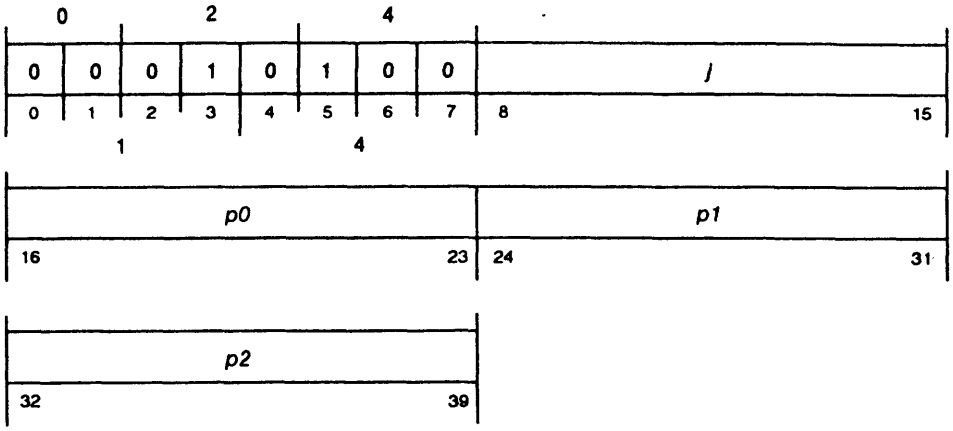
Example

```
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 places the characters into the destination field depending 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 each digit transfer affects SI.
T	Unchanged

Related Instructions

DNDF	End Float

Exceptions

DMVF initiates a decimal/ASCII fault if any of the digits processed is not valid for the specified data type.

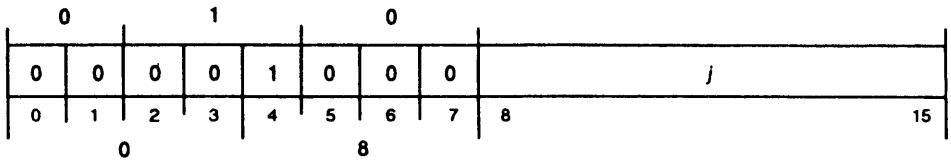
Example

```
;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 each digit transfer affects SI.
T	Set to 1

Related Instructions

DMVS	Move Number with Zero Suppression

Exceptions

DMVN initiates a decimal/ASCII fault if the characters moved are not valid for the specified data type.

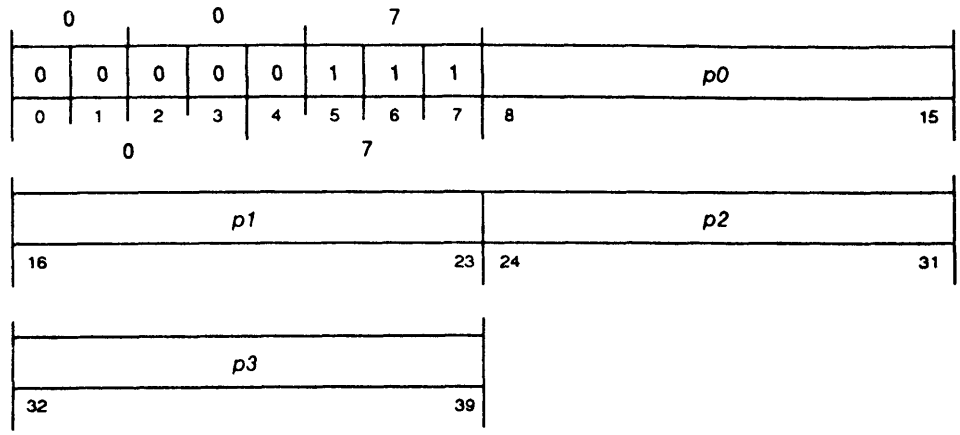
Example

```
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 0 and

S is 0, the translation is equal to *p0*.

S is 1, the translation is equal to *p1*.

If the digit is non-0 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 the digit transfers affect SI.
T	Set to 1 if source digit nonzero; otherwise unchanged.

Related Instructions

DSSO, DSSZ, Set S flag to one or zero.

DSTO, DSTZ Set T flag to one or zero.

Exceptions

DMVO initiates a decimal/ASCII fault if the character is not valid for the specified data type.

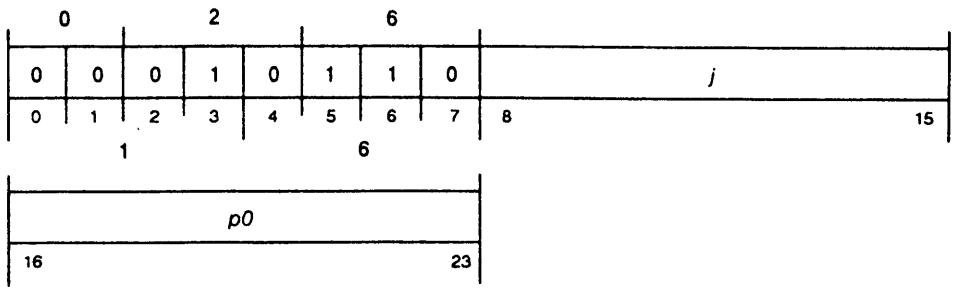
Example

```
;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)
If T = 0, p0[char] → 0s & spaces
SI + j → SI
DI + j → DI
? → 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 each digit transfer affects SI.
T	Undefined

Related Instructions

DMVO	Move Digit with Overpunch

Exceptions

DMVS initiates a decimal/ASCII fault if any of the digits processed is not a numeric (0–9) or a space.

DMVS destroys the data type specifier.

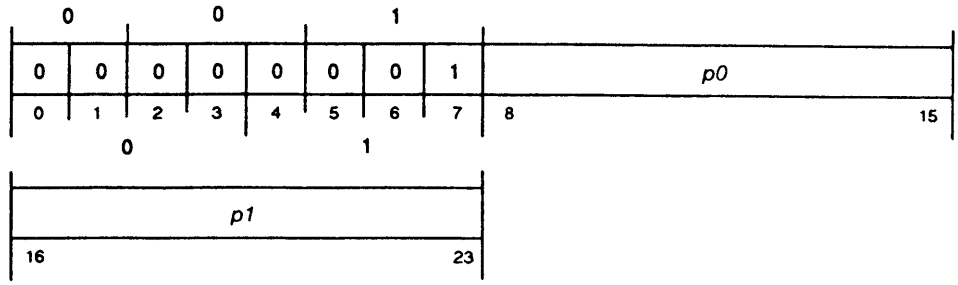
Example

```
DMVS
```

End Float DNDF

Edit Subopcode

DNDF *p0,p1*



Function:
If T = 1,
DI → DI
T → T
If T = 0,
If S = 0, *p0*[char] → (DI)
If S = 1, *p1*[char] → (DI)
1 → T
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 S is

0, the instruction places *p0* in the destination field at the position specified by DI.

1, the instruction places *p1* in the destination field at the position specified by DI.

DNDF then sets T to 1, and increments DI by 1.

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

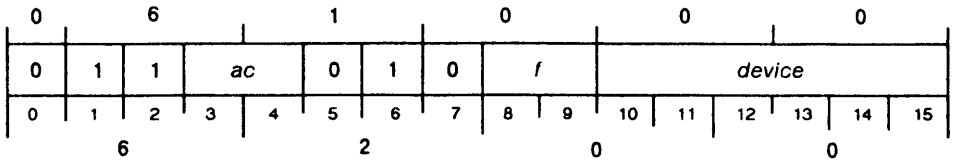
Example

```
;If T is still 0, 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 1	Set to 0
C	Set to 0	Set to 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

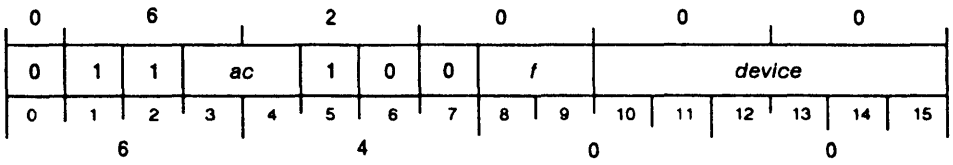
Example

```
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 1	Set to 0
C	Set to 0	Set to 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

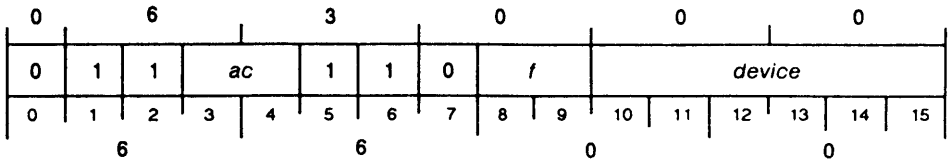
Example

```
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 1	Set to 0
C	Set to 0	Set to 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.

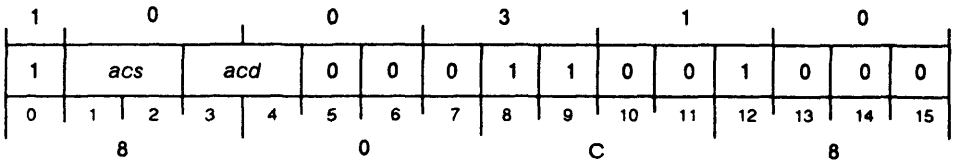
Example

```
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 (bits 0–15 are undefined; bits 16–27 are unchanged).

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, indicating borrow.
If 1, result positive, indicating 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.

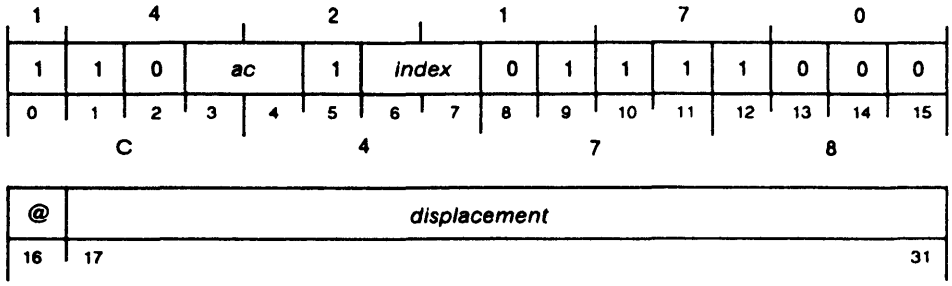
Example

```
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 dispatch table in memory. The effective address resolved from the instruction arguments specifies the starting location of the table. The signed 16-bit integer in *ac* defines the displacement within the table.

Before being used to access the table, the displacement value is compared against two values, an upper limit and a lower limit. These limits define the address range of the table. The two words immediately preceding the table contain the limit values. The upper limit is stored at the table address minus one; the lower limit is stored at the table address minus two.

If the table displacement number is equal to or within the limit values, **DSPA** reads the address from the table; checks it for validity (resolving any indirection); and, if valid, places the effective address in the program counter.

The table address to read is derived as follows:

Table address = **DSPA** effective address - lower limit + displacement number

Note that all values within the table, including the limits, are treated as signed 16-bit integers. Any unused locations within the table should be set to -1.

Arguments

ac(16–31) Before execution, contains signed 16-bit integer; used as the displacement number within 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 check for a valid address, if the dispatch table number is a −1 (177777_8), the instruction terminates and processing continues with the next sequential word. (A −1 value at any point in an indirect chain is treated as another indirection.)

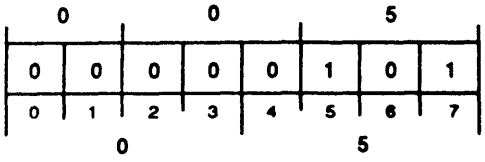
Example

```
    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: $1 \rightarrow S$

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

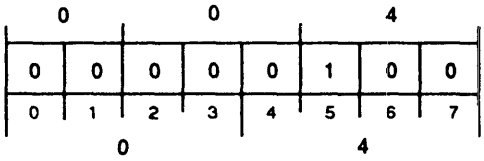
Example

```
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

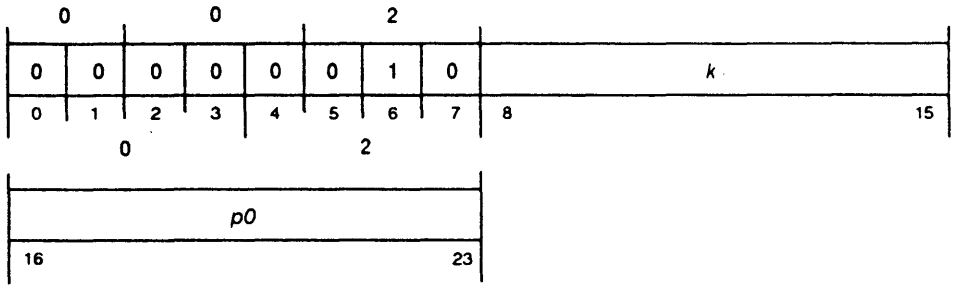
Example

```
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 doubleword in the wide stack, setting bits 0–23 of the stack doubleword 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 doubleword at: WSP + 2 + 2**k* If positive, stack doubleword 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 doubleword(0–23) = 0 stack doubleword(24–31) = *p0*

Related Instructions

DDTK	Decrement and Jump if Nonzero

Exceptions

None

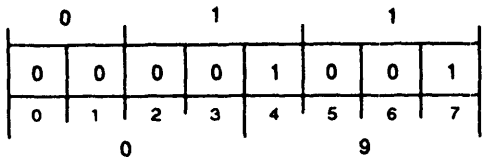
Example

```
WEDIT               ;Enter Edit subprogram.
...
DDTK  2,T           ;Stores "T" into doubleword in stack at
                    ;address of 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

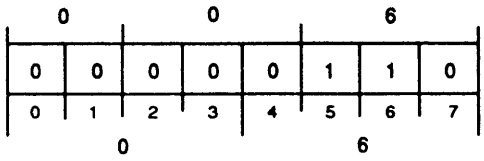
Example

```
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

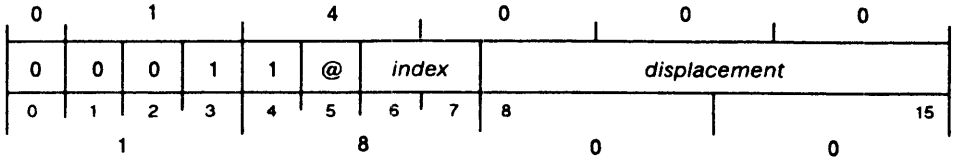
Example

```
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 and writes the result back into the location. If the result written to memory is 0, the instruction then skips the next sequential 16-bit word. 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

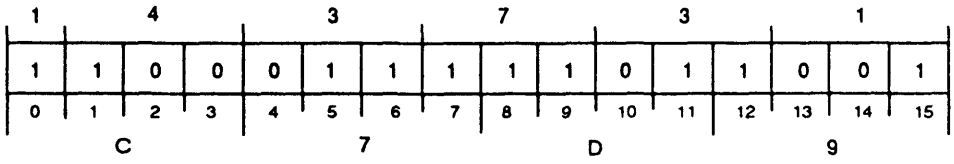
None

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 0. 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

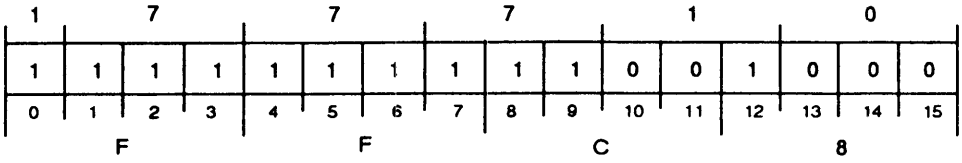
None

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 — After execution, contains processor identification information as follows:

Bits	Contents
0-15	Model number (binary value of processor's allocated model number).
16-23	Current microcode revision.
24-31	Memory size (amount of physical memory available, measured in 256-Kbyte modules with an origin of 0). Note that the actual memory size in bytes is equal to (AC0(24-31) + 1) * 262144_{10} For example, 3_8 indicates 1 Mbyte; 7_8 indicates 2 Mbytes.

NOTE: *Systems which contain 64 Mbytes or more of physical memory return 377_8 to bits 24-31 of AC0.*

AC1-AC3 — Unused

Carry — Unchanged

Overflow — 0

PC — PC + 1

PSR — Unchanged

Stack — Unchanged

Related Instructions

NCLID, LCPID Return CPU identification information. (Use the **NCLID** instruction on systems with more than 64 Mbytes of physical memory.)

Exceptions

None

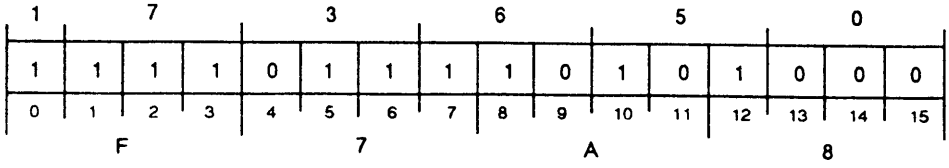
Example

```
    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 flag (T), and the Sign flag (S); the 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 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 certain types of overlap may produce unusual side effects.

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 Edit subopcode descriptions, 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:

- 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.
- 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.
- 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.
- 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 the section, "Decimal and Byte Operations," of the chapter, "Fixed-Point Computing." 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

EDIT considers the subprogram as data and does not check for execute protection.

If **EDIT** is interrupted, the processor places restart information on the narrow stack, sets PSR(IRES) to 1, and places a value of 177777_8 in AC0.

If the sign of the source integer is invalid, a decimal/ASCII fault occurs, and the **EDIT** subprogram terminates.

If the data type indicator in AC1 specifies that the source integer is data type 6 or 7, a decimal/ASCII fault occurs, and the instruction terminates.

See also the exceptions for the individual subopcodes.

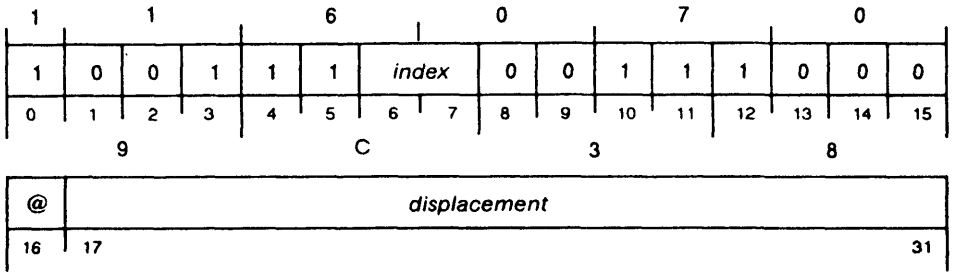
Example

Refer to the programming example for the **WEDIT** instruction (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 by 1 the unsigned 16-bit integer in the specified memory location. The instruction then 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 ≠ 0) PC + 2 (result = 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

DSZ, XNDSZ, XWDSZ, LNDSZ, LWDSZ
Decrement contents of memory location by 1 and skip if result is 0.

Exceptions

None

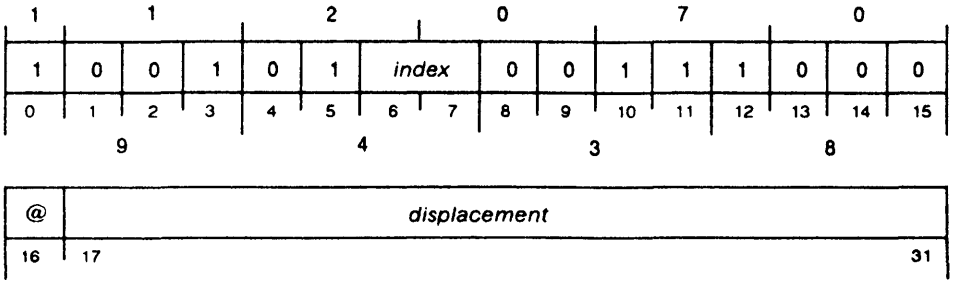
Example

```
    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 by 1 the unsigned 16-bit integer in the specified memory location. The instruction then 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 ≠ 0) PC + 2 (result = 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

ISZ, XNISZ, XWISZ, LNISZ, LWISZ
Increment the contents of memory location by 1 and skip if result is 0.

Exceptions

None

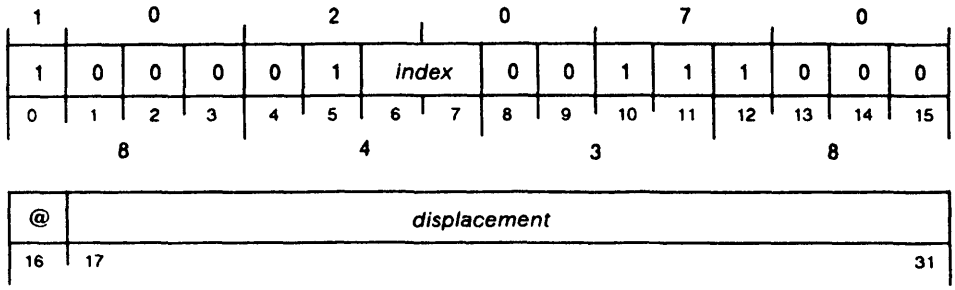
Example

```
    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 an effective 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

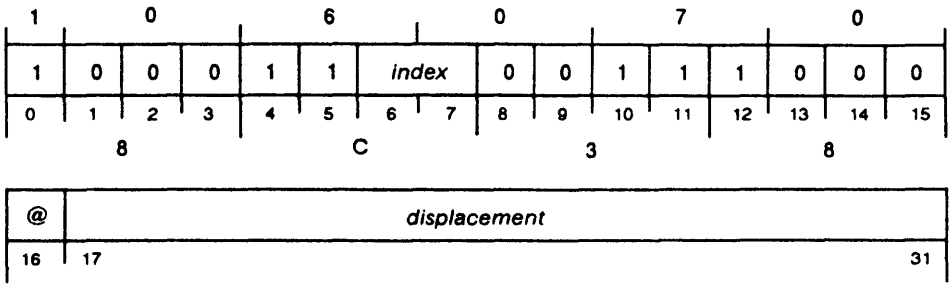
Example

```
    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 program counter 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

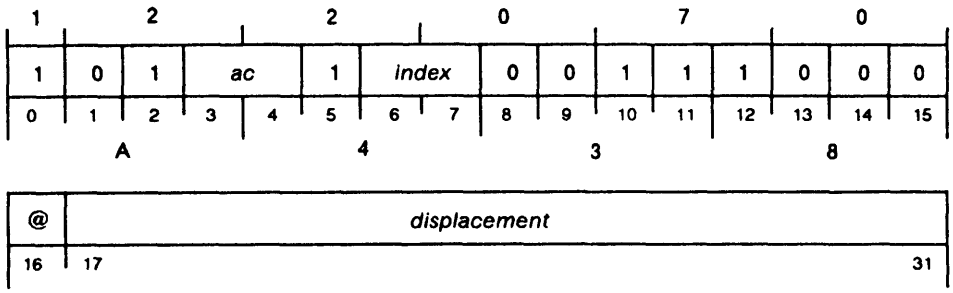
Example

```
    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 caller. ACs are not necessarily restored.
```

Extended Load Accumulator ELDA

ECLIPSE Instruction

ELDA *ac*,[@]*displacement*[,*index*]



Function: (E) → *ac*

Parameters: None

ELDA loads the contents of a memory word 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

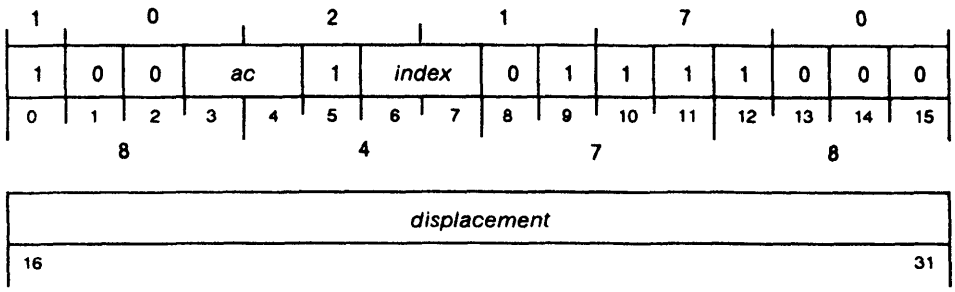
Example

```
    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 (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 memory.

Exceptions

None

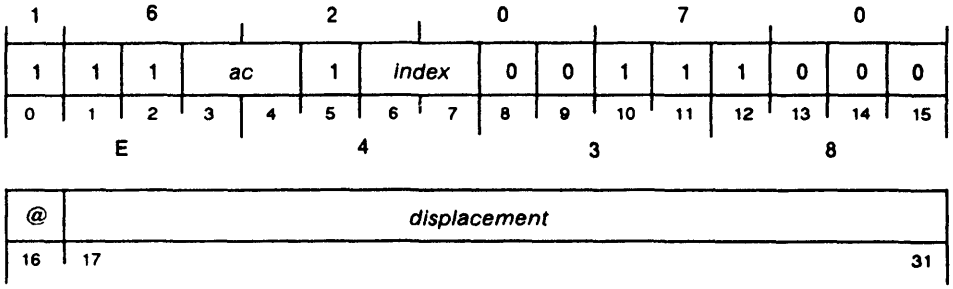
Example

```
    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

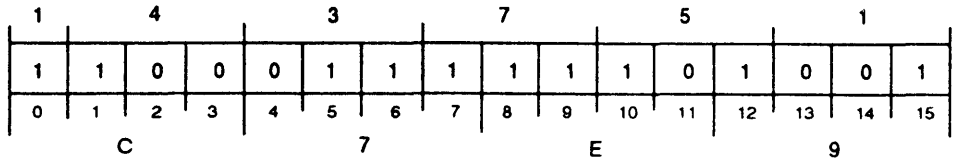
None

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 adds a data element to the queue, 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 enqueueing 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 itself 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 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 enqueueing 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 add the data element to the queue.

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. (When adding an element to an empy queue, **ENQH** ignores the contents of AC1.)

If AC1 contains -1, new element is added to head of queue.

Instruction adds an element by updating the various links of the involved elements:

The element to be added — 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.

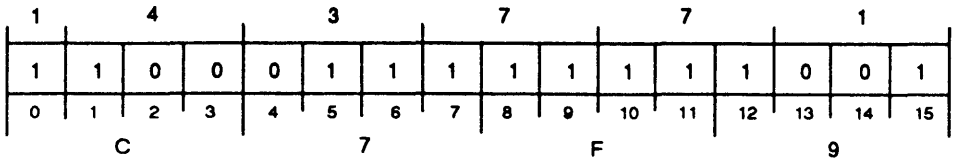
If the ring numbers of the link addresses are less than the current ring, the appropriate protection fault occurs.

Example

```
XLEF 0,QUEDES    ;Load into AC0 effective address of queue descriptor.
WLDAI -1,1       ;Load AC1 with -1.
XLEF 2,NEWDAT    ;Load into AC2 effective address of new data element.
ENQH             ;Add 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 adds a data element to the queue, 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 enqueueing 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 itself 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 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 enqueueing 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 add the data element to the queue.

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. (When adding an element to an empy queue, **ENQT** ignores the contents of AC1.)

If AC1 contains −1, new element is added to tail of queue.

Instruction adds an element by updating the various links of the involved elements:

The element to be added — 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.

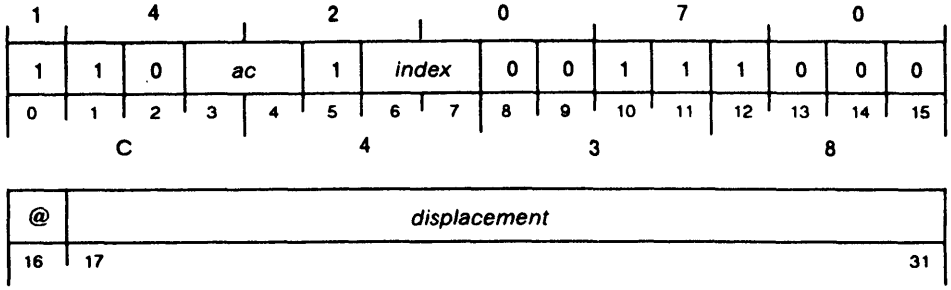
Example

```
    ;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 removed
        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 the specified accumulator into a memory word.

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

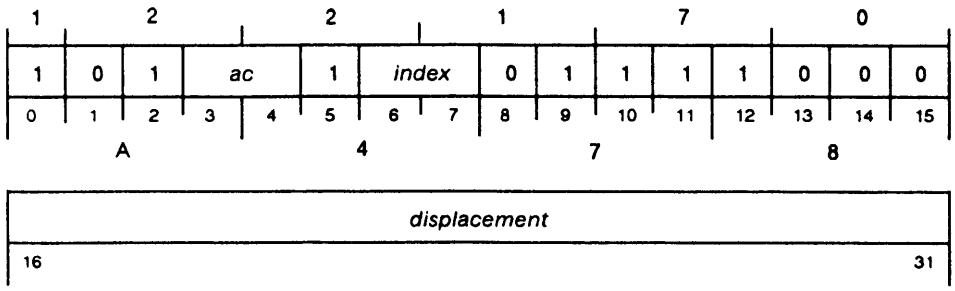
Example

```
    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 memory.

Exceptions

None

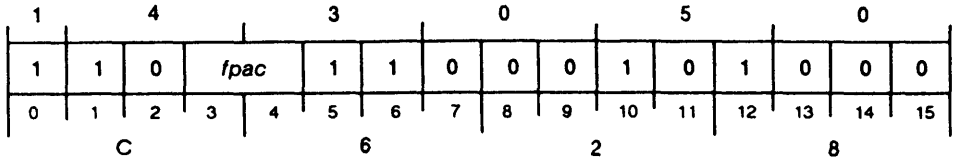
Example

```
    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

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

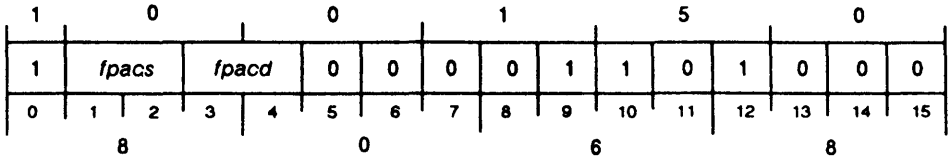
Example

```
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 double-precision floating-point number in *fpacs* to the double-precision 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 OVF flag in the FPSR to one and terminates the instruction.

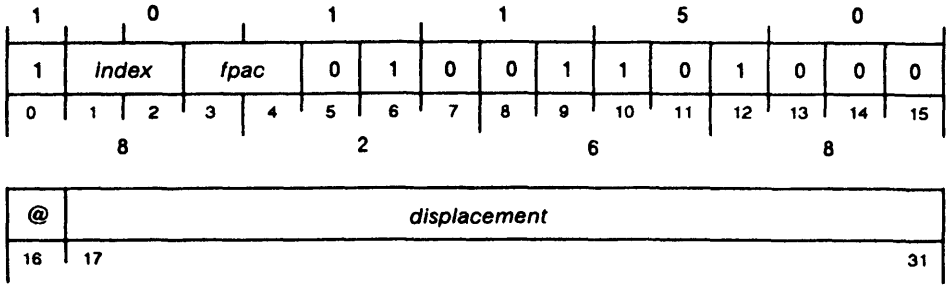
Example

```
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 double-precision floating-point number in memory to the double-precision 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.
PC	PC + 2
FPSR	Z and N flags updated.
Stack	Unchanged

Related Instructions

XFAMD, LFAMD, FAMS, XFAMS, LFAMS
Add the contents of memory to a floating-point accumulator.

Exceptions

If the addition produces an exponent overflow, the processor sets the OVF flag in the FPSR to one, and terminates the instruction.

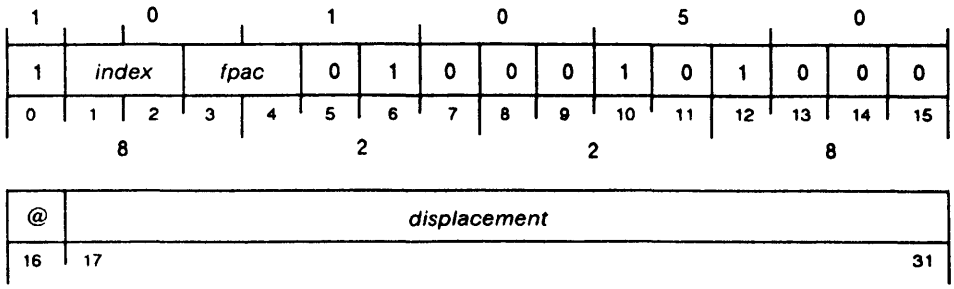
Example

```
FLDD   0,FLOATX     ;Add the two double-precision floating-point
FAMD   0,FLOATY     ;numbers in memory locations FLOATX and FLOATY,
FSTD   0,FLOATZ     ;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 single-precision floating-point number from memory to the single-precision 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 (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.
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 OVF flag in the FPSR to one, and terminates the instruction.

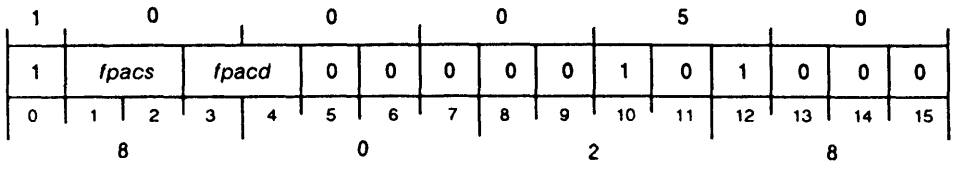
Example

```
FLDS   1,FLOATX     ;Add the two single-precision floating-point
FAMS   1,FLOATY     ;numbers in memory locations FLOATX and FLOATY,
FSTS   1,FLOATZ     ;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 single-precision floating-point number in *fpacs* to the single-precision 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 (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 FPSR(OVF) to 1, and terminates the instruction.

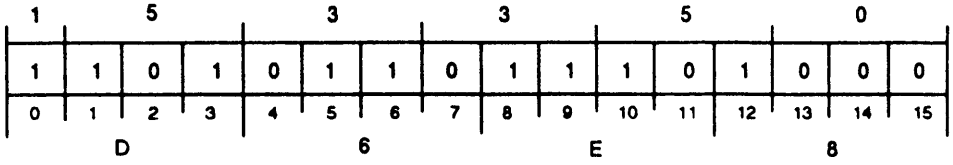
Example

```
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

FPAC0-FPAC3	Unused.
FPSR	Bits 0-4 and bits 28-31 set to 0; bits 33-63 undefined.
PC	PC + 1
Stack	Unchanged

Related Instructions

IORST	I/O Reset

Exceptions

None

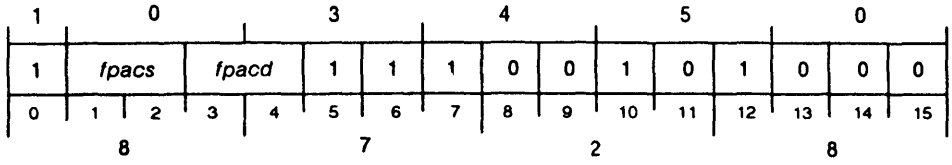
Example

```
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 updated Z and N flags reflecting result of comparison:

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

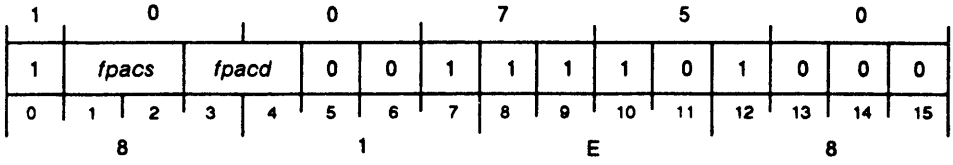
Example

```
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 double-precision floating-point number in *fpacd* by the double-precision floating-point number in *fpacs*, placing the normalized 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 0, 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.

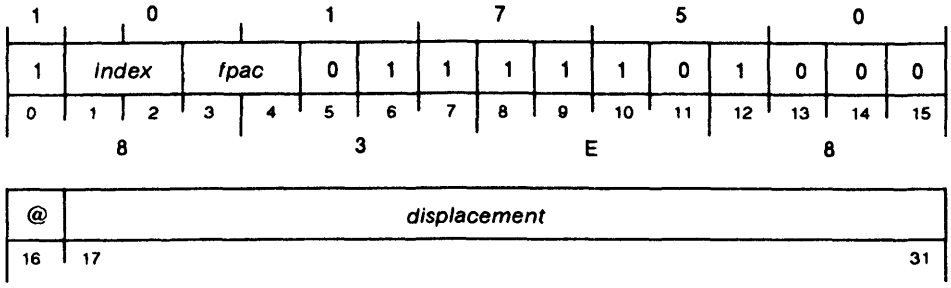
Example

```
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 double-precision floating-point number in a floating-point accumulator by a double-precision floating-point number from memory and places the normalized 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.

PC PC + 2

FPSR Updated Z and N flags.

Stack Unchanged

Related Instructions

XFDMD, LFDMD, FDMS, XFDMS, LFDMS
Divide a floating-point accumulator by the contents of memory.

Exceptions

If the divisor (in memory) is 0, 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.

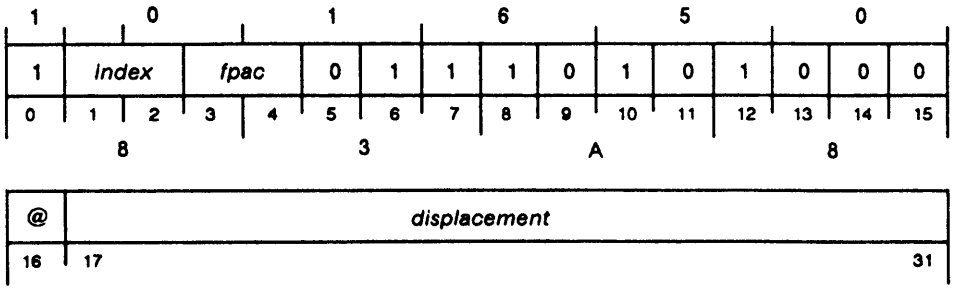
Example

```
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 single-precision floating-point number in a floating-point accumulator by a single-precision floating-point number from memory 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 (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.

PC PC + 2

FPSR Updated Z and N flags.

Stack Unchanged

Related Instructions

XFDMS, LFDMS, FDMD, XFDMD, LFDMD
Divide a floating-point accumulator by the contents of memory.

Exceptions

If the divisor (in memory) is 0, 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.

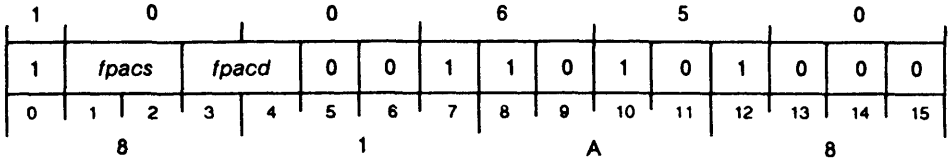
Example

```
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 single-precision floating-point number in *fpacd* by the single-precision floating-point number in *fpacs* and places the normalized 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 (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 0, the processor sets FPSR(INV) to 1, places error code 0 in FPSR(INP) and the address of the instruction in FPSR(FPPC), and terminates the instruction.

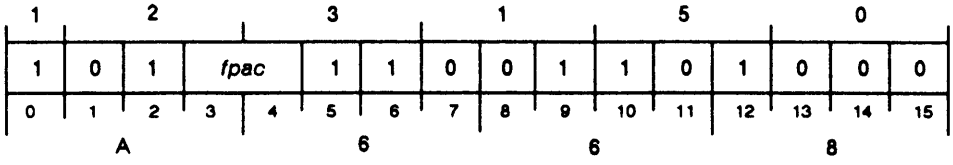
Example

```
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

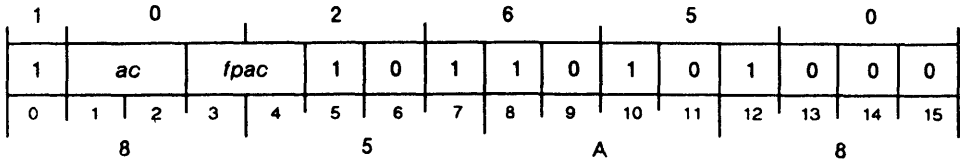
Example

```
XWLDA    0,NEWEXP        ;Use bits 17-23 of the doubleword 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 double-precision 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 0 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, **FFAS** sets FPSR(MOF) to 1.

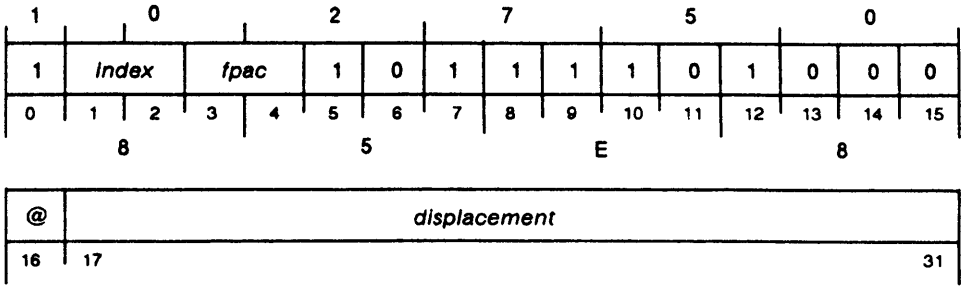
Example

```
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 double-precision floating-point number in *fpac* to a signed 32-bit integer and places the result in memory.

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 memory.

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 significant bits of this absolute value and appends a 0 to 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 memory.

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	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.

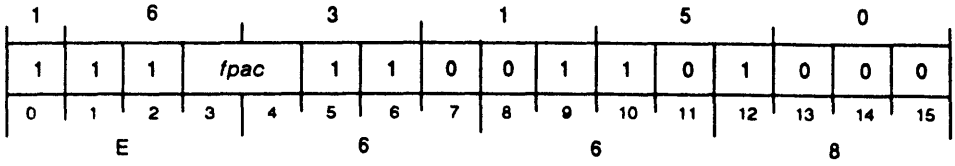
Example

```
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 double-precision floating-point number in *fpac* by two.

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.

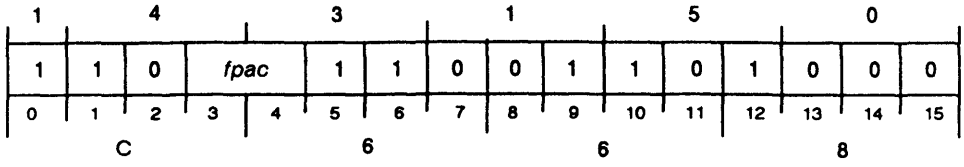
Example

```
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 one, the specified *fpac* is set to true zero.

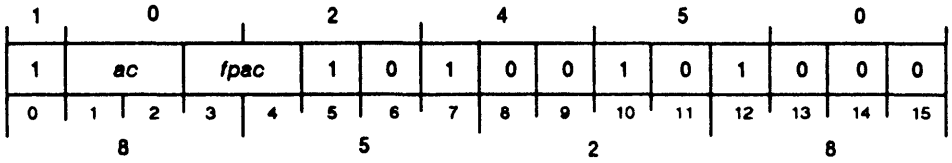
Example

```
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 (bits 32-63 set to 0).

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	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

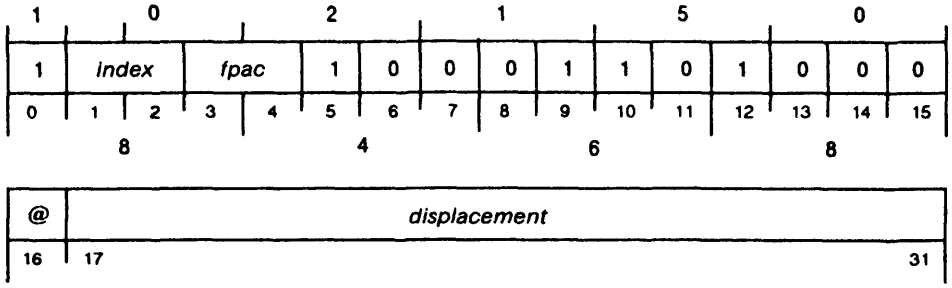
Example

```
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 double-precision floating-point number from memory 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.

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.

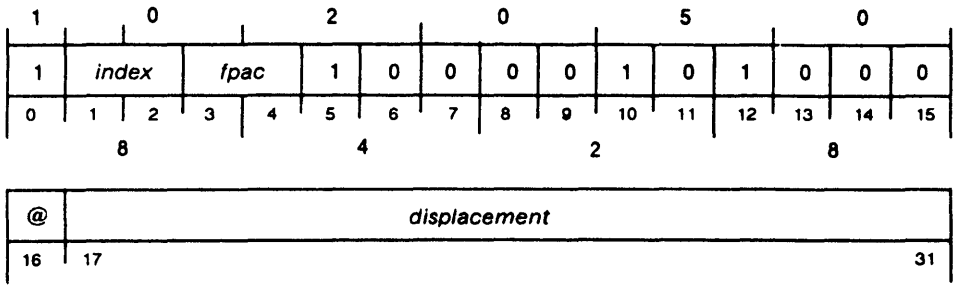
Example

```
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 single-precision floating-point number from memory into the specified *fpac*. Unnormalized data is moved without change.

Arguments

fpac(0-31) After execution, contains 32-bit floating-point number (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.

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.

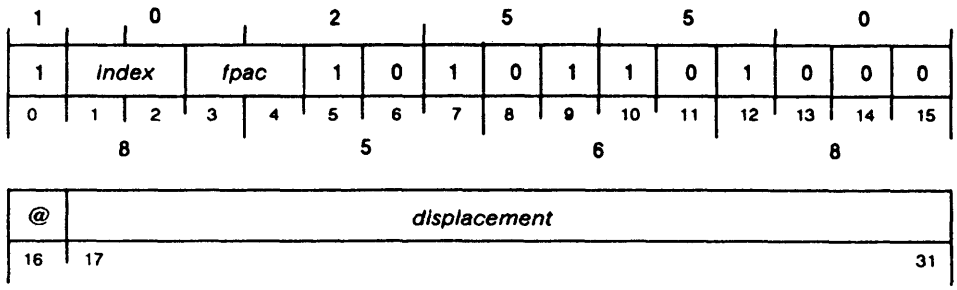
Example

```
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 double-precision 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

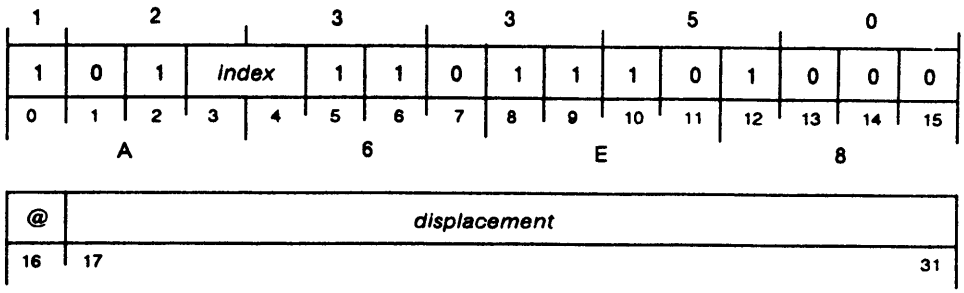
Example

```
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(1-8) receive memory(1-8).

FPSR(9-11) must be loaded as zeros.

FPSR(12-15) are not set from memory. These bits contain the floating-point identification code, are loaded by the processor, and cannot be changed.

FPSR(16-22) are not loaded from memory. The processor sets these bits to 0.

FPSR(23-32) are set to 0.

FPSR(33-63) are set according to the state of the ANY flag:

If ANY = 0, FPSR(33-63) are undefined

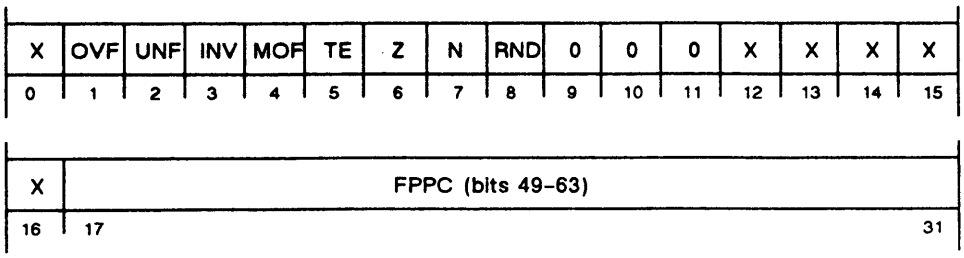
If ANY = 1,

FPSR(33-35) receive the value of the current segment

FPSR(36-48) receive zeros

FPSR(49-63) receive memory(17-31)

The contents of the doubleword in memory (as they correspond to the bits in the FPSR) are as follows:



X = Processor ignores this bit

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 floating-point status register ANY and TE flags are both 1 after the FPPC is loaded, the processor jumps to a floating-point fault handler routine.

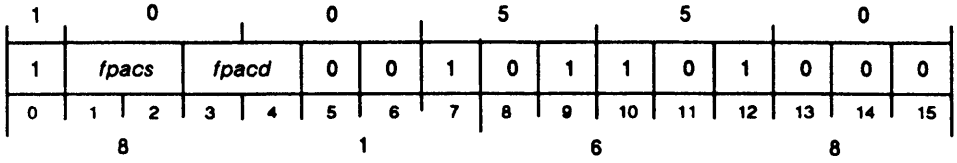
Example

```
FLST   STATUS        ;Load the narrow FPSR from the doubleword
                     ;at memory location STATUS.
```

Multiply Double (FPAC by FPAC) FMD

ECLIPSE Instruction

FMD *fpacs,fpacd*



Function: *fpacd* * *fpacs* → *fpacd*

Parameters: None

FMD multiplies the double-precision floating-point number in *fpacd* by the double-precision floating-point number in *fpacs* and places the normalized 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.

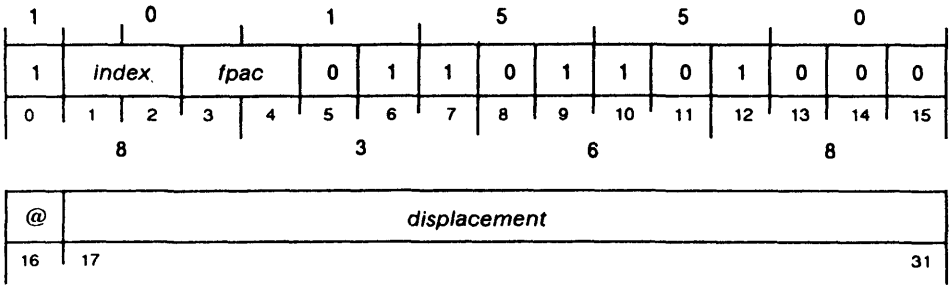
Example

```
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 double-precision floating-point number from memory by the double-precision floating-point number in the *fpac* and places the normalized 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.
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 FPSR(OVF) to 1, and terminates the instruction.

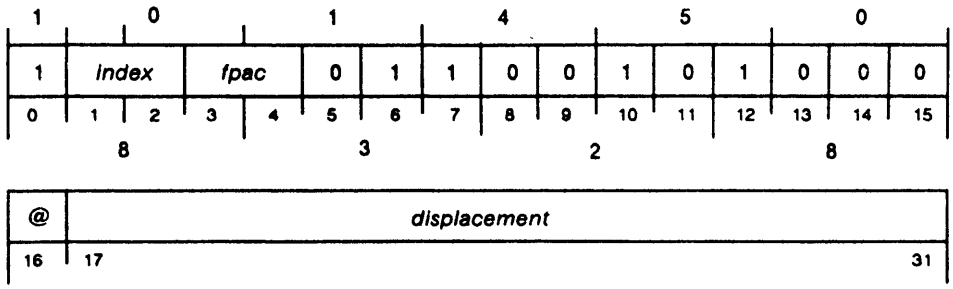
Example

```
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 single-precision floating-point number from memory by the single-precision floating-point number in *fpac* and places the normalized result in *fpac*.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contains normalized 32-bit result (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.

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 FPSR(OVF) to 1 and terminates the instruction.

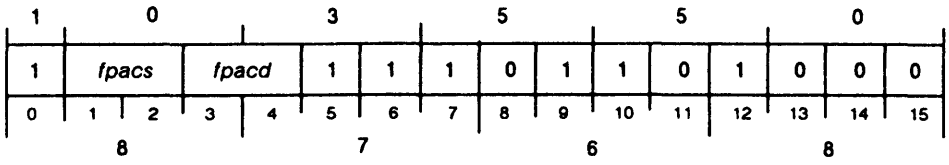
Example

```
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* into *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	After execution, contains *fpacs*.

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.

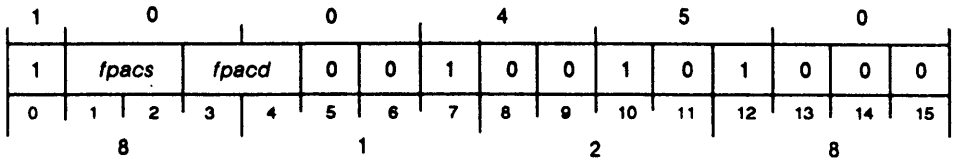
Example

```
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 single-precision floating-point number in *fpacs* by the single-precision floating-point number in *fpacd* and places the normalized 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 (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 one and terminates the instruction.

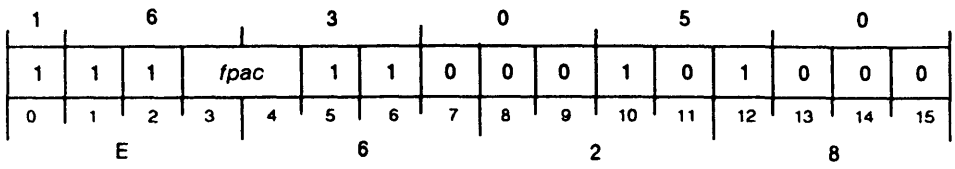
Example

```
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 0) 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

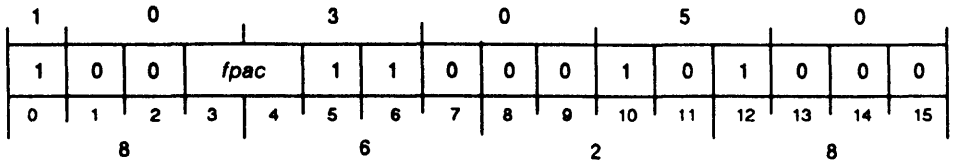
Example

```
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 floating-point accumulator. 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 one.

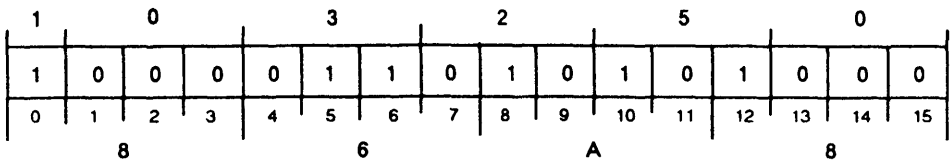
Example

```
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

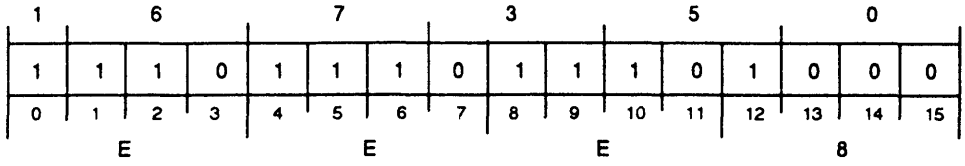
Example

```
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

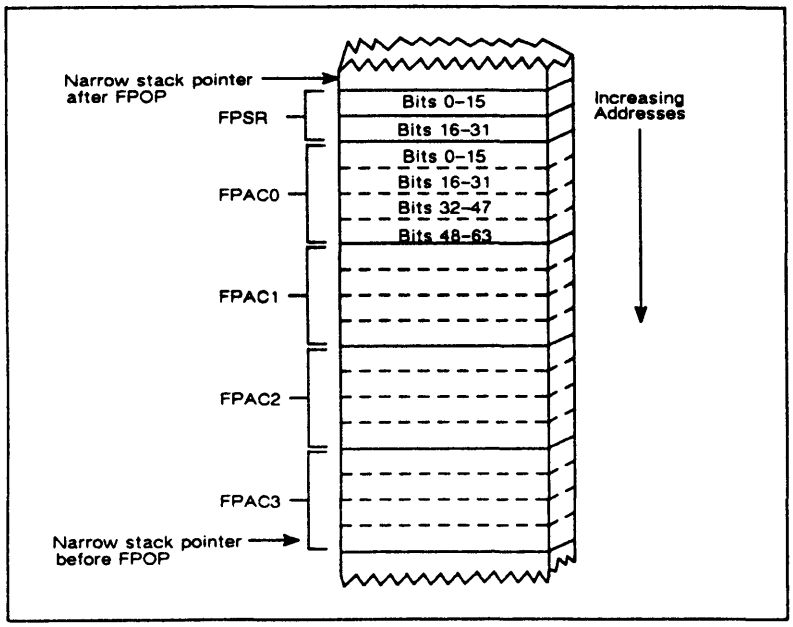


Function: 18 stack words → registers
1st four words → FPAC3
2nd four words → FPAC2
3rd four words → FPAC1
4th four words → FPAC0
last 2 words → FPSR

Parameters: None

NOTE: Certain FPSR bits 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 (FPSR). Unnormalized data is moved without change. The format of the 18-word block is as follows:



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 (bits are in parentheses):

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(1-8) receive memory(1-8).

FPSR(9-11) must be loaded as zeros.

FPSR(12-15) are not set from memory. These bits contain the floating-point identification code, are set by the processor, and cannot be changed.

FPSR(16–22) are not loaded from memory. The processor sets these bits to 0.

FPSR(23–32) are set to 0.

FPSR(33–63) are set according to the state of the ANY flag:

If ANY = 0, FPSR(33–63) are undefined.

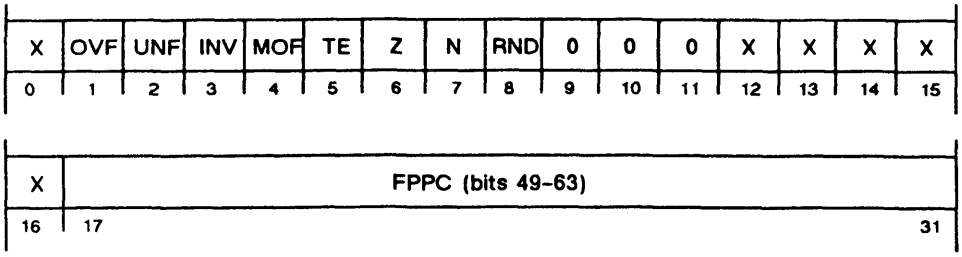
If ANY = 1,

FPSR(33–35) receive the value of the current segment,

FPSR(36–48) receive zeros,

FPSR(49–63) receive memory(17–31).

The contents of the stack words (as they correspond to the bits in the FPSR) are



X = Processor ignores this bit

Arguments

None

Registers, Flags, and Stacks

FPAC0–FPAC3	After execution, contain data from narrow stack.
PC	PC + 1
FPSR	After execution, contains data from narrow stack.
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

FPOP initiates a floating-point trap if FPSR(ANY) and FPSR(TE) are both 1 after FPSR(FPPC) is loaded.

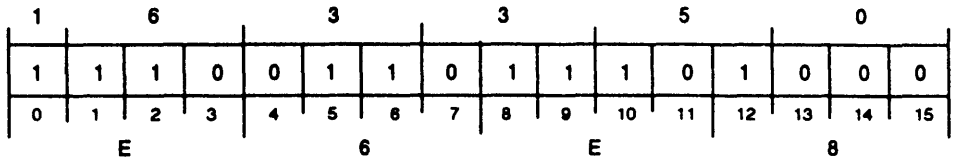
Example

```
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.
```

Push Floating-Point State FPSH

ECLIPSE Instruction

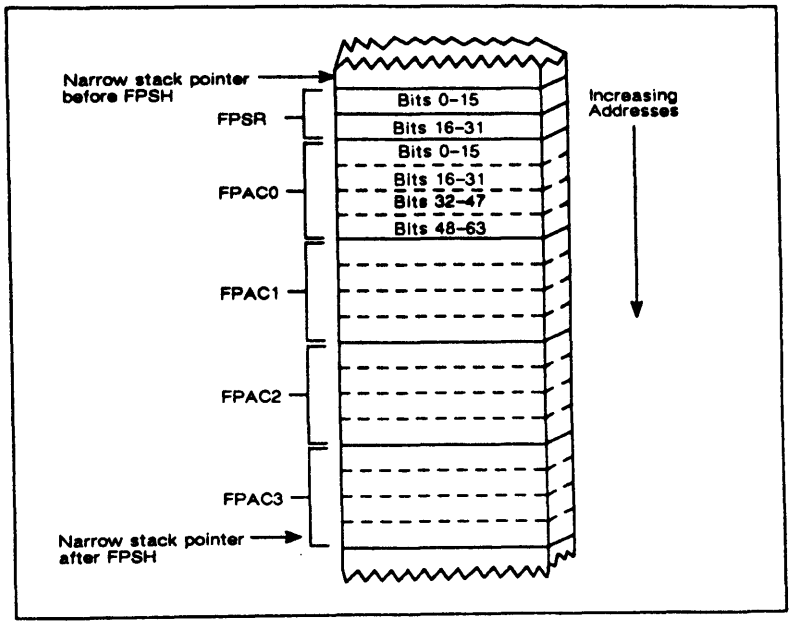
FPSH



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. Unnormalized data is moved without change. The format of the 18 words pushed is as follows:



The first two words pushed onto the stack, a 32-bit operand, are taken from the floating-point status register as follows (bits are in parentheses):

FPSR(0-15) are written to memory(0-15).

The contents of the second stack word depend on the state of the ANY flag:

If ANY = 0, memory(16-31) is undefined.

If ANY = 1, FPSR(49-63) are written to memory(17-31) — memory(16) is set to 0.

The next 16 words pushed on the stack are taken from the four floating-point accumulators.

Arguments

None

Registers, Flags, and Stacks

FPAC0-FPAC3	Provide data to the stack. After execution, contents unchanged.
PC	PC + 1
FPSR	Provides data to the stack. After execution, contents 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 an FPSR value with any combination of bit 5 (TE) and bits 1-4 concurrently set.

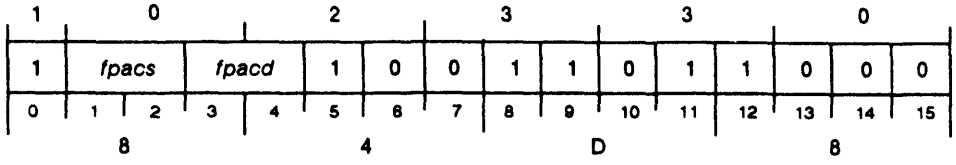
Example

```
FPSH              ;Save all FPACs, so WFSIND won't destroy
LFLDD    0,FDATA1 ;them. We want to take the sine of FDATA1.
WFSIND   SINE     ;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 single-precision result by normalizing the mantissa of the result and appending the *fpacs* sign and exponent (bits 0-7). Then it places the result in *fpacd*.

Arguments

fpacs(8-31) Before execution, if FPSR(RND) = 1, contains unrounded mantissa.

After execution, contents unchanged unless *fpacs* and *fpacd* are same accumulator.

fpacs(32-63) Before execution, if FPSR(RND) = 1, contains rounding digits in 3 ranges:

0 to $7FFFFFFF_{16}$ inclusive. Manitssa of result equals unrounded inital mantissa without change.

80000000_{16}. Manitssa of result formed by adding least significant bit of unrounded mantissa to unrounded mantissa.

80000001_{16} to $FFFFFFFF_{16}$ inclusive. Mantissa of result equals unrounded mantissa plus 1.

After execution, contents unchanged unless *fpacs* and *fpacd* are same accumulator.

fpacd(0-31) After execution:

If FPSR(RND) = 0, contains *fpacs*.

If FPSR(RND) = 1, contains rounded bits from *fpacs*(0-31).

Bits 32-63 are set to 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

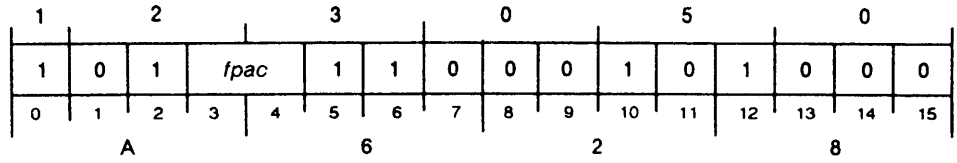
Example

```
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 the floating-point accumulator 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
Carry	Unchanged
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

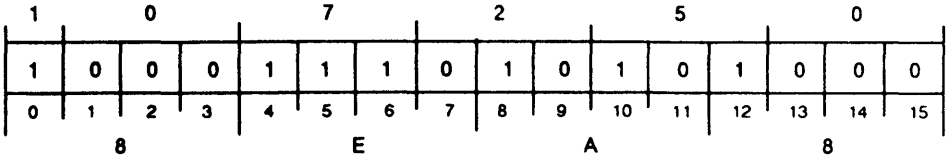
Example

```
FAS   0,1          ;Add FPAC0 to FPAC1, and store the result in AC0.
FRH   1
```

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

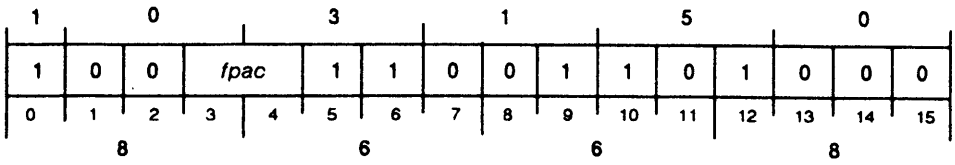
Example

```
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.

The instruction 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 64-bit result.

Registers, Flags, and Stacks

AC0(17-23)	Before execution, contains 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

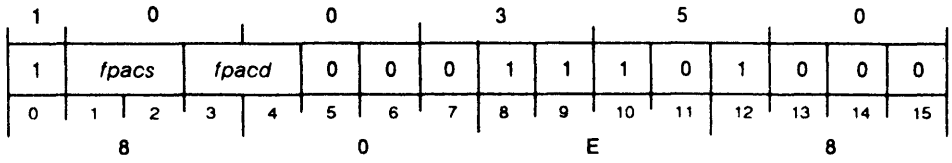
Example

```
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 double-precision floating-point number in *fpacs* from the double-precision floating-point number in *fpacd* and places the normalized 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.

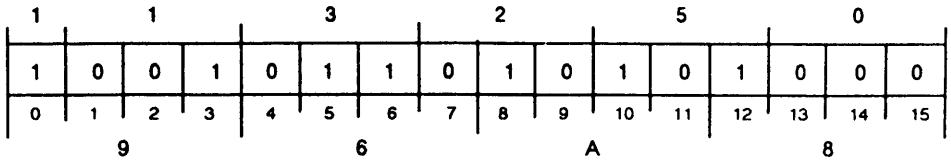
Example

```
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 one.

Arguments

None

Registers, Flags, and Stacks

FPAC0-FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if Z = 0) PC + 2 (if Z = 1)
Stack	Unchanged

Related Instructions

Floating-point status register skip instructions.

Exceptions

None

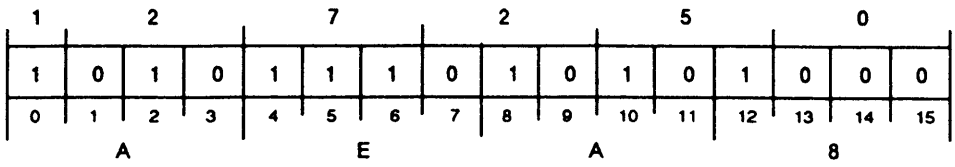
Example

```
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 zero.

Arguments

None

Registers, Flags, and Stacks

FPAC0-FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if N ≠ 0) PC + 2 (if N = 0)
Stack	Unchanged

Related Instructions

Floating-point status register skip instructions.

Exceptions

None

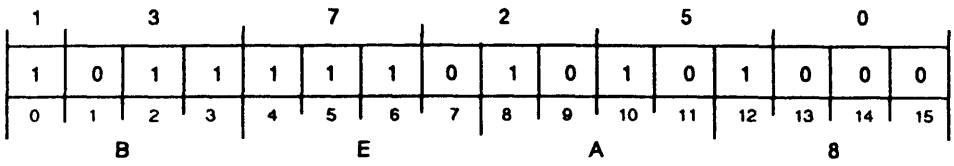
Example

```
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 zero.

Arguments

None

Registers, Flags, and Stacks

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 status register skip instructions.

Exceptions

None

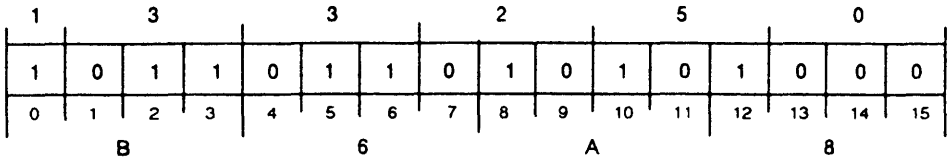
Example

```
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 one.

Arguments

None

Registers, Flags, and Stacks

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 status register skip instructions.

Exceptions

None

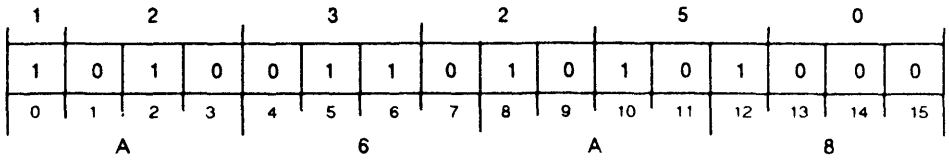
Example

```
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 one.

Arguments

None

Registers, Flags, and Stacks

FPAC0-FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if N ≠ 1) PC + 2 (if N = 1)
Stack	Unchanged

Related Instructions

Floating-point status register skip instructions.

Exceptions

None

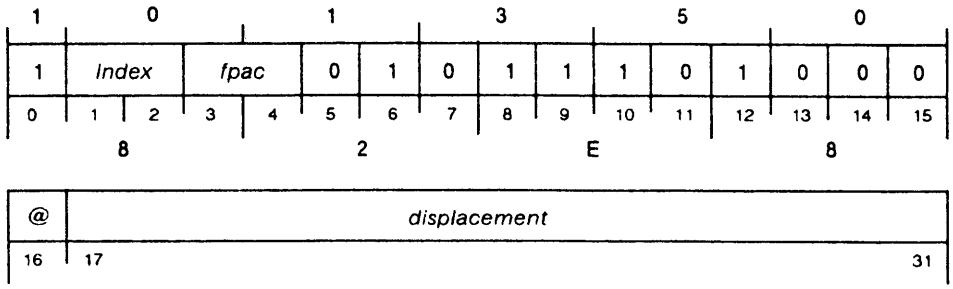
Example

```
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 double-precision floating-point number in memory from a double-precision floating-point number in a floating-point accumulator, and places the normalized 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.

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 one and terminates the instruction.

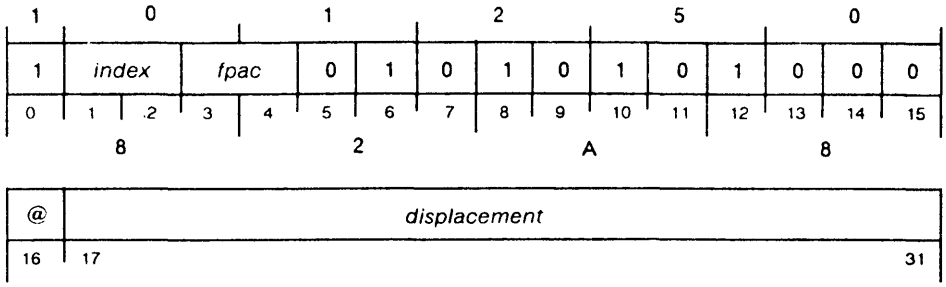
Example

```
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 single-precision floating-point number in memory from a single-precision 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 (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.
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 one and terminates the instruction.

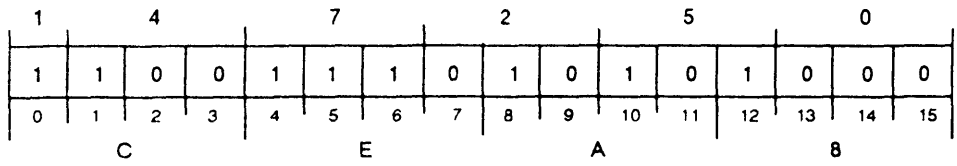
Example

```
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 zero. (**FSND** was previously called "Skip on No Zero Divide" and remains valid for this function. Refer to the section, "Faults and Status," in the chapter, "Floating-Point Computing.")

Arguments

None

Registers, Flags, and Stacks

FPAC0-FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if INV ≠ 0) PC + 2 (if INV = 0)
Stack	Unchanged

Related Instructions

Floating-point status register skip instructions.

Exceptions

None

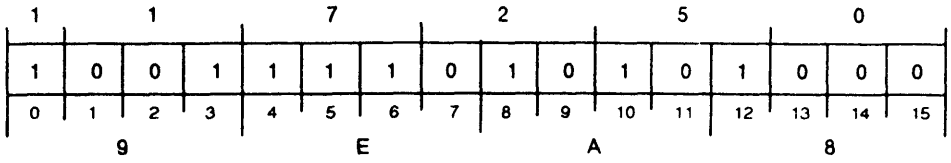
Example

```
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.
JMP    NOERROR      ;Otherwise, jump to 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 zero.

Arguments

None

Registers, Flags, and Stacks

FPAC0-FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if Z ≠ 0) PC + 2 (if Z = 0)
Stack	Unchanged

Related Instructions

Floating-point status register skip instructions.

Exceptions

None

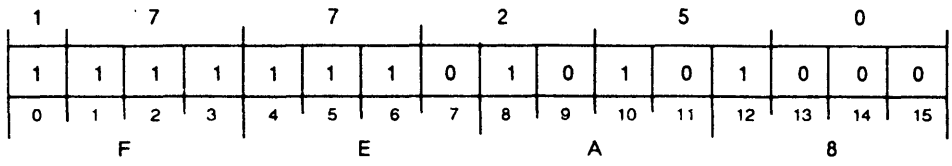
Example

```
FCMP   1,2          ;Compare the values in FPAC1 and FPAC2.
FSNE                ;
JMP    EQUAL        ;If the values are not equal, jump to NEQUAL.
JMP    NEQUAL       ;If the values are equal, jump to EQUAL.
```

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 zero.

Arguments

None

Registers, Flags, and Stacks

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 status register skip instructions.

Exceptions

None

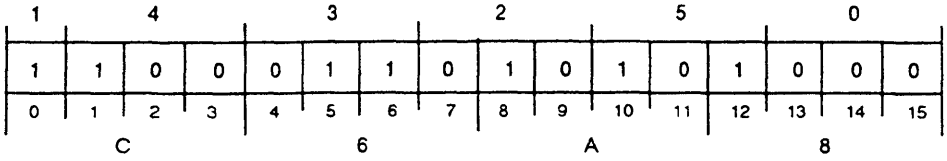
Example

```
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 zero.

Arguments

None

Registers, Flags, and Stacks

FPAC0–FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if MOF≠ 0) PC + 2 (if MOF = 0)
Stack	Unchanged

Related Instructions

Floating-point status register skip instructions.

Exceptions

None

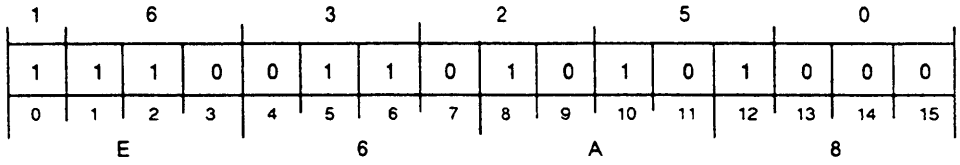
Example

```
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 flag (OVF) of the floating-point status register is zero.

Arguments

None

Registers, Flags, and Stacks

FPAC0–FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if OVF ≠ 0) PC + 2 (if OVF = 0)
Stack	Unchanged

Related Instructions

Floating-point status register skip instructions.

Exceptions

None

Example

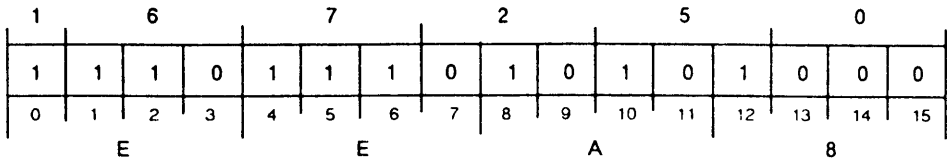
```
FAS   2,3          ;Add FPAC3 to FPAC2. If overflow results,
FSNO               ;jump to OVERFL. Otherwise, jump to NOOVER.
JMP   OVERFL
JMP   NOOVER
```

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 zero. (**FSNOD** was previously called "Skip on No Overflow and No Zero Divide.")

Arguments

None

Registers, Flags, and Stacks

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 status register skip instructions.

Exceptions

None

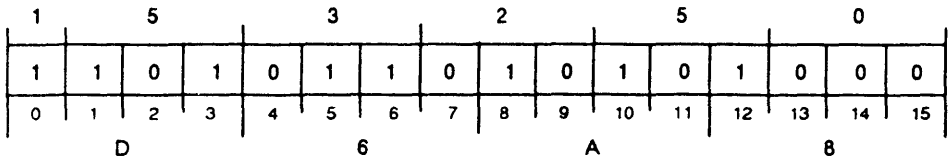
Example

```
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 is 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 zero.

Arguments

None

Registers, Flags, and Stacks

FPAC0-FPAC3	Unused
FPSR	Unchanged
PC	PC + 1 (if UNF ≠ 0) PC + 2 (if UNF = 0)
Stack	Unchanged

Related Instructions

Floating-point status register skip instructions.

Exceptions

None

Example

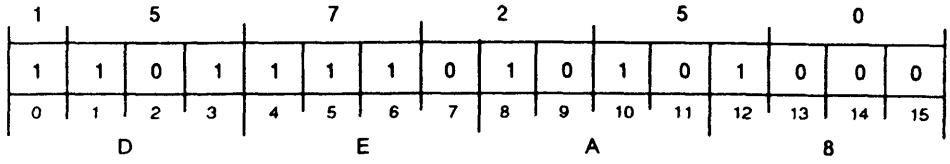
```
FSD    0,3          ;Subtract FPAC0 from FPAC3. If an underflow
FSNU                ;results, jump to UNDERF. Otherwise, jump to
JMP    UNDERF       ;NOUNDER.
JMP    NOUNDER
```

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 zero. (**FSNUD** was previously called "Skip on No Underflow and No Zero Divide.")

Arguments

None

Registers, Flags, and Stacks

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 status register skip instructions.

Exceptions

None

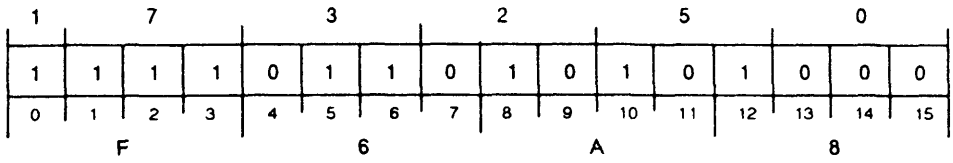
Example

```
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 zero.

Arguments

None

Registers, Flags, and Stacks

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 status register skip instructions.

Exceptions

None

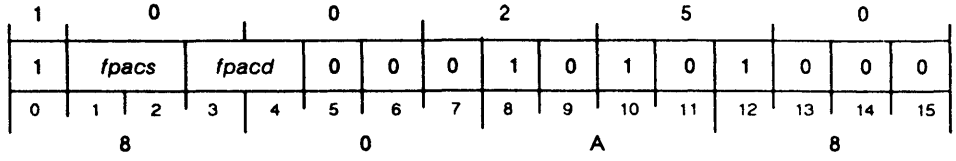
Example

```
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 single-precision floating-point number in *fpacs* from the single-precision floating-point number in *fpacd* and places the normalized 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 (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 one and terminates the instruction.

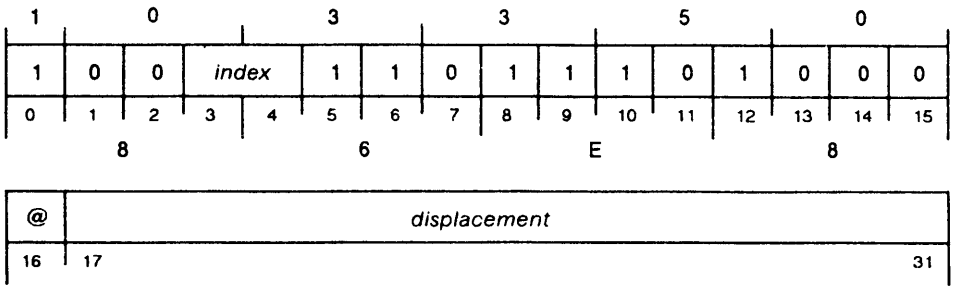
Example

```
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. The arguments specify the address of the first memory location.

The contents of the floating-point status register are stored as follows:

FPSR(0-15) are written to memory(0-15).

The contents of the second memory word depend on the state of the ANY flag:

If ANY = 0, memory(16-31) is undefined.

If ANY = 1, FPSR(49-63) are written to memory(17-31) — memory(16) is 0.

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 an FPSR value with any combination of bit 5 (TE) and bits 1-4 concurrently set.

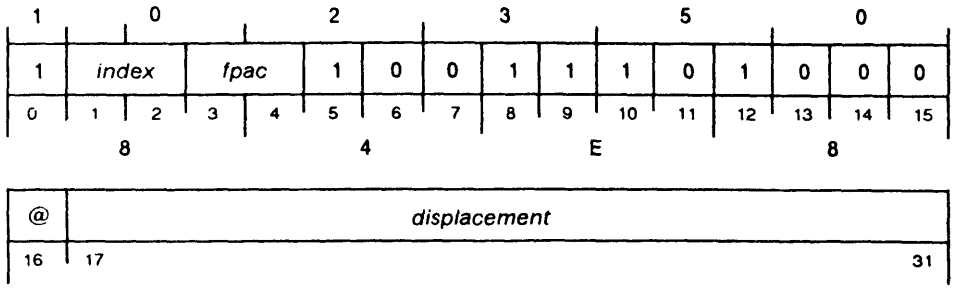
Example

```
FSST   STATUS        ;Store the narrow FPSR as a doubleword
                     ;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. The arguments specify the address of the first memory location. 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

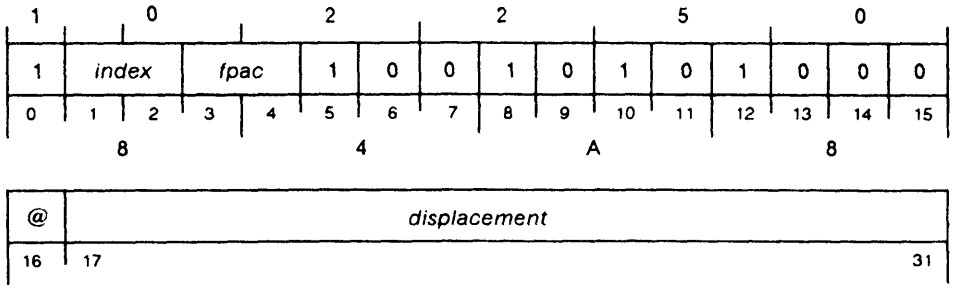
Example

```
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 single-precision floating-point number in the specified floating-point accumulator into two sequential 16-bit memory locations. The arguments specify the address of the first memory location. 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 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

XFSTS, LFSTS, FSTD, XFSTD, LFSTD
Store the contents of a floating-point accumulator into memory.

Exceptions

None

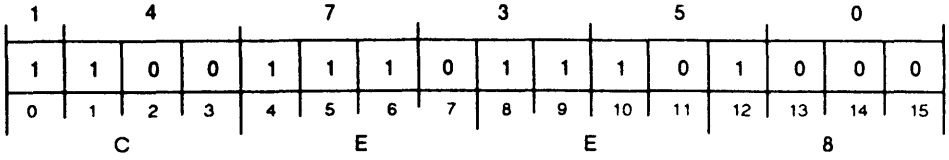
Example

```
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 zero, disabling floating-point fault detection.

Arguments

None

Registers, Flags, and Stacks

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

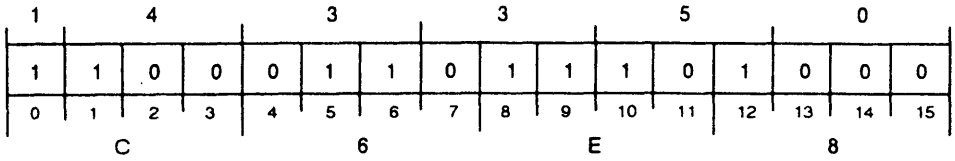
Example

```
FTD
FDD   3,2        ;Disable floating-point traps before dividing
                 ;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 one, enabling floating-point fault detection.

Arguments

None

Registers, Flags, and Stacks

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, FPSR(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.

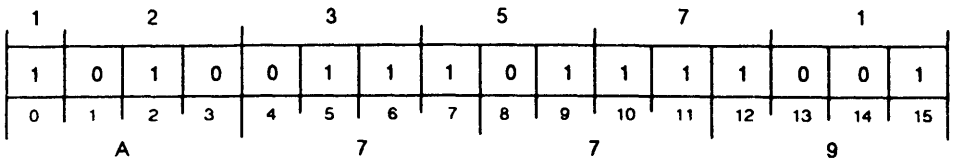
Example

```
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 sets the processor status register flags OVK and OVR to 0, disabling 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

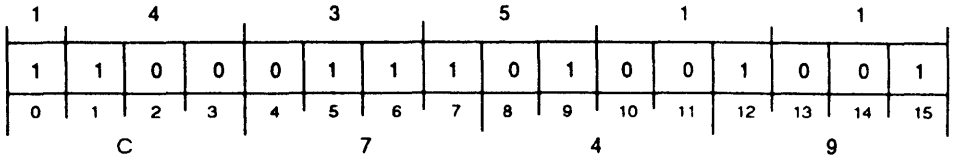
Example

```
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 sets the processor status register flags OVK to 1 and OVR to 0, enabling 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

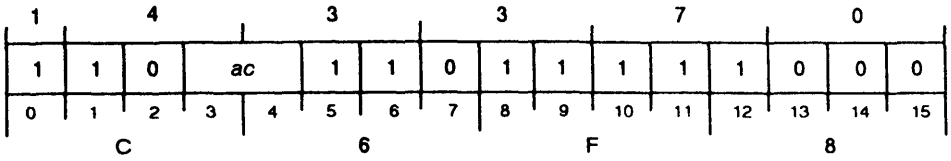
Example

```
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

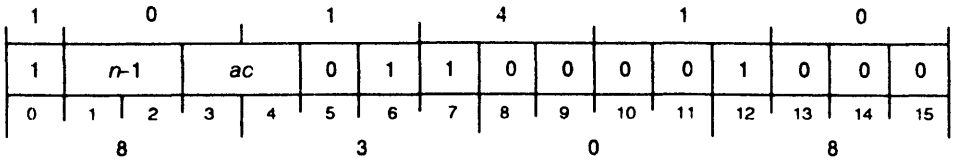
Example

```
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 equal to 4, *ac*(16–31) shifted out and set to 0. Since Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field, you 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

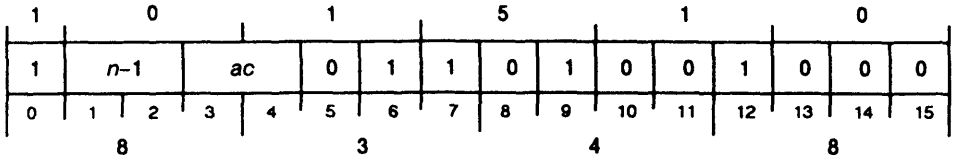
Example

```
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 equal to 4, *ac*(16–31) shifted out and set to 0. Since Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field, you 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

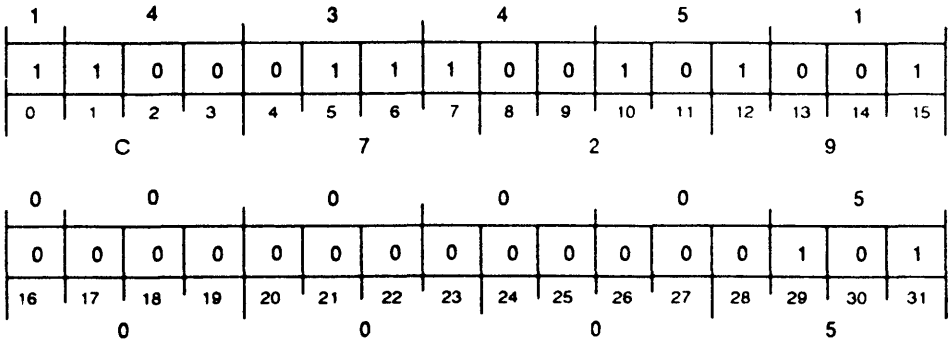
Example

```
ADC   1,1          ;AC1 is all ones.
HXR   2,1          ;Shift right 2 hex digits.
                   ;AC1[16-31] is now 000377₈.
```

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 valid 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.

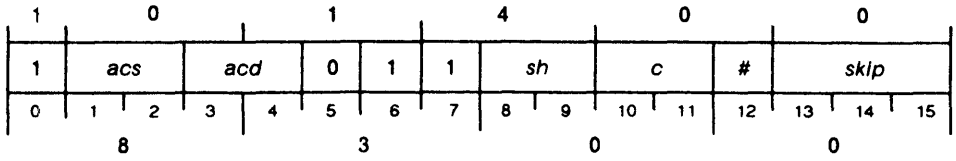
Upon a system reset, the interrupt mode is set to 0 (dedicated) with the I/O channels dedicated to the initial processor.

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.

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. The instruction increments the unsigned 16-bit integer in *acs* by 1 and places the result in the shifter. **INC** performs the specified shift operation, and loads the shift result 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*, 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*, 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 (#) set, 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, 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 skip 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

A 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 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 the no-load option (#) in combination with either the never skip or always skip option. Instruction may not end in 1000_2 or 1001_2 (reserved for other instructions).

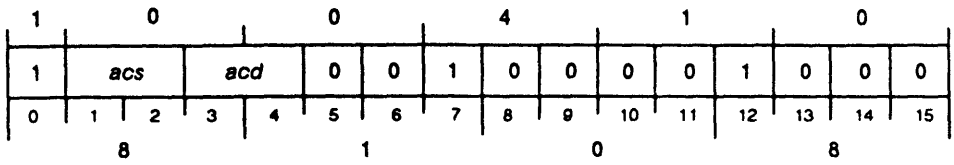
Example

```
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*. The instruction 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 resulting 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

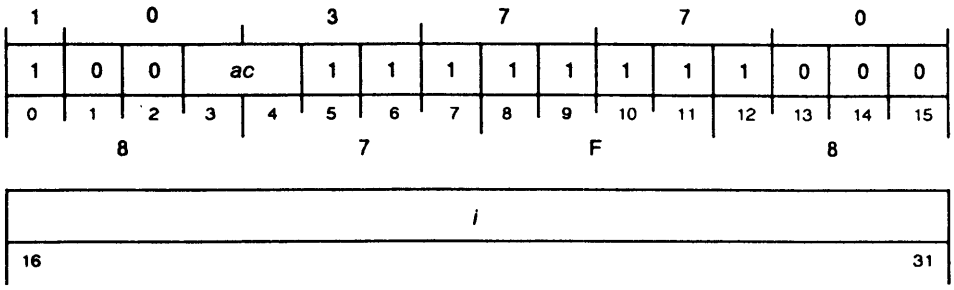
Example

```
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: OR *ac* → *ac*

Parameters: None

IORI forms the Inclusive OR of 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

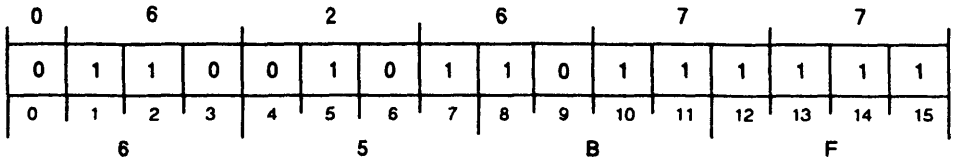
Example

```
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: Clear all I/O devices
0 → priority mask
0 → PSR
0 → FPSR(0-8)
0 → ION
0 → Busy and Done flags
off → address translator

Parameters: None

NOTE: **IORST = DICC 0,CPU**

IORST sends a reset signal to all devices on all I/O channels to clear their states. The instruction disables logical address translation and sets the following to 0: the 16-bit priority mask, the PSR, bits 0 through 8 of the FPSR, and ION.

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

In multiple-CPU systems, **IORST** *also resets IMODE to 0, clears all pending cross interrupts, and redirects all IOC traffic to the processor that issued the* **IORST** *instruction.*

Arguments

None

Registers, Flags and Stacks

AC0-AC3	Unused
Carry	Unchanged
ION	Set to 0
Overflow	Unaffected
PC	PC + 1
PSR	Set all bits to 0.
FPSR	Set bits 0-8 to 0.
Stack	Unchanged

Related Instructions

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

Exceptions

None

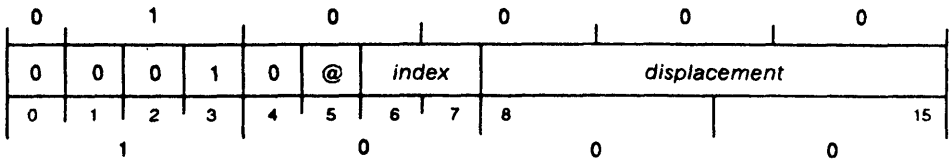
Example

```
IORST               ;First instruction of an 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 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 ≠ 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

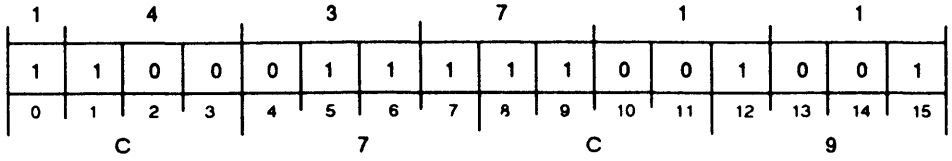
None

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 Zero — 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 0. (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

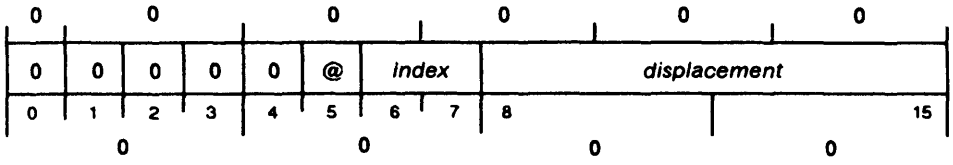
Example

```
    ;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

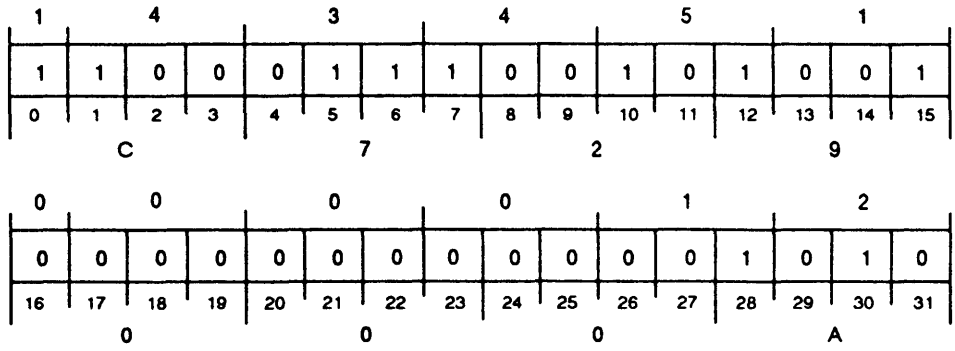
Example

```
    JMP   CLOSE       ;Jump to a location that is close by and
    .                 ;hence does not require an extended displacement.
    .
CLOSE:
          . . .
```

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 —> AC0 (from state block)**

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. The instruction 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 the **JPFLOAD** instruction 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)

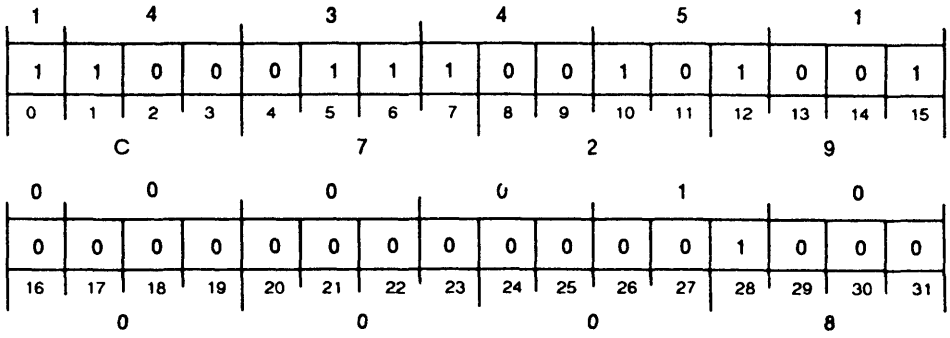
Exceptions

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 loaded from (with the **JPLOAD** instruction). 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

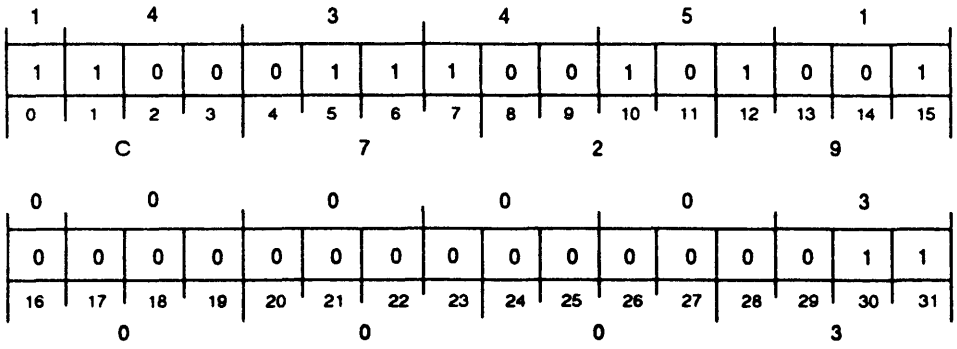
If no JP state block currently exists, **JPFLUSH** executes the next sequential word and returns error code 5 to AC1.

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

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 in 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

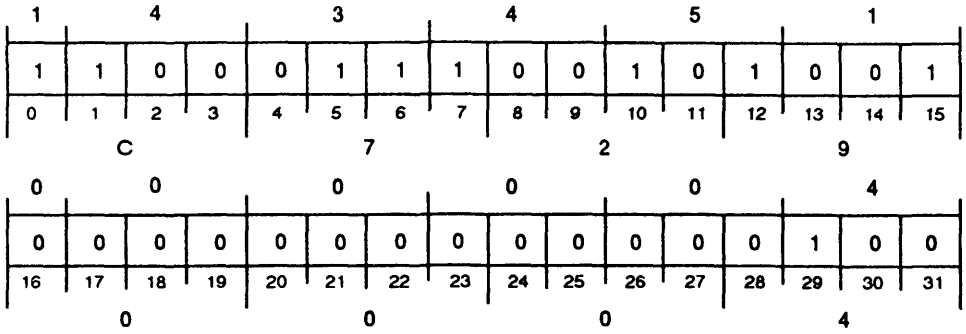
Exceptions

None

Load Control Store into JP — JPLCS

Multiprocessor Instruction

JPLCS
(error return)
(normal return)



Function: Control Store —> 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: If loading control store causes an error, AC1 contains error code, and, depending on error type, AC0, AC2, and AC3 are either unchanged or contain LCS error codes.

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

JPLCS loads writable control store into a target 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." (**JPLCS** does not output comment blocks, but does check the blocks for correct length.) **JPLCS** skips the next sequential word if no errors are detected during execution. **JPLCS** allows interrupts (on the source processor only) and is interrupt resumable. When the target processor either finishes loading its control store or encounters an error, it returns to its halted state.

Arguments

None

Registers, Flags, and Stacks

AC0 Before execution, contains doubleword as follows:

Bits	Contents	Description
0-15	Microcode Options	Specifies which microcode options to load (machine-specific). Currently defined values are: Bits 0-13 are reserved and should be set to 0. If bit 14 is 1, load architectural clock microcode. (This bit should be used on processors that support either architectural clocks or the PIT/RTC combination.) 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 load and verify options for microcode. 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 — 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 (physical address) to first block of data (bit 0 must be 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 block in error.

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

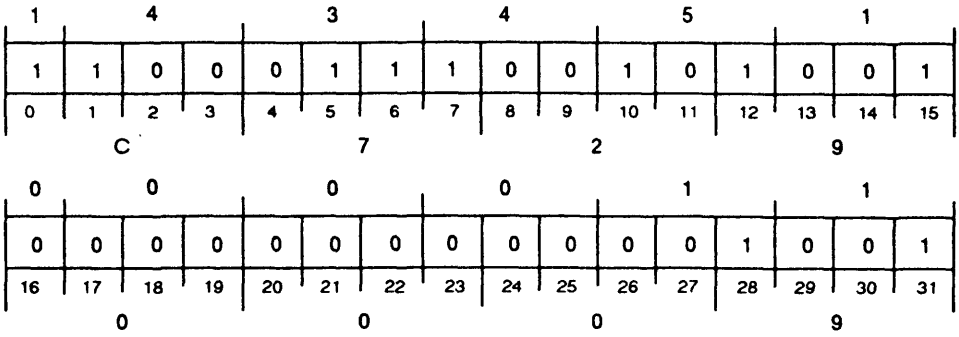
If one of the following conditions is true, JPLCS executes the next sequential word, and returns an error code to AC1:

Condition	Code
Not stopped	1
LCS error	3 (also see 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 —> AC0 (from state block)

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. In addition to the registers, flags, and stack parameters below, **JPLOAD** also loads the segment base registers and the state of the address translation unit (on or off).

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.
FPAC0–FPAC3	After execution, contain FPAC0–FPAC3 from state block.
FPSR	After execution, contains FPSR 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

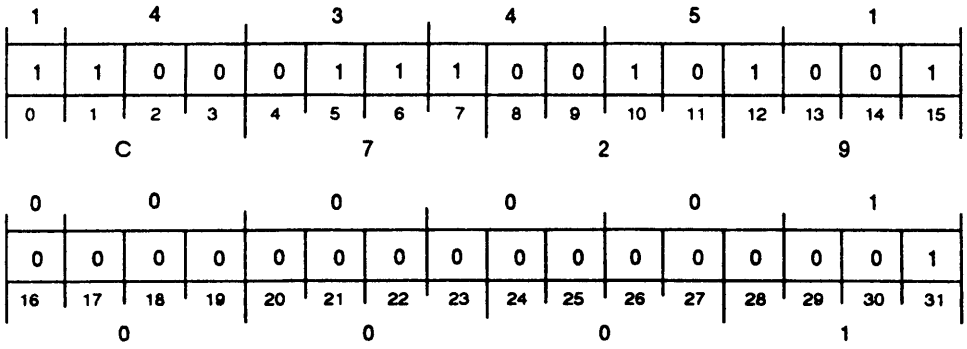
JPLOAD does not load SBR0 from the JP state block.

JPLOAD does not load any segment base registers if the address translation unit will be restored in the off state.

Start Another Processor — JPSTART

Multiprocessor Instruction

JPSTART
(error return)
(normal return)



Function: **Start target processor(AC0)**
Target processor loads 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 requesting processor converts JP state block address to physical address.**

Possible errors returned to AC1 are the following: processor failure, processor 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 target processor starts executing with the program counter value loaded from the state block. The **JPLOAD/JPFLUSH** address (internal pointer) 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 from.
	After execution, contents unchanged. If error 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 the address translation unit will be restored in the off state, the target processor does not have to load any segment base registers.

If an invalid address is loaded into SBR0, the processor disables the address translator and a protection fault occurs (AC1 contains code 3). This means that logical addresses are identical to physical addresses, and the fault is processed in phsyical address space.

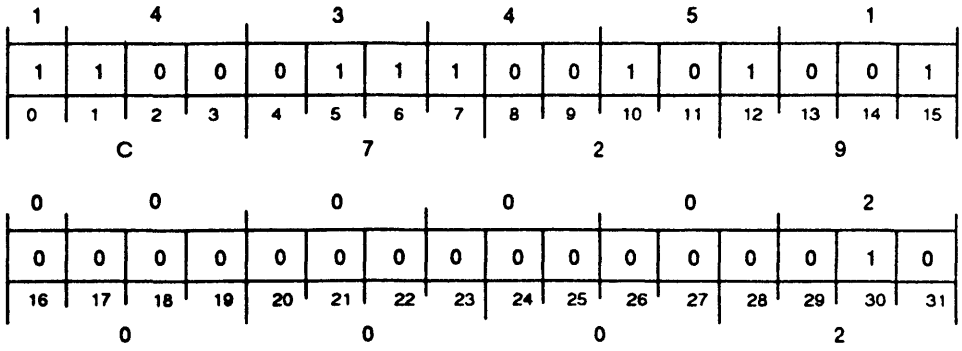
If one of the following conditions is true, **JPSTART** executes the next sequential word, and returns an error code to AC1:

Condition	Code
Target 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 target 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 state block length; bits 16-31 contain the current target processor status. The following are valid status values:

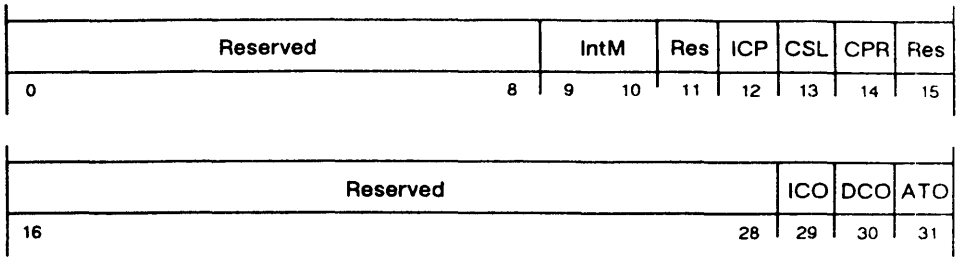
Value	Status
0	Stopped
1	Starting (optional)
2	Running
3	Stopping (optional)
4	Waiting for control store instruction

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

AC1 After execution,

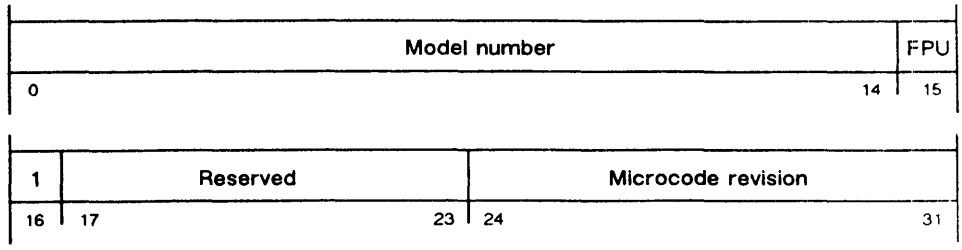
if error is detected, contains error code.

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	Res	Reserved for internal DGC use.
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	Res	Reserved for internal DGC use.
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 doubleword 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 an optional 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

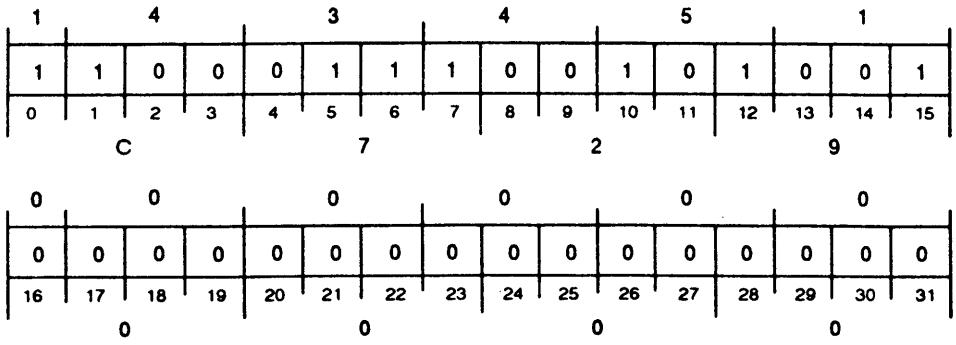
If one of the following conditions is true, **JPSTATUS** executes the next sequential word, and returns an error code to AC1:

Condition	Code
Nonexistent processor	2
Processor failure	4

Stop Another Processor — JPSTOP

Multiprocessor Instruction

JPSTOP
(error return)
(normal return)



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

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

NOTE: **Requesting 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** instruction function is performed, and the target processor will write a JP state block using the address in AC1. The **JPLOAD/JPFLUSH** address is unchanged. The target processor saves its stack parameters and then halts. (Processors stop at interruptible points that are not masked by ION.)

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, contents unchanged. If error 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

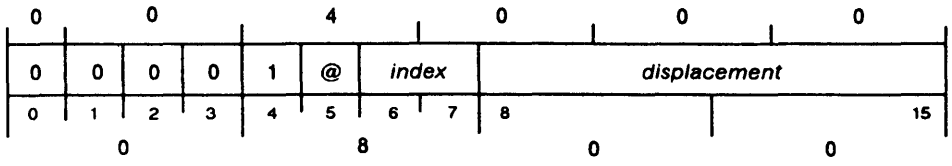
If one of the following conditions is true, **JPSTOP** executes the next sequential word, and returns an error code to AC1:

Condition	Code
Processor failure	4
Nonexistent processor	2
Processor not running	7

Jump to Subroutine JSR

ECLIPSE Instruction

JSR [@]*displacement*[,*index*]



Function: E → PC
PC+1 → AC3

Parameters: None

JSR increments the program counter by one (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	After execution, contains 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

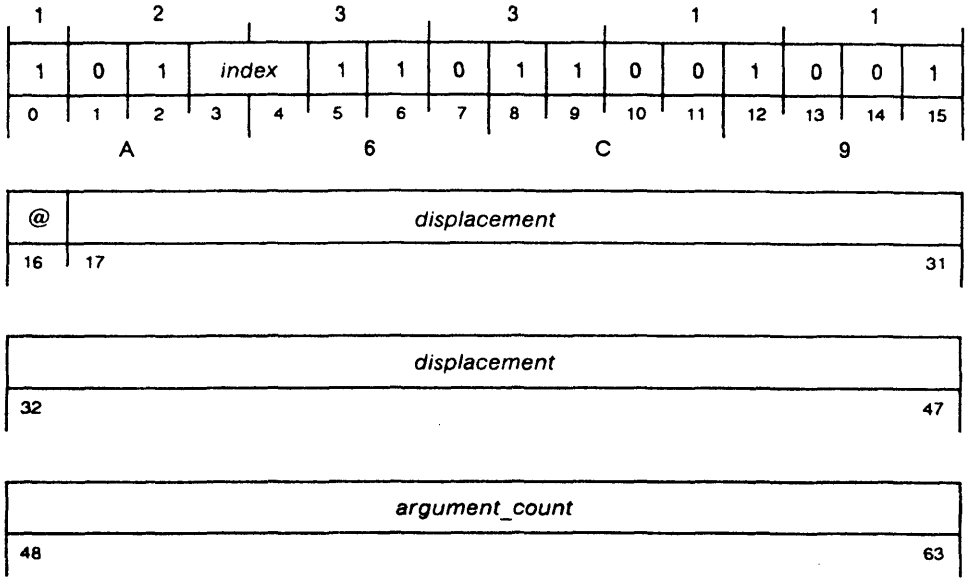
None

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 0 and loads the program counter with the target address.

For additional information on **LCALL**, refer to the section, "Transferring Program Control to Another Segment," in the chapter, "Program Flow Management."

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 AC1 contains error code 7. 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, AC1 contains error code 6, 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* doubleword depending on value of high bit (bit 48) of *argument_count*.

If high bit is 0, **LCALL** pushes onto the wide stack a doubleword 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 doubleword 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 doubleword and ignores *argument_count* coded with **LCALL** instruction. The instruction then places current PSR into bits 0–15 of stack doubleword.

(If target address is in inner segment, **LCALL** copies the number of doublewords specified in *argument_count* from the outer segment stack to the inner segment stack, and then pushes the PSR/*argument_count* doubleword onto inner stack. Note that, in this case, the instruction does not push the PSR/*argument_count* onto the outer segment's stack — if the *argument_count* is already on the outer segment stack, the instruction pops this value before crossing to the lower segment.)

Registers, Flags, and Stacks

AC0–AC2	Unused
AC3	After execution, contains PC + 4 (always refers to 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* doubleword.

If target address is inner segment, inner segment wide stack contains PSR/*argument_count* doubleword and copy of arguments.

Related Instructions

WSAVR, WSAVS

Push five doublewords onto the wide stack. Generally, one of these should be the first instruction in the subroutine.

WRTN

Pops six doublewords from wide stack. Generally, should be the last instruction in the subroutine.

When returning to an outer segment, the Wide Return instruction pops the return block, loads the PSR, and removes the number of arguments, specified by the **LCALL** *argument_count*, from both the inner segment stack and the outer segment stack.

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* doubleword, 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:

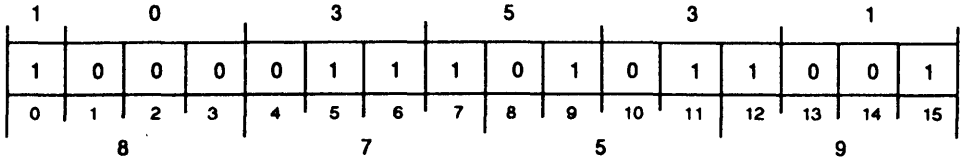
Error Code	Description	Program Counter Contents
2	Wide stack overflow	Wide stack fault handler
3	Invalid segment	Protection violation fault handler
6	Invalid gate	Protection violation fault handler
7	Illegal outward call	Protection violation fault handler

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 doubleword of CPU identification information into AC0.

Arguments

None

Registers, Flags, and Stacks

AC0 After execution, contains processor identification information as follows:

Bits	Contents
0–15	Model number (binary value of processor's allocated model number).
16–23	Current microcode revision.
24–31	Memory size (amount of physical memory available, measured in 256-Kbyte modules with an origin of 0). Note that the actual memory size in bytes is equal to (AC0(24–31) + 1) * 262144_{10} For example, 3_8 indicates 1 Mbyte; 7_8 indicates 2 Mbytes.

NOTE: *Systems which contain 64 Mbytes or more of physical memory return 377_8 to bits 24–31 of AC0.*

AC1–AC3 Unused

Carry Unchanged

Overflow 0

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

ECLID, NCLID Return CPU identification information. (Use the **NCLID** instruction on systems with more than 64 Mbytes of physical memory.)

Exceptions

None

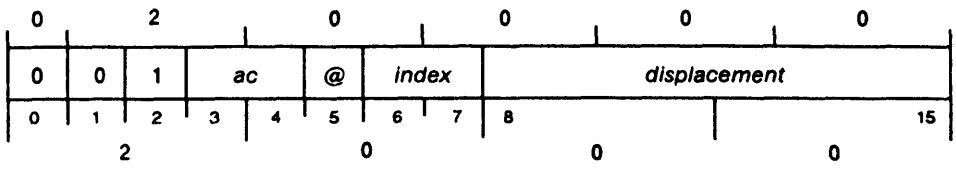
Example

```
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, XWLDA, LNLDA, LWLDA
Load an accumulator with the contents of memory.

Exceptions

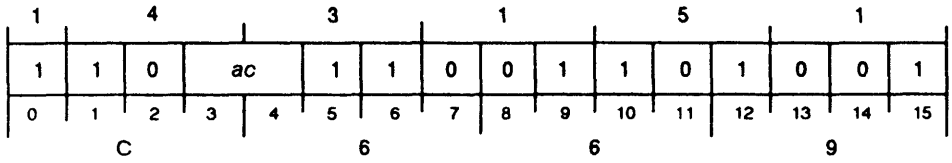
None

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 the 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

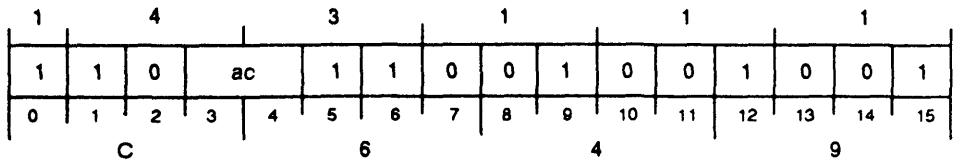
None

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

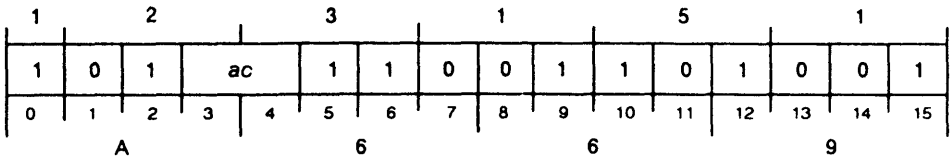
None

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

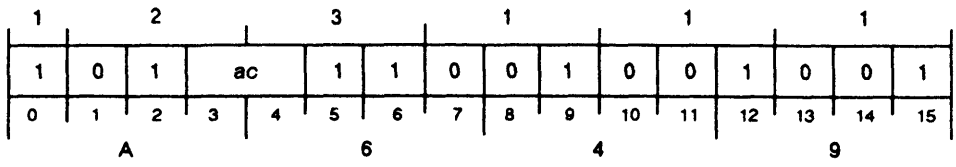
None

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

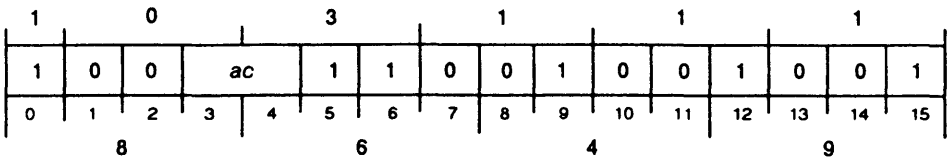
None

Example

```
LDASP 0             ;Get the wide stack pointer.
XWSTA 0,SAVE_SP     ;Save its value in memory.
```

Load Accumulator with Doubleword at WSP LDATS

LDATS *ac*



Function: (wsp) → *ac*

Parameters: None

LDATS uses the contents of the wide stack pointer as the address of a doubleword. It loads the contents of the addressed doubleword 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

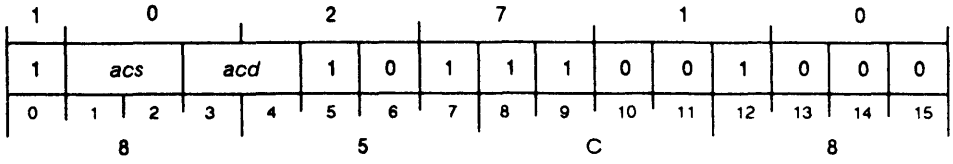
Example

```
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 memory into *acd*.

Arguments

acs(16-31) Before execution, contains byte address for load 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 (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

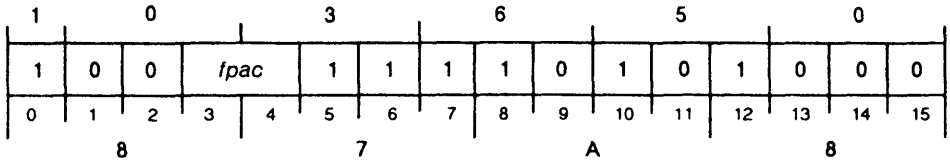
Example

```
    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. (Note that ECLIPSE 16-bit processors leave the contents undefined.)
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 decimal/ASCII fault, leaving the contents of *fpac* undefined.

For data type 7, an integer with more than 8 digits produces a decimal/ASCII fault.

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.

An attempt to load a -0 sets *fpac* to true zero.

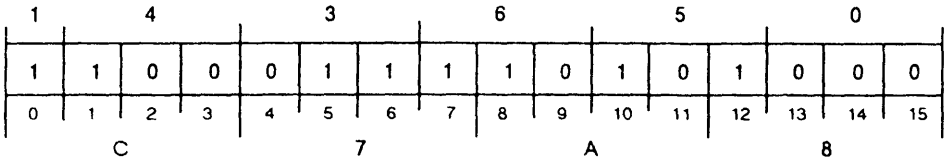
Example

```
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. The instruction 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 floating-point accumulator unless the integer in that floating-point accumulator 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

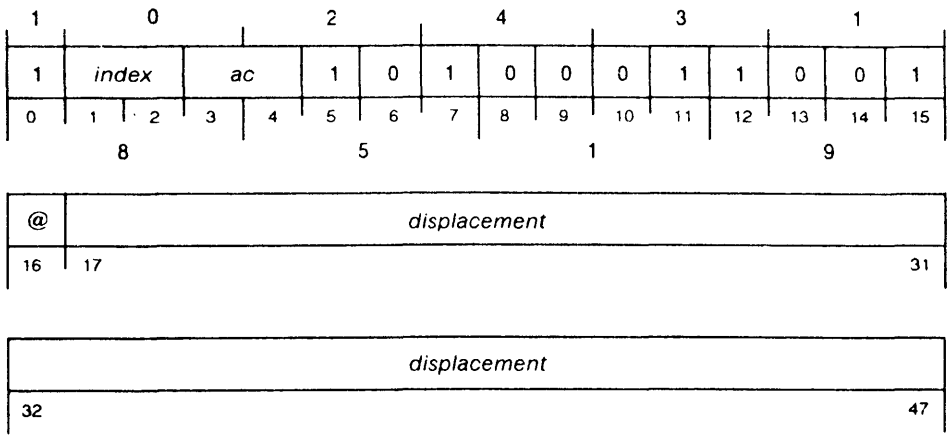
None

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 doubleword before the table as the lower limit (L) and the first doubleword 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 doubleword at E + 2(*ac*) - 2L.

The processor then tests this doubleword. If the doubleword equals -1 (37777777777_8), the processor transfers program control to the instruction following **LDSP**. Otherwise, the processor transfers program control by loading this doubleword plus the address of this doubleword 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 0–3 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*.

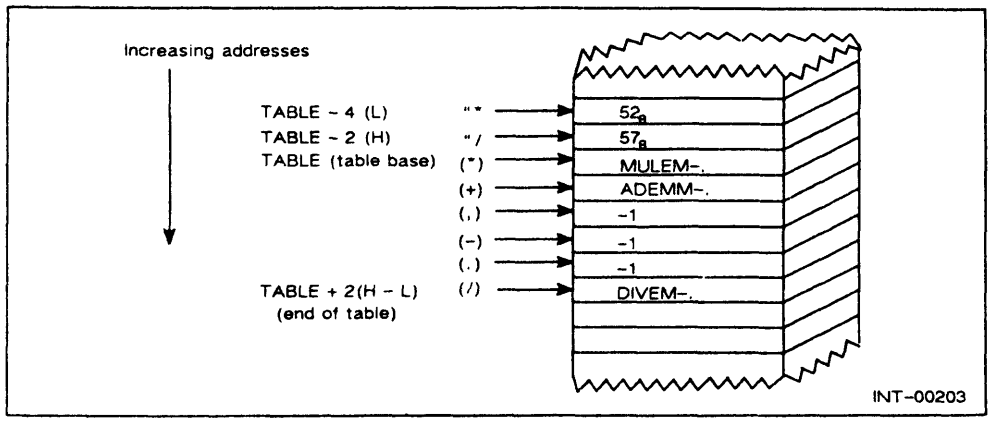
Exceptions

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 occurs.

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-.    ;/
```

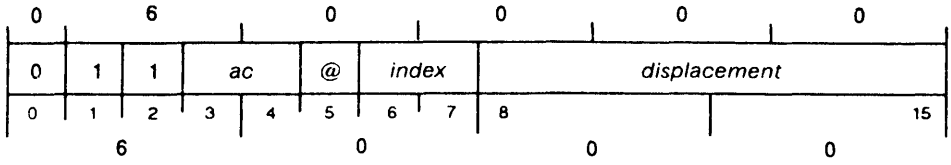
LDSP accesses the dispatch table (beginning at the address, TABLE), uses SYMBL to index into the table, and transfers program control to the address in the table entry. The following figure shows the dispatch table construction for this example.



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 returns valid data only 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 contain current segment, bits 4–16 set to 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

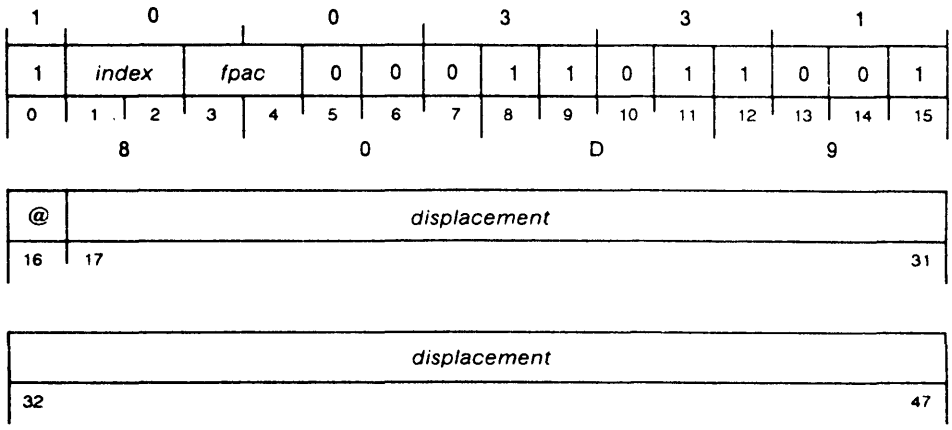
If the LEF mode is not enabled, the processor checks the I/O validity flag; if the I/O validity flag is set (I/O enabled) or the address translator is disabled, the processor executes the instruction as an I/O instruction. Otherwise, a protection violation occurs.

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 double-precision floating-point number in memory to a double-precision floating-point number in *fpac* and places the normalized 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

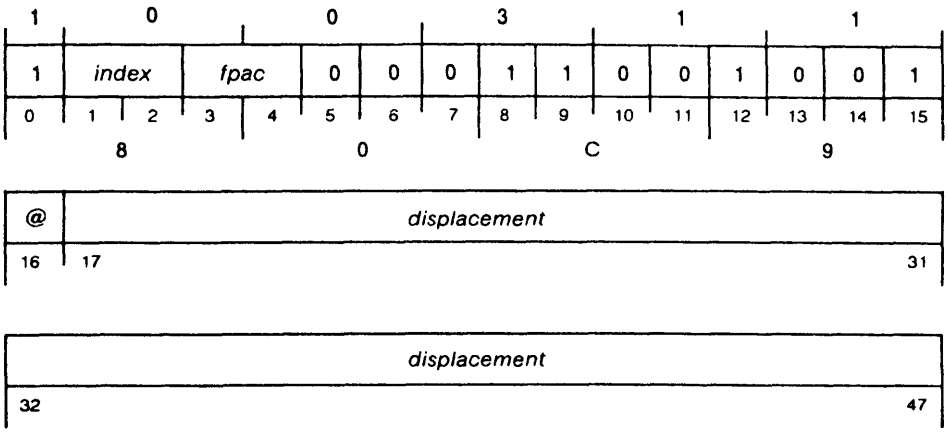
If addition produces an exponent overflow, the processor sets the FPSR(OVF) flag to one and terminates the instruction.

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 single-precision floating-point number in memory to a single-precision floating-point number in *fpac* 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 (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

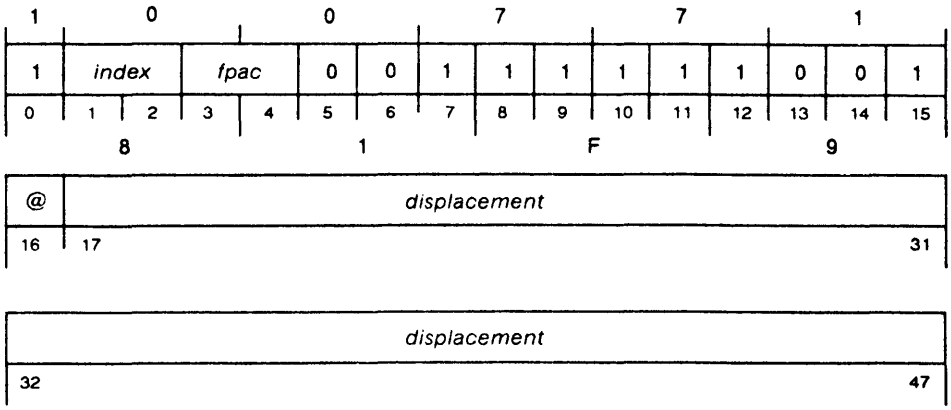
If addition produces an exponent overflow, the processor sets the FPSR(OVF) flag to one and terminates the instruction.

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 double-precision floating-point number in the specified *fpac* by the double-precision floating-point number in memory, and places the normalized 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

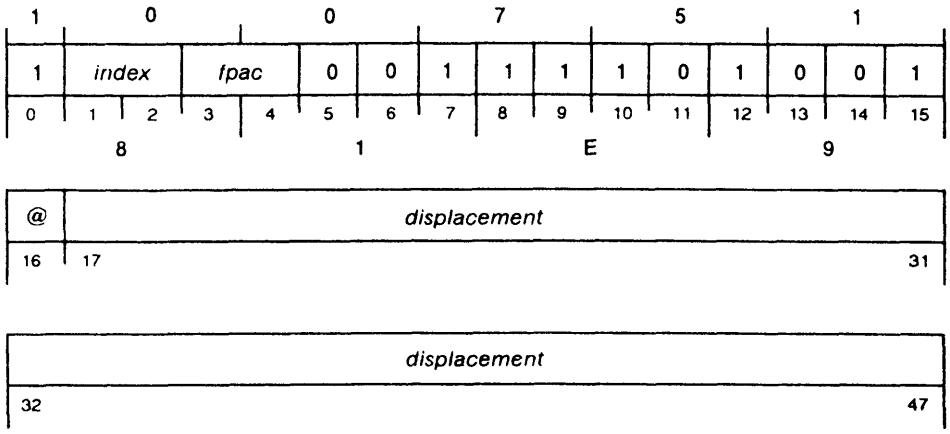
If the divisor (in memory) is 0, the processor sets FPSR(INV) to 1, places error code 0 in FPSR(INP) and the address of the instruction in FPSR(FPPC), and terminates the instruction.

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 single-precision floating-point number in a floating-point accumulator by a single-precision 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 (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, XFDMD, XFDMS, LFDMD
Divide a floating-point accumulator by the contents of memory.

Exceptions

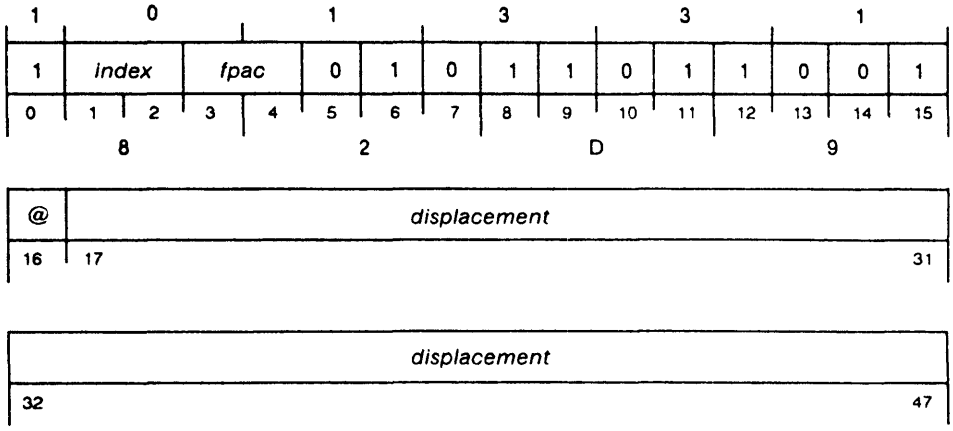
If the divisor (in memory) is 0, the processor sets FPSR(INV) to 1, places error code 0 in FPSR(INP) and the address of the instruction in FPSR(FPPC), and terminates the instruction.

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 starting at the effective address. 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

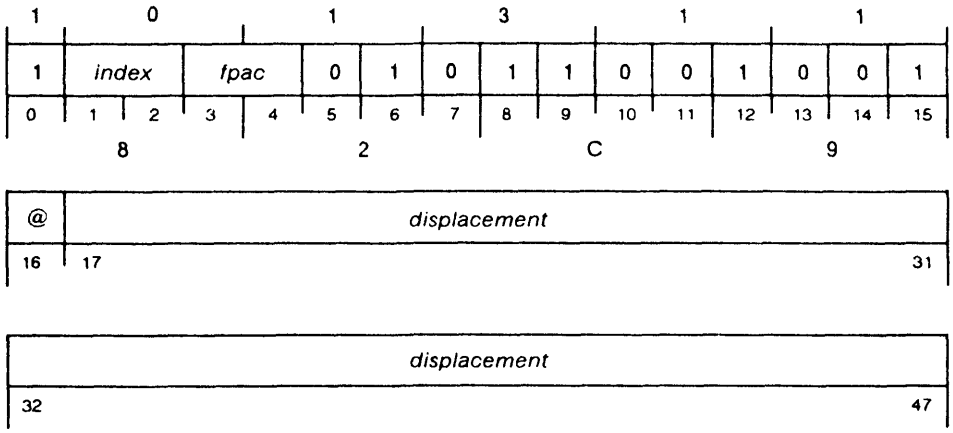
None

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 (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

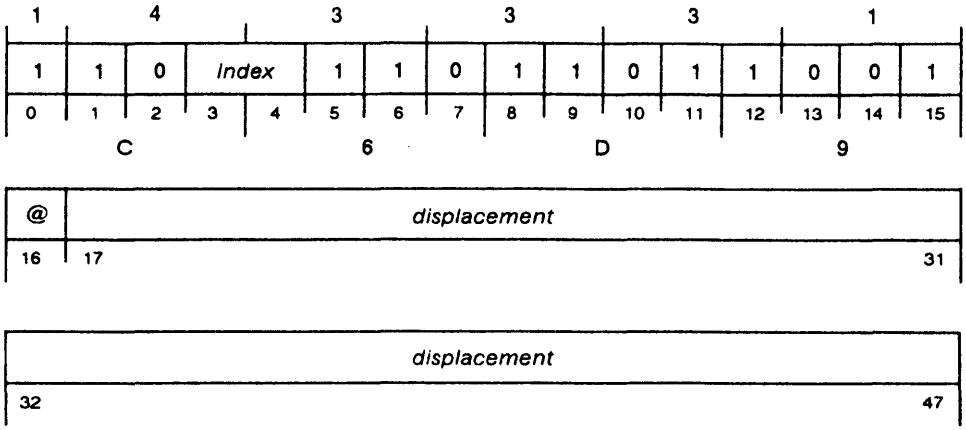
None

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 doublewords of memory as follows (bits are in parentheses):

FPSR(0) not set from memory. If any of first memory doubleword(1-4) is 1, ANY set to 1; otherwise, ANY is 0.

FPSR(1-8) from first memory doubleword(1-8).

FPSR(9-11) must be loaded from first memory doubleword(9-11) as zeros.

FPSR(12-15) not set from memory. Contains unchangeable floating-point identification code which is set by processor.

FPSR(16-21) set by processor to zeros.

FPSR(22) loaded from first memory doubleword(22) only if system supports an FPU capable of parallel operation; otherwise set by processor to 0.

FPSR(23-27) set by processor to zeros.

FPSR(28-31) are set according to new value of FPSR(INV):

If INV is 0, FPSR(28-31) undefined.

If INV is 1, FPSR(28-31) contains first memory doubleword(28-31).

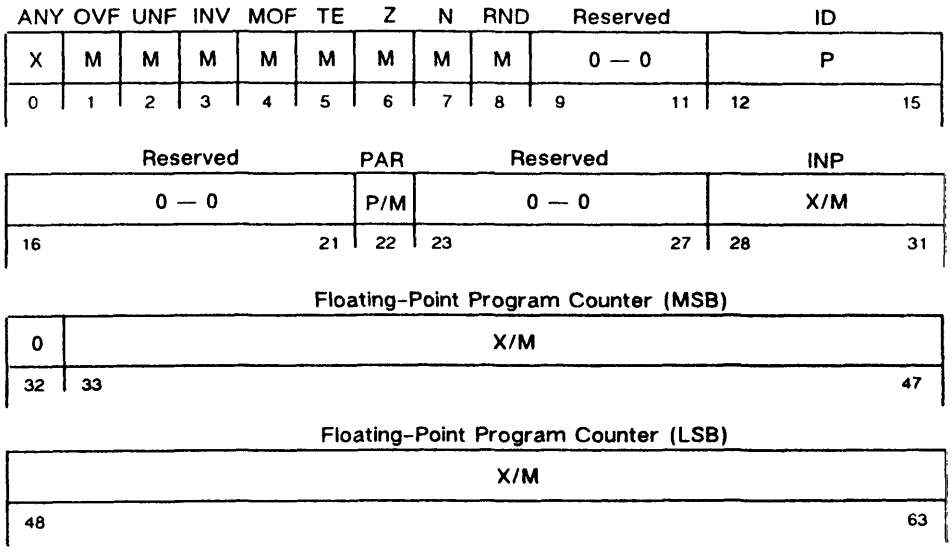
FPSR(32) set by processor to 0.

FPSR(33-63) set according to new value of FPSR(ANY):

If ANY is 0, FPSR(33-63) undefined.

If ANY is 1, FPSR(33-63) contains second memory doubleword(1-31).

The contents of the FPSR after execution is as follows:



X = Bit set according to state of another bit, or undefined
M = Bit set from memory
P = Bit set by processor

PC PC + 3

Stack Unchanged

Related Instructions

FLST Load Floating-Point Status

FSST, LFSST Store floating-point status register to memory.

Exceptions

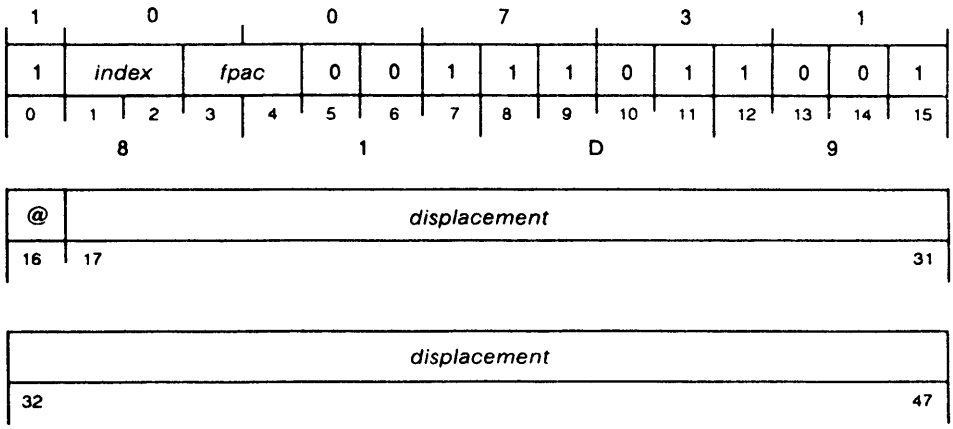
LFLST initiates a floating-point trap if FPSR(ANY) and FPSR(TE) are both one after FPSR(FPPC) is loaded.

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 double-precision floating-point number in *fpac* by a double-precision floating-point number in memory, and places the normalized result in *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

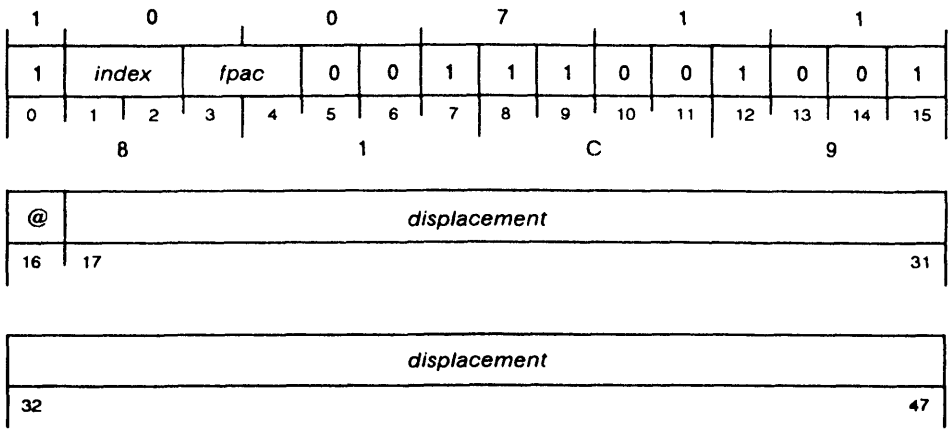
If multiplication produces an exponent overflow, the processor sets FPSR(OVF) to one and terminates the instruction.

Example

```
LFLDD 0,DATA1      ;Multiply the two double-precision floating-point
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 single-precision floating-point number in *fpac* by a single-precision floating-point number in memory and places the normalized result in *fpac*.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contains normalized result (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

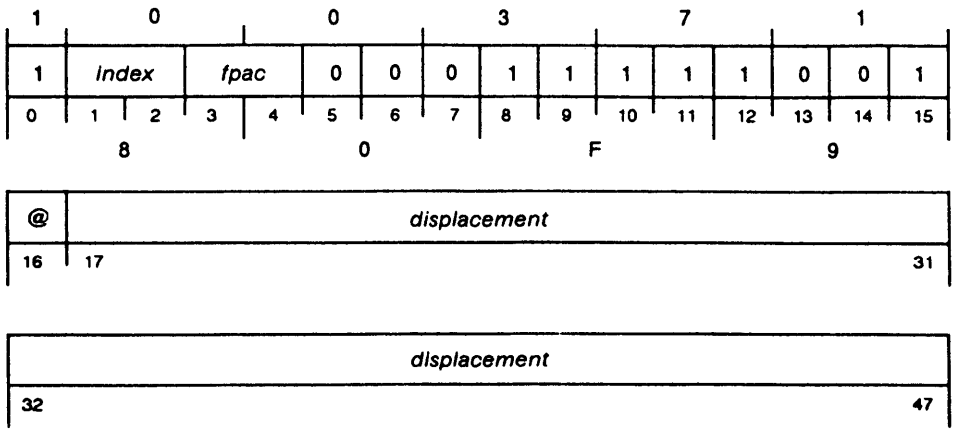
If multiplication produces an exponent overflow, the processor sets FPSR(OVF) to one and terminates the instruction.

Example

```
LFLDS  3,MUL1      ;Multiply the two single-precision floating-point
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 double-precision floating-point number in memory from a double-precision floating-point number in *fpac* and places the normalized result in *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

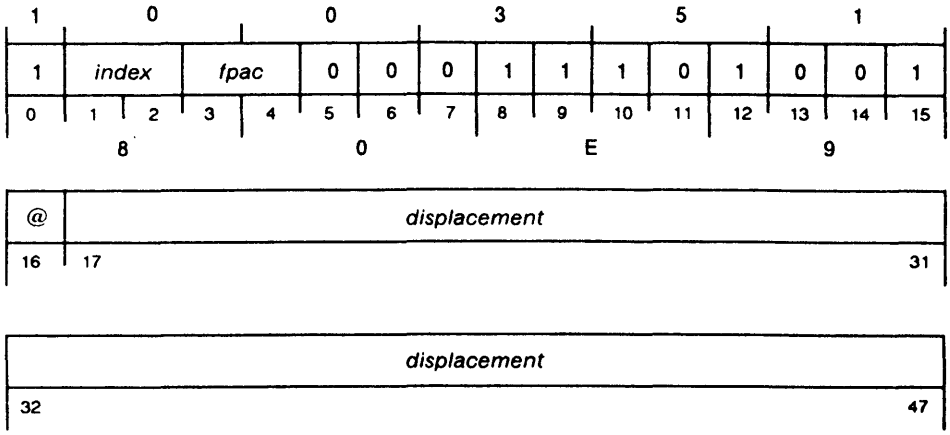
If subtraction produces an exponent overflow or underflow, the processor sets the appropriate FPSR flag to one and terminates the instruction.

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 single-precision floating-point number in memory from the single-precision floating-point number in *fpac* and places the normalized result in *fpac*.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contains normalized 32-bit result (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

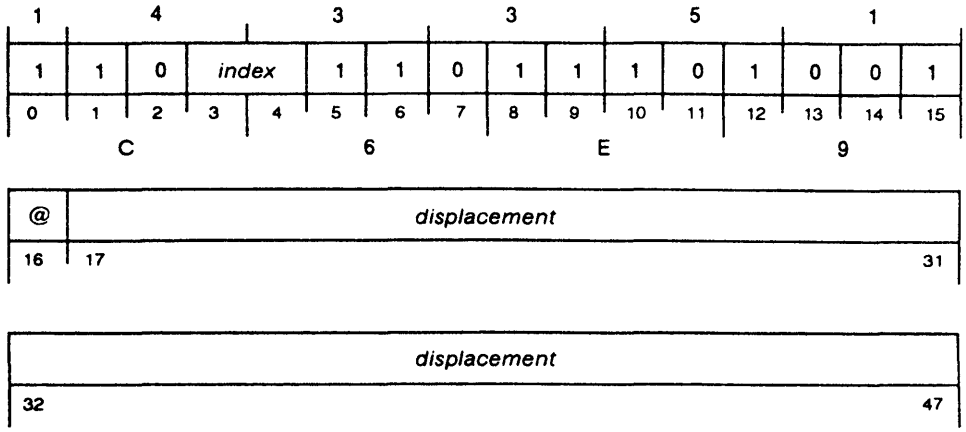
If subtraction produces an exponent overflow or underflow, the processor sets the appropriate FPSR flag to one and terminates the instruction.

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 floating-point status register 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 doublewords as follows (bits are in parentheses): First memory doubleword(0-31) contains FPSR(0-31). Second memory doubleword contains: memory(0) set to 0. If FPSR(ANY) is 0, memory(1-31) undefined. If FPSR(ANY) is 1, memory(1-31) contains FPSR(33-63). 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

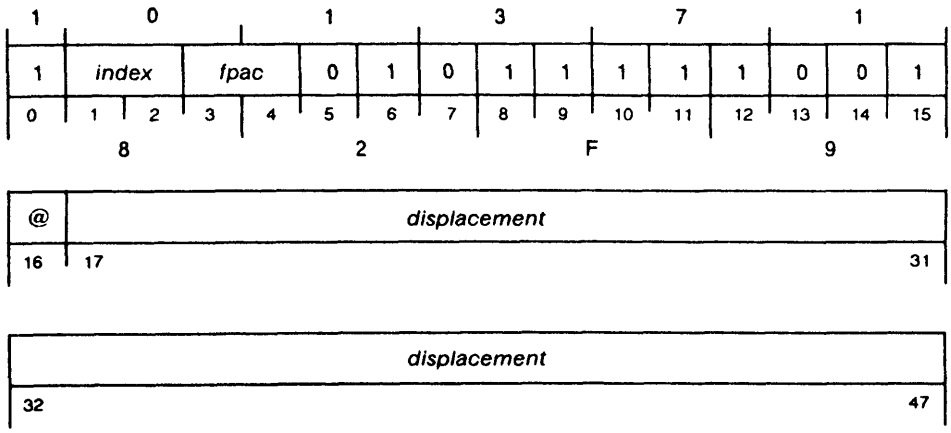
LFSST will not store an FPSR value with any combination of bit 5 (TE) and bits 1-4 concurrently set to one.

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 value in *fpac* into two consecutive doublewords in memory starting at the effective address. **LFSTD** will move unnormalized data without change.

Arguments

fpac Before execution, contains 64-bit value.

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

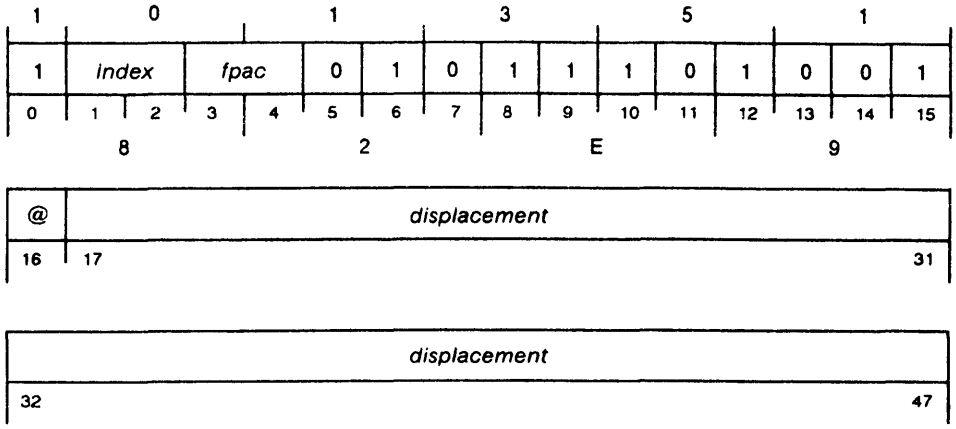
None

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 value in *fpac* into a doubleword in memory at the effective address. **LFSTS** will move unnormalized data without change.

Arguments

fpac(0-31) Before execution, contains 32-bit value.

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

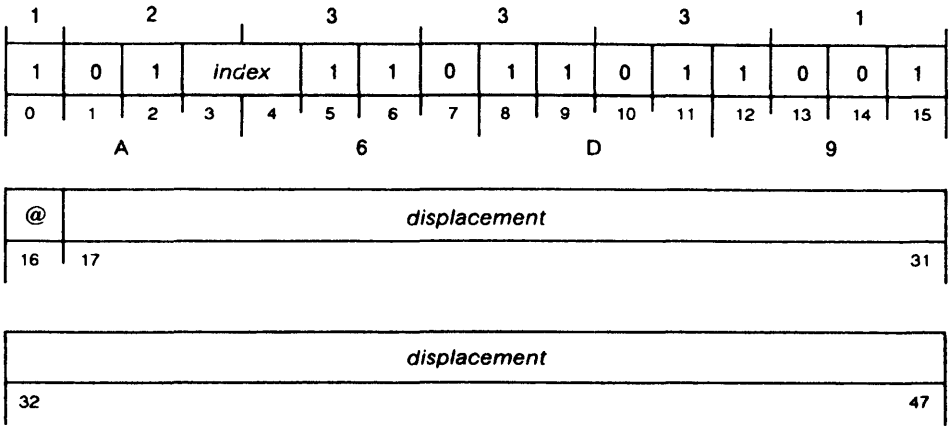
None

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

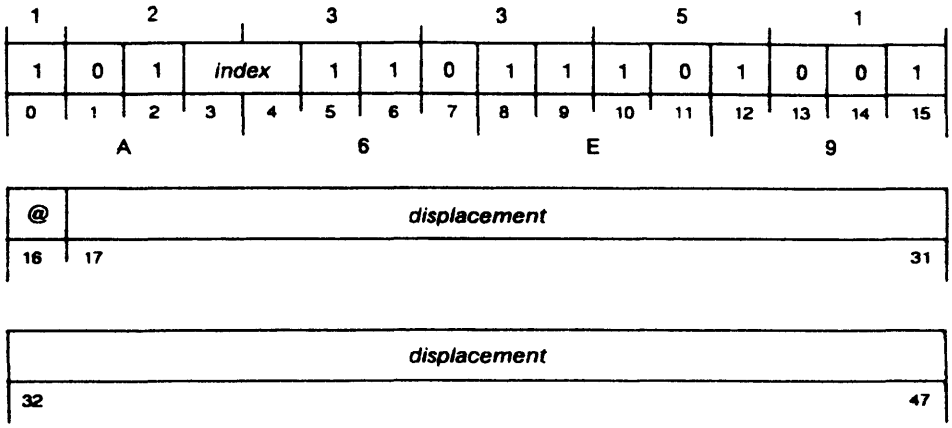
None

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 then 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

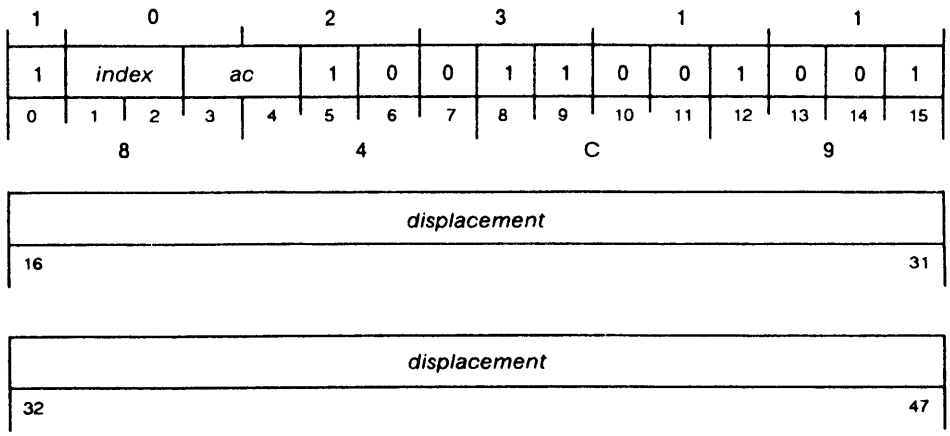
None

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 *ac*, and then zero-extends the value to 32 bits.

Arguments

ac(24-31) After execution, contains byte result (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

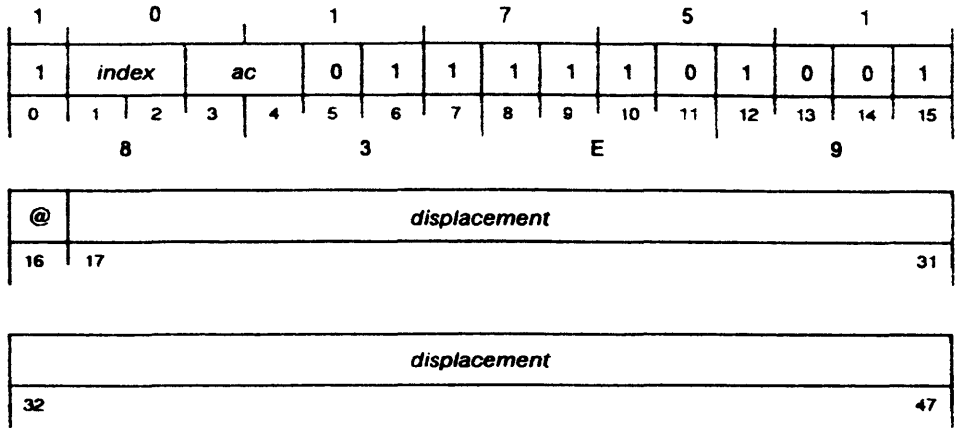
None

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 (bit 0 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

LEF, ELEF, XLEF
Load an effective address into an accumulator.

Exceptions

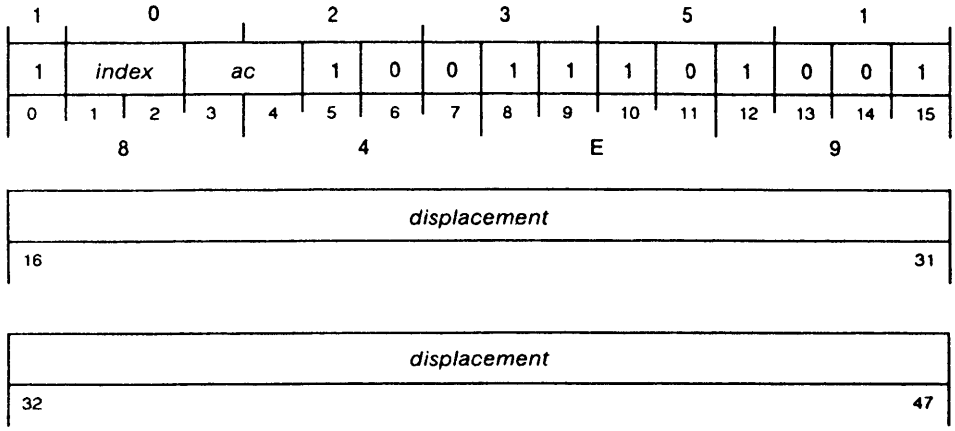
None

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

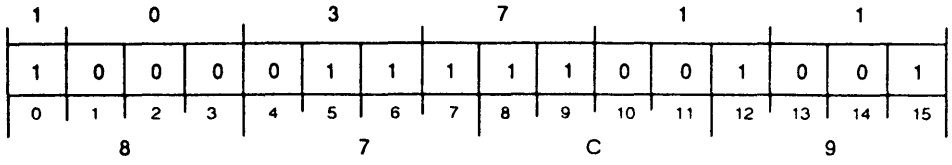
Example

```
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, zero-filled]
0 → referenced bit
unchanged → modified bit

Parameters: AC0 = ? → 0(bits 0-29), mod(bit 30), ref(bit 31)
AC1 = page frame # (13-31) → unchanged

LMRF loads the modified and referenced bits of the page frame, specified in AC1, into AC0. The referenced bit of the addressed page frame is then set to 0.

Arguments

None

Registers, Flags, and Stacks

AC0	After execution, contains modified (bit 30) and referenced bit (bit 31) (bits 0-29 set to 0).
AC1(13-31)	Before execution, contains page frame 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

Exceptions

The results are undefined if the address translator is not enabled or you specify either a nonexistent page frame or a page frame outside main memory.

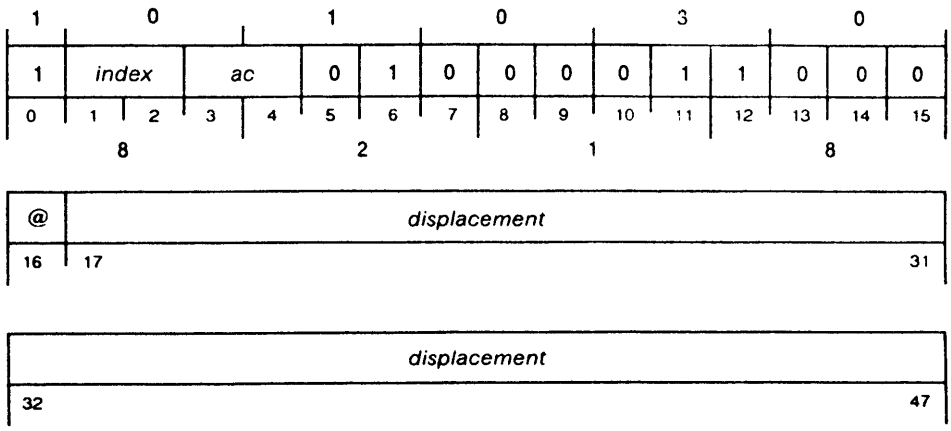
Example

```
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 carry → CRY

Parameters: None

LNADD adds the signed 16-bit integer contained in memory to the signed 16-bit integer contained in the specified accumulator, sign-extending the 16-bit result to 32 bits.

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 contents of memory to an accumulator.

Exceptions

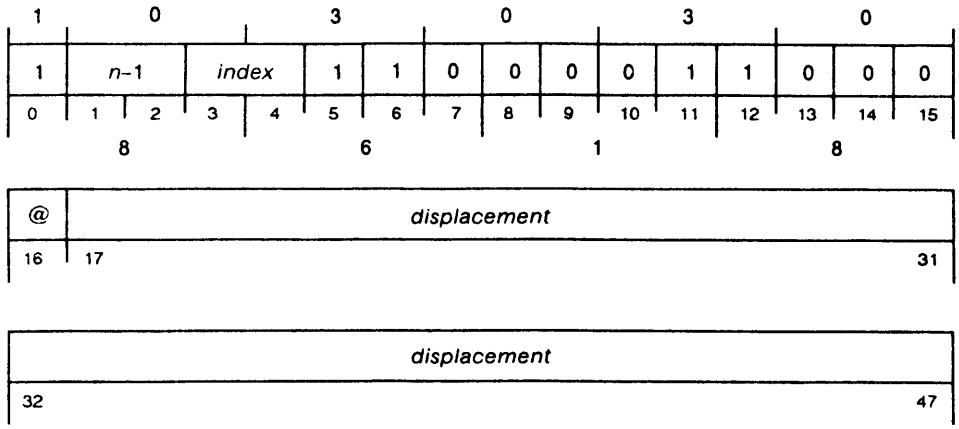
If the add produces a result greater than 32,767, PSR(OVR) is set to one.

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 carry → CRY

Parameters: None

LNADI adds a value in the range of 1 to 4 to the signed 16-bit integer in memory.

Arguments

n — Integer in range 1 to 4.

Since Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field, you 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 contents of memory.

Exceptions

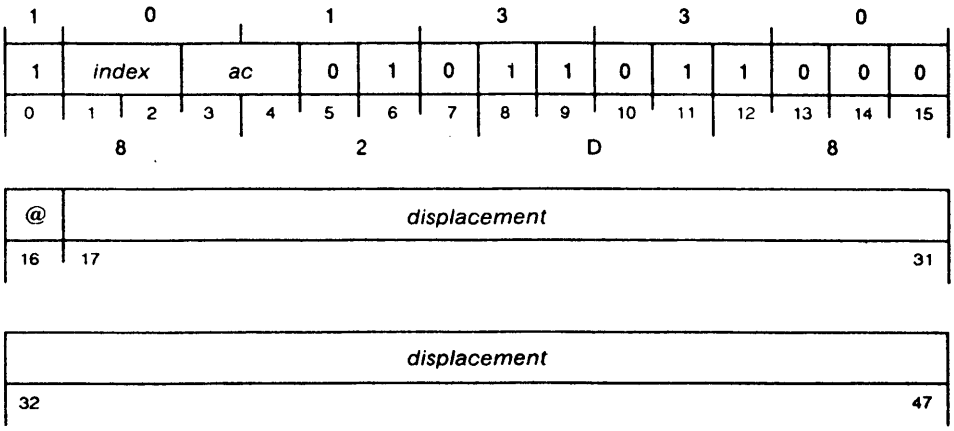
If the add produces a result greater than 32,767, PSR(OVR) is set to one.

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 *ac* 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 *ac*.

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	Set to 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

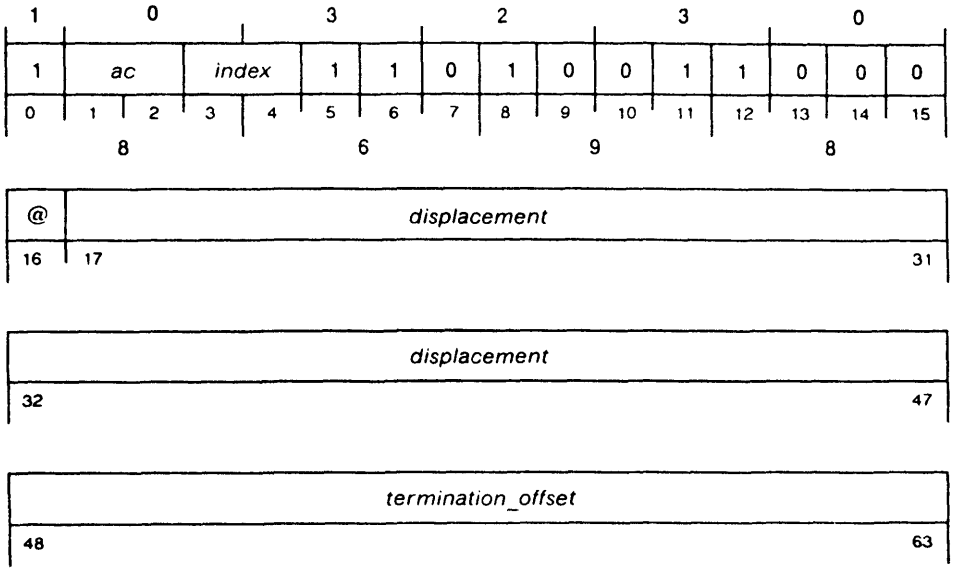
If the quotient is outside the range -32,768 to +32,767 inclusive, or if the memory location contains 0, an overflow occurs, PSR(OVR) is set to 1, and *ac* is unchanged.

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 carry → 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 through the 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.
- 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 signed PC-relative address for normal return. Argument ranges from 0 to 64 Kwords. (This value is sign-extended to 32 bits for the addition. The final value contains the current segment of execution in bits 1-3.)

[@]*displacement*[,*index*]
: Specifies effective address of a memory word 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 a value one greater than actual loop count.

WBR
: Use the Wide Branch instruction to end the DO-loop (to loop back to the **LNDO** instruction).

Exceptions

In any return block, the contents of the specified memory location and the program counter value are both undefined.

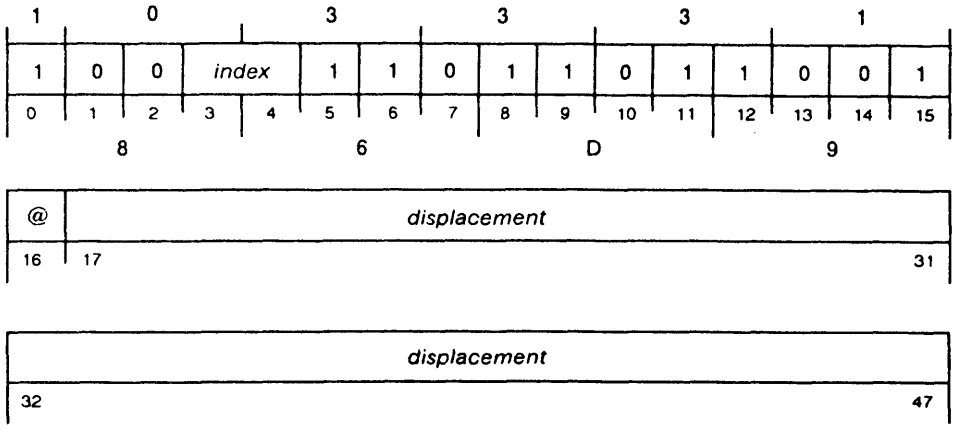
If a fixed-point overflow fault occurs while incrementing the DO-loop variable, the contents of the memory location and the PC value in the return block are undefined. (AC0 will contain the address of the DO-loop instruction.)

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 memory 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

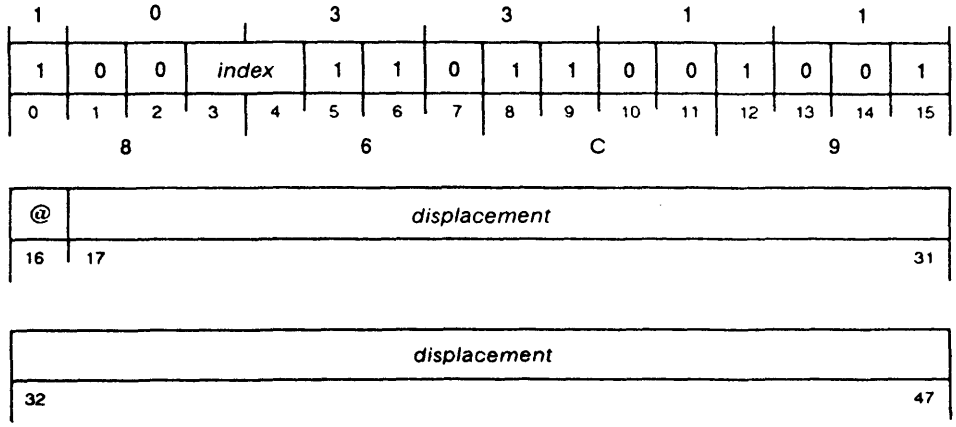
None

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 memory 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

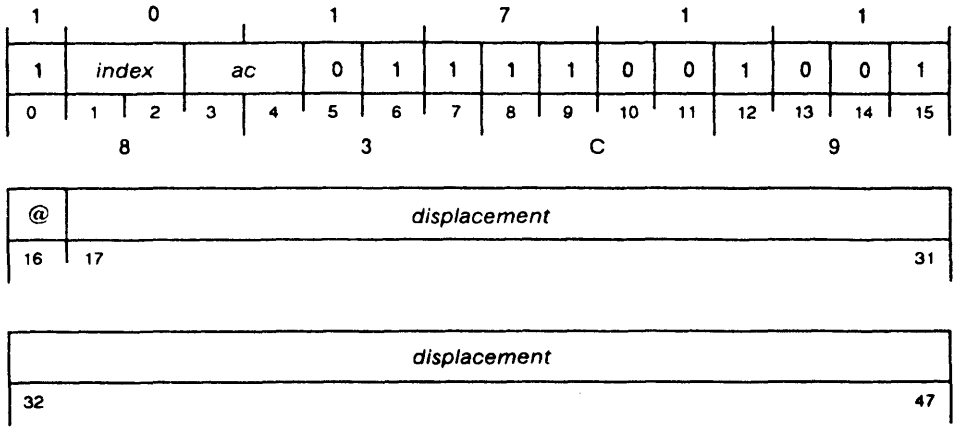
None

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

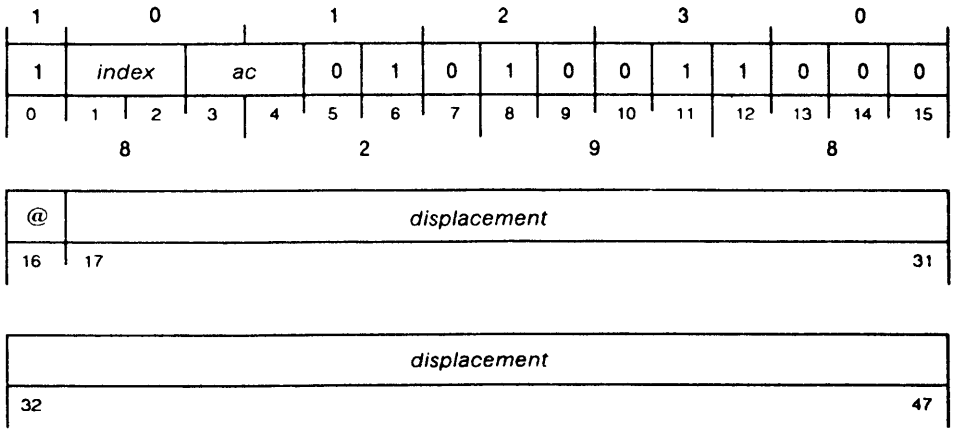
None

Example

```
LNLDA 0,SINGLE_WORD          ;Get 16 bit value and sign-extend.
LWSTA 0,DOUBLE_WORD          ;Store the value as a doubleword.
```

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 memory by the signed 16-bit integer contained in *ac*. It then sign-extends the result to 32 bits and places the result in the *ac*.

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	Set to 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

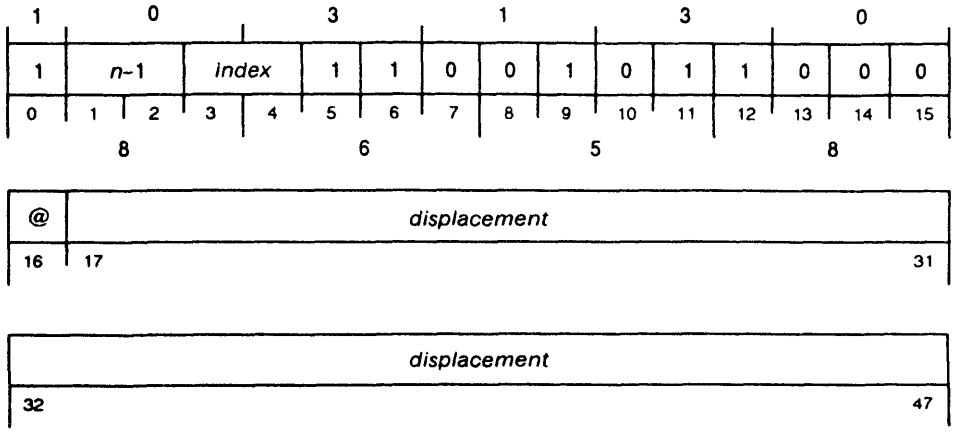
If the result is outside the range of -32,768 to +32,767 inclusive, an overflow occurs, and PSR(OVR) is set to 1.

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 carry → 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.

Since Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field, you 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

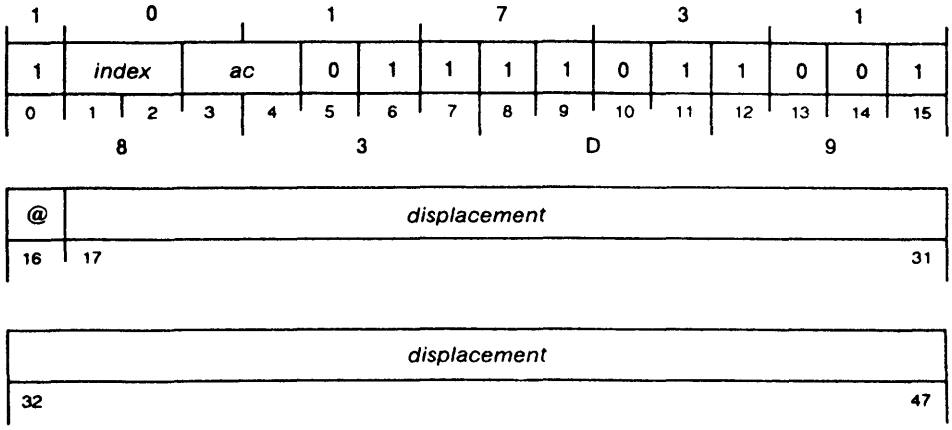
None

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 memory.

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

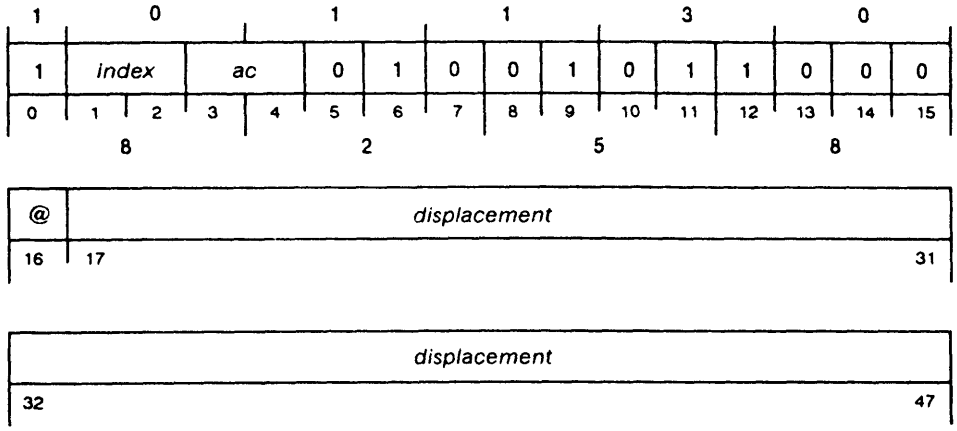
None

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 carry → CRY

Parameters: None

LNSUB subtracts the signed 16-bit integer contained in memory from the signed 16-bit integer contained in *ac*. Then it sign-extends the result to 32 bits and stores it in *ac*.

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

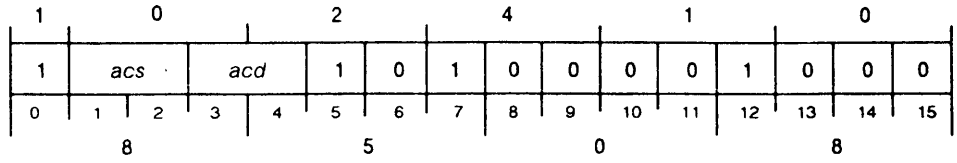
Example

```
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 one bit to zero.

Exceptions

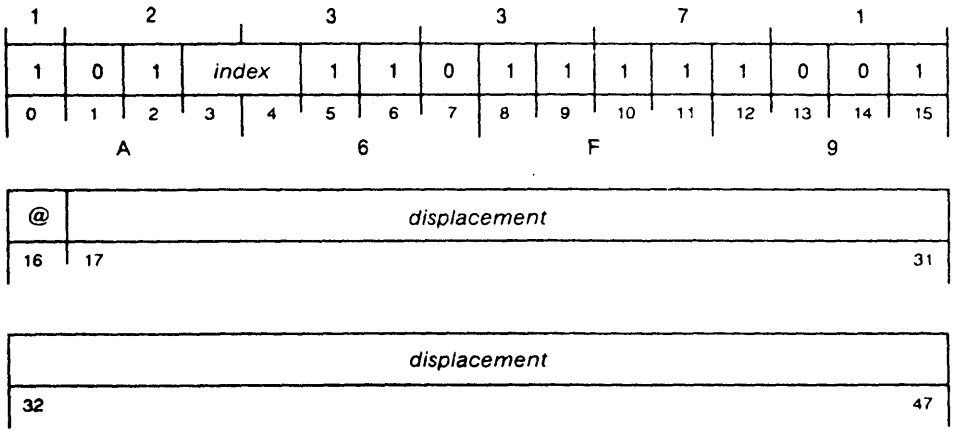
None

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 doubleword of wide stack contains effective address (bit 0 set to 0).

Related Instructions

XPEF Push Address (Extended Displacement)

Exceptions

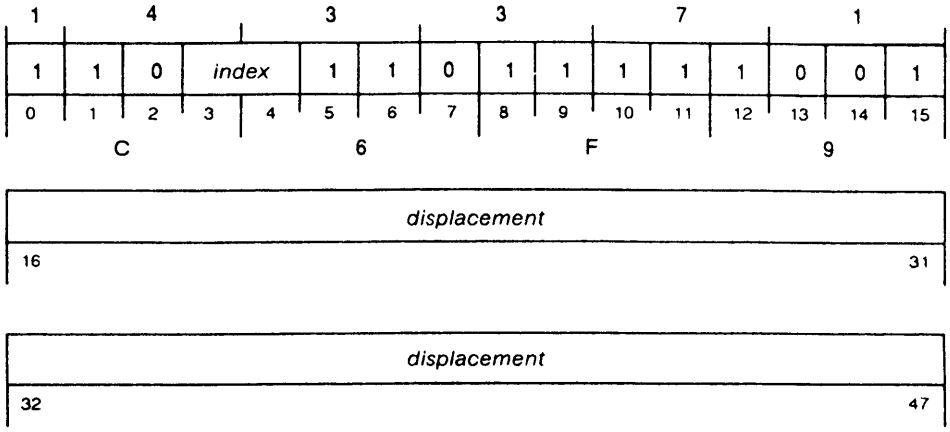
None

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 doubleword of wide stack contains byte address.

Related Instructions

XPEFB	Push Byte Address (Extended Displacement)

Exceptions

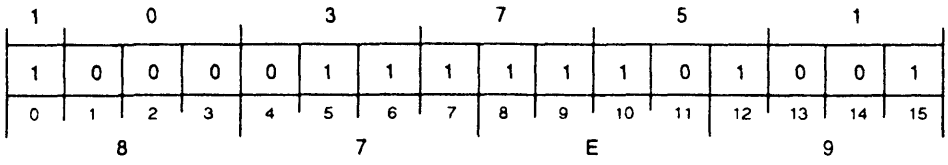
None

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
(fault return)
(normal return)



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 word if no fault occurs.

Arguments

None

Registers, Flags, and Stacks

AC0	After execution, contains last resident pagetable 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	Unused
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 occurs:

Address translator disabled.

Ring field of logical address is less than the current ring field (AC1 contains protection fault code 4).

Invalid or depth fault on SBR check.

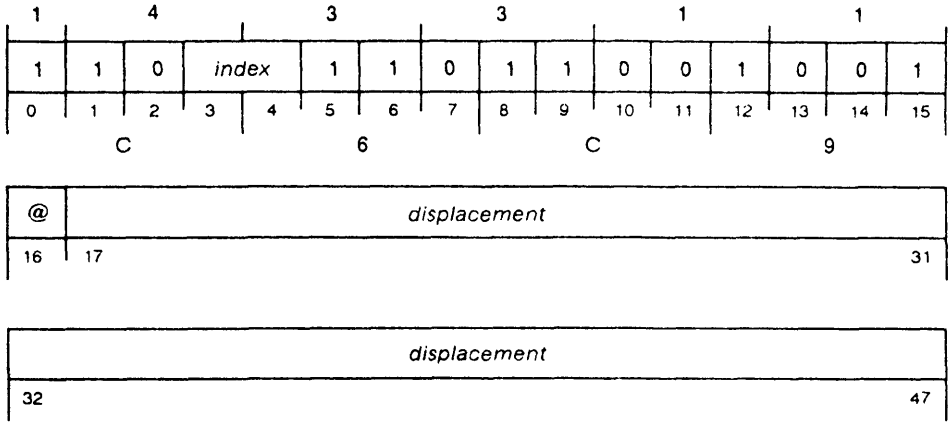
Page or invalid fault on PTE check.

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.

Registers, Flags, and Stacks

AC0-AC3	Unused
Carry	Unchanged
Overflow	0
PC	Effective address
PSR	Unchanged
Stack	Top doubleword 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

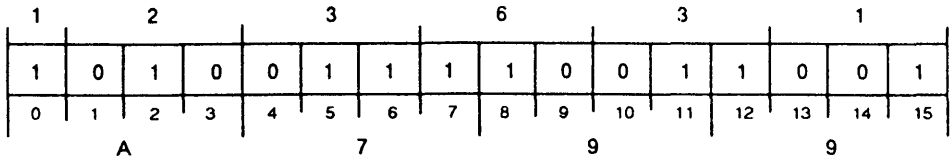
None

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 (PSR) into bits 0–15 of AC0 and fills bits 16–31 of AC0 with zeros. Reserved bits of the PSR are returned in AC0 as 0.

Arguments

None

Registers, Flags, and Stacks

AC0(0–15)	After execution, contains PSR bits (reserved bits and bits 16–31 are zeros).
AC1–AC3	Unused
Carry	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

SPSR	Store Processor Status Register

Exceptions

None

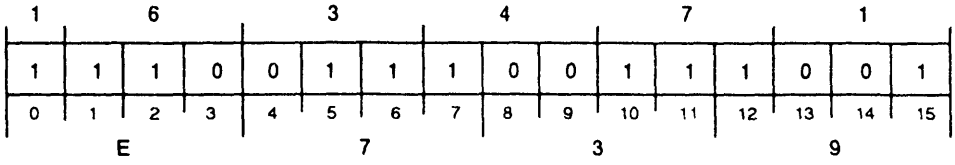
Example

```
LPSR              ;Put the PSR into AC0[0-15].
XWSTA 0,PSR       ;Store the PSR in memory.
```

Load Pagetable 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 → unchanged
AC2 = x → physical address of PTE returned in AC0
AC3 = SBR table address → unchanged

NOTE: The addresses in AC1 and AC3 may not specify indirection.

Exceptions, in order of detection are: ATU off, not in ring 0, invalid SBR, pagetable depth error, invalid second level PTE, nonresident first level PTE.

Conditions not considered errors are: invalid first level PTE, nonresident object page.

LPTE obtains the pagetable entry (PTE) that corresponds to the translation of the logical address contained in AC1. If no exceptions are encountered, AC0 contains the PTE, AC2 contains the physical address of the PTE, the next sequential word is skipped and execution continues.

Arguments

None

Registers, Flags, and Stacks

AC0	After execution, contains pagetable entry.
AC1	Before execution, contains logical address whose PTE is desired (address may not specify indirection). Segment bits (1–3) of address used to index by doublewords into segment base register (SBR) table 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 doublewords. Each doubleword 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 Pagetable 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 segment 0, invalid SBR, pagetable depth error, invalid second-level PTE, nonresident first-level PTE.

The following conditions are not considered errors: invalid first-level PTE and nonresident object page.

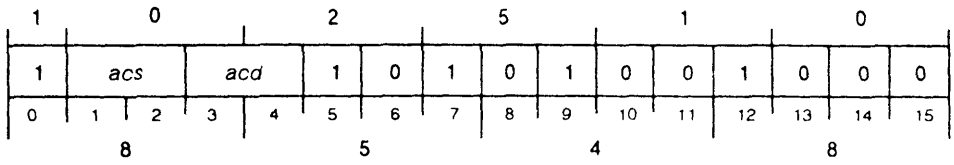
Example

```
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 to *acd*, but 0 → leading 1.

LRB counts the number of high-order zeros in *acs*, 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

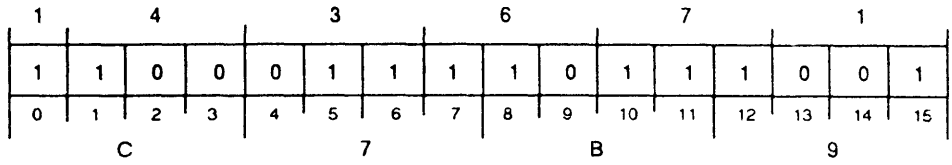
Example

```
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 now enables it.

Arguments

None

Registers, Flags, and Stacks

AC0 — Before execution, contains starting address of an eight-doubleword block containing new SBR values. Values arranged in table and moved to SBRs as follows:

Doubleword 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 doublewords.

If an invalid address is loaded into SBR0, the processor disables the address translator and a protection fault occurs (AC1 contains code 3). This means that logical addresses are identical to physical addresses, and the fault is processed in physical address space.

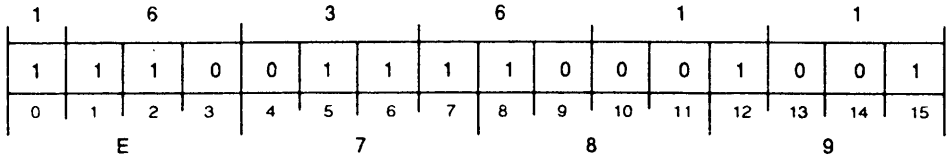
Example

```
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 1–7 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 now enables it.

Arguments

None

Registers, Flags, and Stacks

AC0 — Before execution, contains starting address of a seven-doubleword block containing new SBR values. Values arranged in table and moved to SBRs as follows:

Doubleword 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

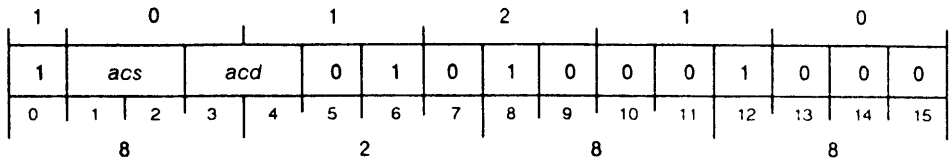
Example

```
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 number is 0, no shifting is performed.

If magnitude is greater than 15, all bits of *acd* are set to 0.

Sign of number determines direction of shift. If bit 24 is

0, shifting is to left.
1, shifting is to right.

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

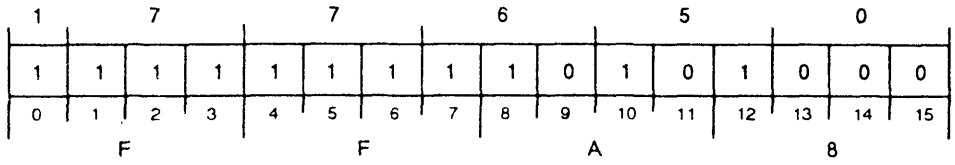
Example

```
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 memory 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 code value as follows:

Code	Value of Number
+1	Positive nonzero
-1	Negative nonzero
0	Positive zero
-2	Negative zero

AC2(16-31) — After execution, contains original contents of AC3 (16-31).

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

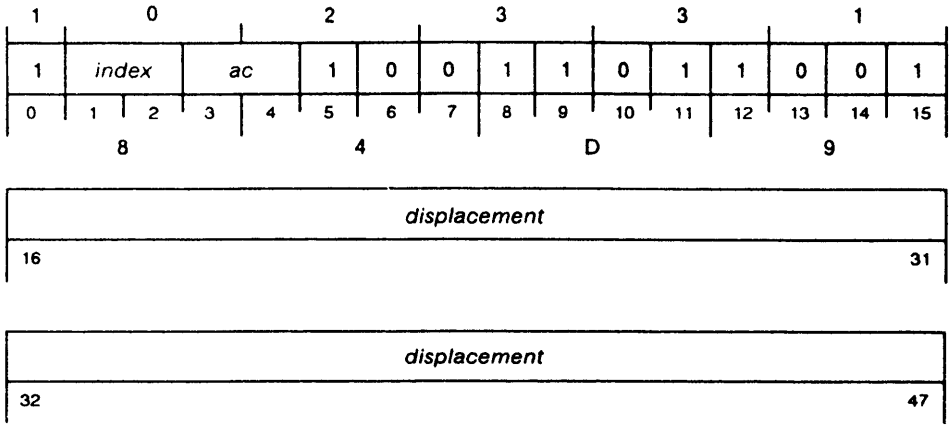
None

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 code into AC1 which reflects sign
                    ;of the 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 *ac* 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

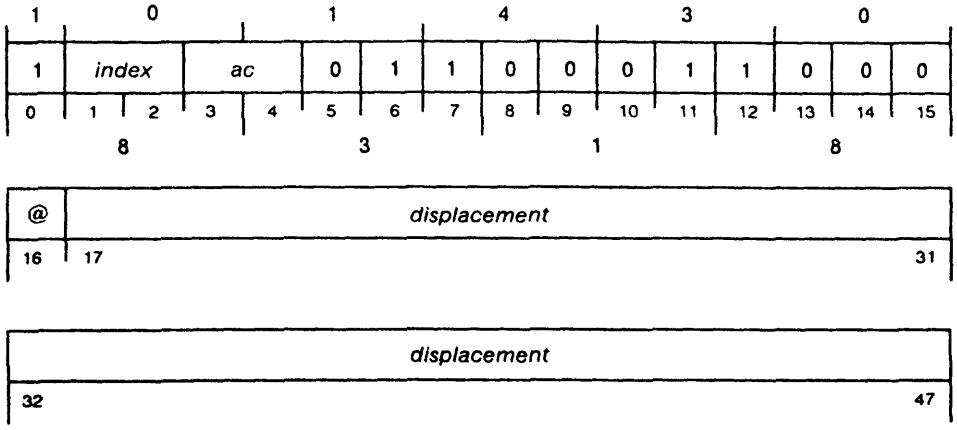
None

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 carry → 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 *ac*.

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

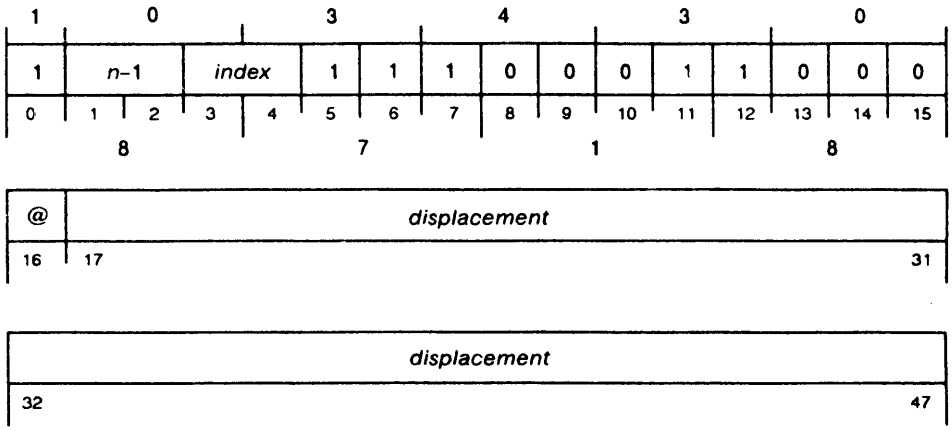
If the result of the add produces a result greater than 2,147,483,647, PSR(OVR) is set to one.

Example

```
LWLDA 0,FIRST     ;Get one value (32 bits).
LWADD 0,SECOND    ;Add the second value (32-bit arithmetic).
LWSTA 0,RESULT    ;Store the doubleword result.
```

Wide Add Immediate (Long Displacement) LWADI

LWADI *n*,[@]*displacement*[,*index*]



Function: *n* + (E) → (E)
ALU carry → CRY

Parameters: None

LWADI adds an integer in the range of 1 to 4 to the signed 32-bit integer contained in memory.

Arguments

n — Integer in range 1 to 4.

Since Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field, you 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

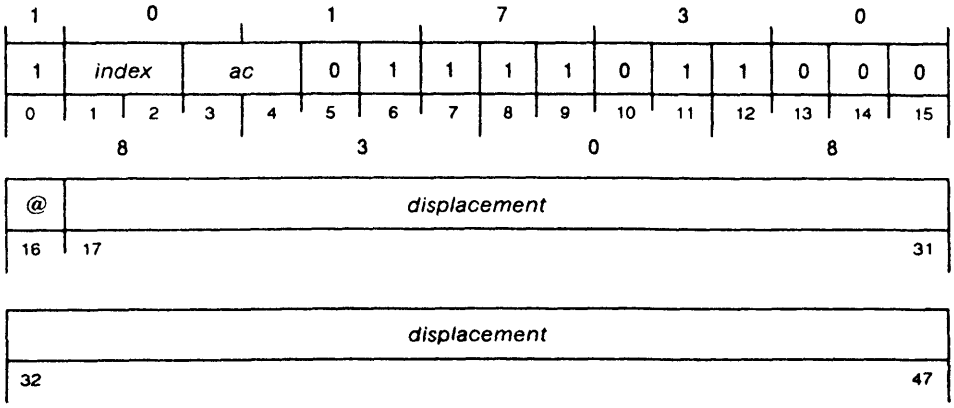
None

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 *ac* to 64 bits. The instruction then divides this value by the signed 32-bit integer contained in memory, placing the quotient into the specified accumulator.

Arguments

ac Before execution, contains signed 32-bit integer.
After execution, contains quotient 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 quotient outside specified range or if memory word is 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

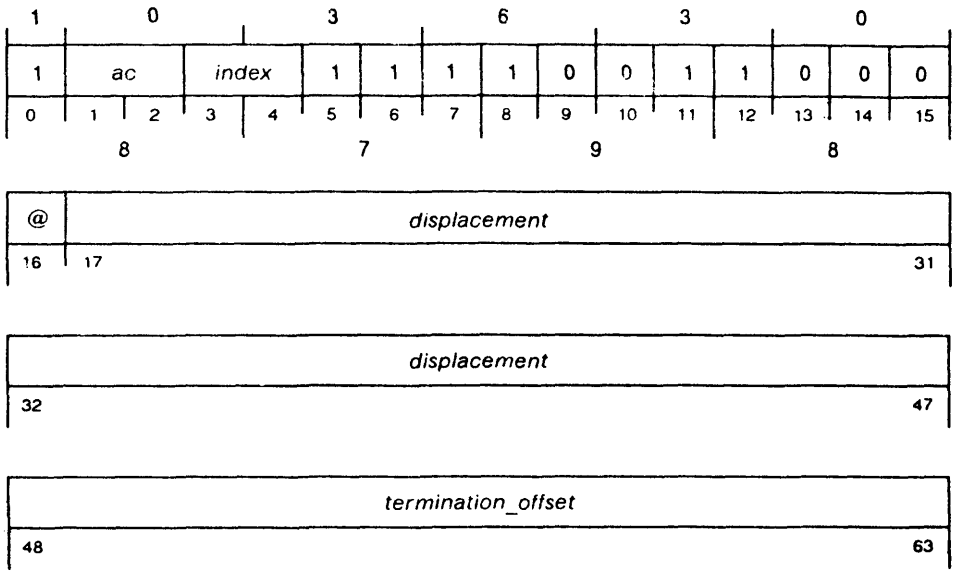
If the quotient is outside the range of -2,147,483,648 to +2,147,483,647 inclusive, or if the memory word is 0, an overflow occurs, PSR(OVR) is set to 1, and *ac* is unchanged.

Example

```
LWLDA 0,DIVIDEND   ;Get the dividend (32 bits wide).
LWDIV 0,DIVISOR    ;Divide by the divisor.
LWSTA 0,RESULT     ;Store the doubleword 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 carry → 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.
- 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 signed PC-relative address for normal return. Argument ranges from 0 to 64 Kwords. (This value is sign-extended to 32 bits for the addition. The final value contains the current segment of execution in bits 1-3.)

[@]*displacement*[,*index*]
: Specifies effective address of a doubleword in memory to be incremented during each pass of DO-loop. Memory doubleword 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	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

LWLDA, XWLDA, WLDAI
: Use these instructions to load *ac* with a value one greater than actual loop count.

WBR
: Use the Wide Branch instruction to end the DO-loop (to loop back to the **LWDO** instruction).

Exceptions

In any return block, the contents of the specified memory location and the program counter value are both undefined.

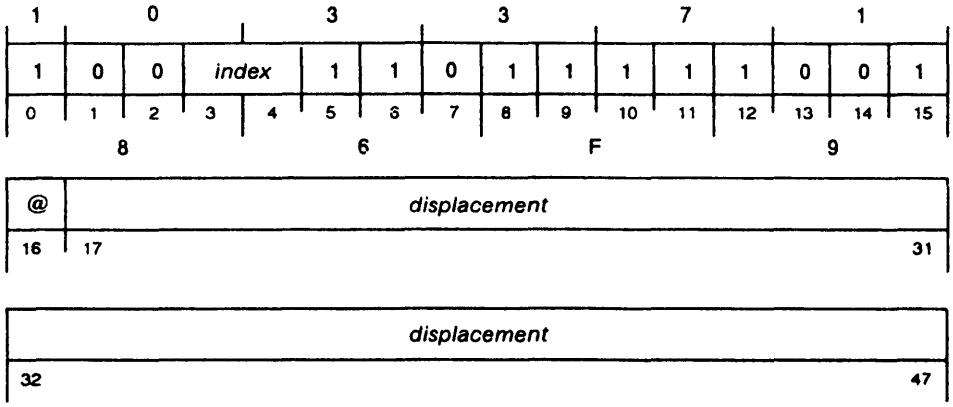
If a fixed-point overflow fault occurs while incrementing the DO-loop variable, the contents of memory and the PC value in the return block are undefined. (AC0 will contain the address of the DO-loop instruction.)

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 1 the unsigned 32-bit integer in this location. If the result is equal to 0, 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 doubleword 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

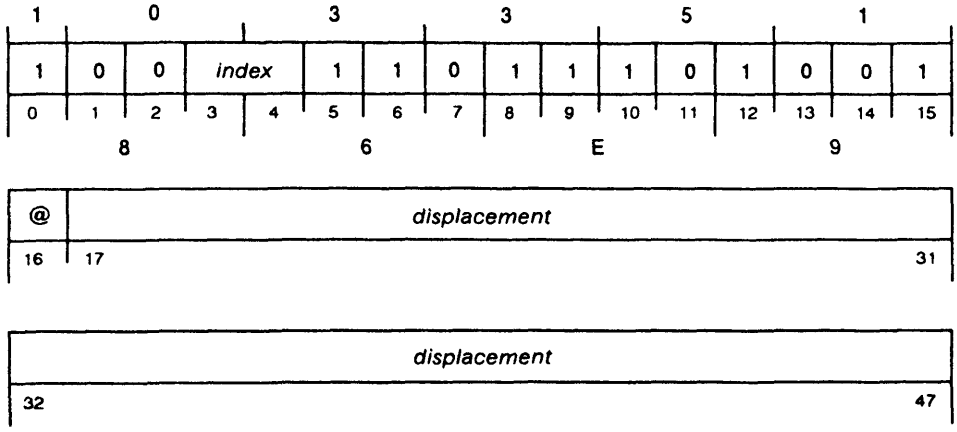
None

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 1 the unsigned 32-bit integer in this location. If the result is equal to 0, 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 doubleword 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

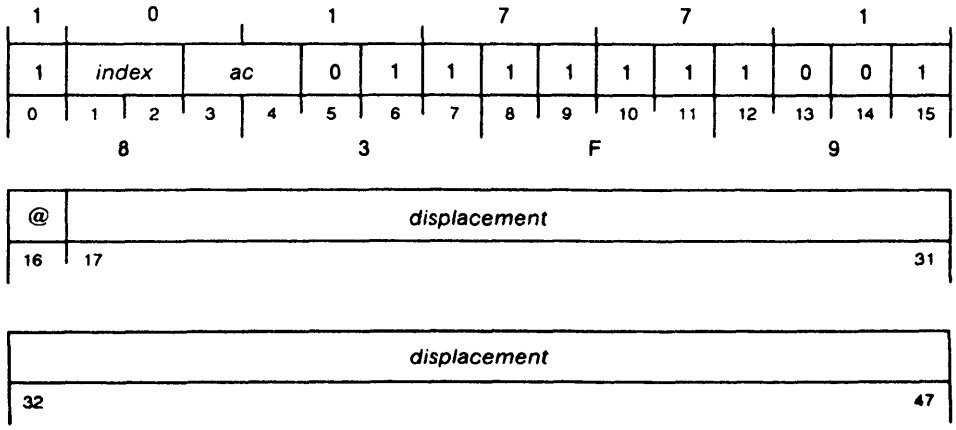
None

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

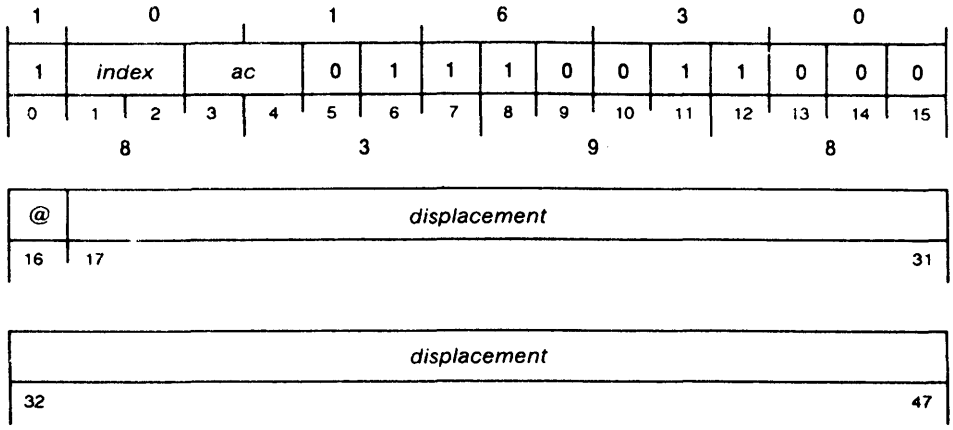
None

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 memory by the signed 32-bit integer in *ac*. Then it loads the least significant 32 bits of the result into *ac*.

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

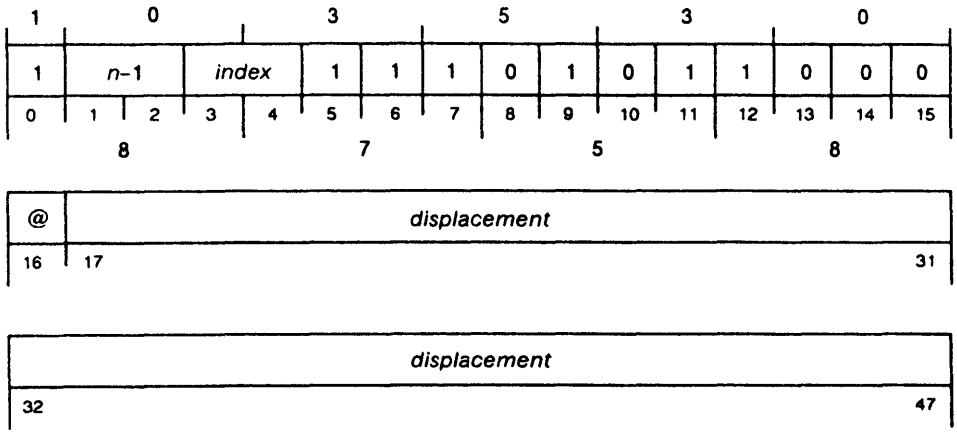
If the result is outside the range -2,147,483,648 to +2,147,483,647 inclusive, an overflow occurs, and PSR(OVR) is set to 1.

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 doubleword result.
```

Wide Subtract Immediate (Long Displacement) LWSBI

LWSBI *n*,[@]*displacement*[,*index*]



Function: (E) - *n* → (E)
ALU carry → CRY

Parameters: None

LWSBI subtracts an integer in the range of 1 to 4 from the signed 32-bit integer contained in memory.

Arguments

n — Integer in range 1 to 4.
Since Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field, you 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

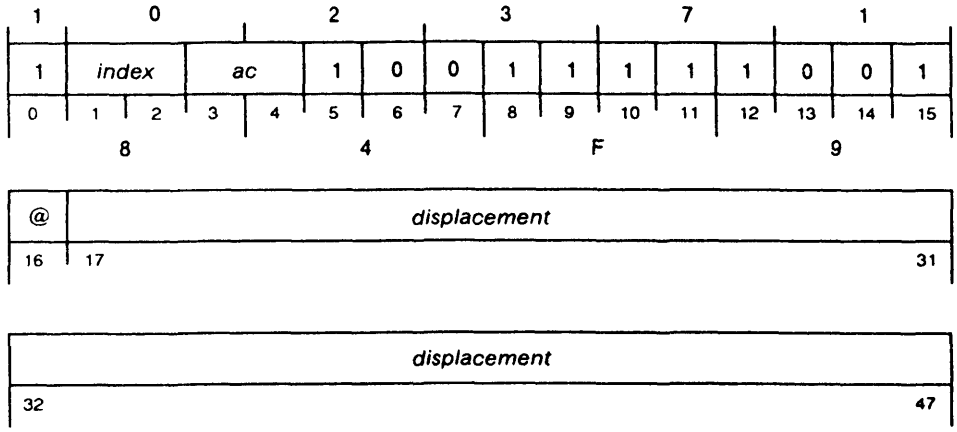
None

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 *ac* into this location.

Arguments

ac Before execution, contains 32-bit value.

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

LWLDA Wide Load Accumulator (Long Displacement)

Exceptions

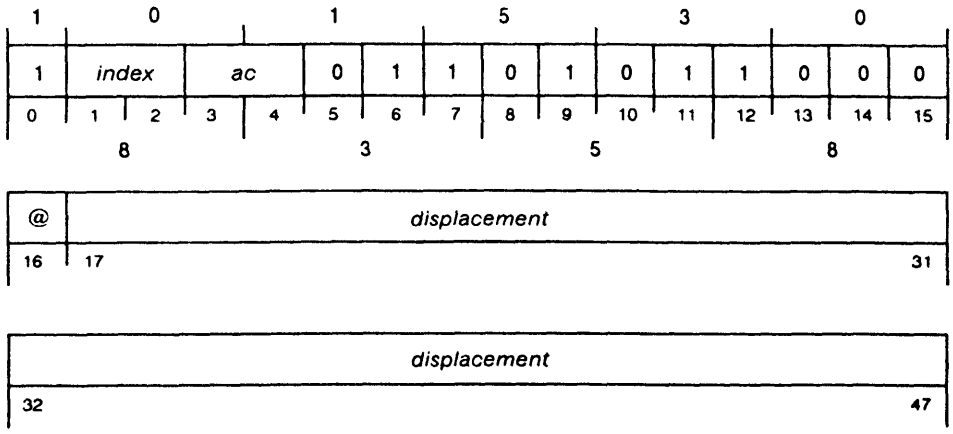
None

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 carry → CRY

Parameters: None

LWSUB subtracts the signed 32-bit integer contained in the memory from the signed 32-bit integer contained in *ac*. Then it loads the result into *ac*.

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

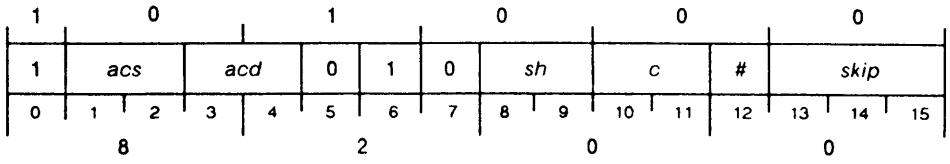
Example

```
LWLDA 0,FIRST      ;Get one value (32 bits).
LWSUB 0,SECOND     ;Subtract the second value (32-bit arithmetic).
LWSTA 0,RESULT     ;Store the doubleword 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*, 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*, 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, 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 tests true. Following table gives test conditions as specified by bits 13 through 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

A 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).

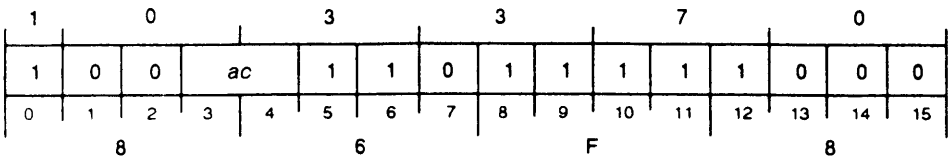
Example

```
    ;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 *ac* 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 the 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 narrow stack fault routine, and control transfers to the stack fault routine.

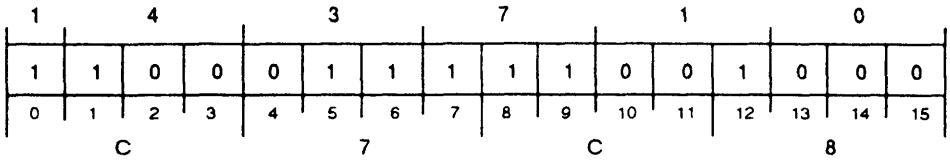
Example

```
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, in AC1 and AC2, and adds the 32-bit intermediate result to an unsigned 16-bit integer in AC0. The result is an unsigned 32-bit integer in AC0 and AC1.

Arguments

None

Registers, Flags, and Stacks

AC0(16-31)	Before execution, contains unsigned 16-bit number to be added to intermediate 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

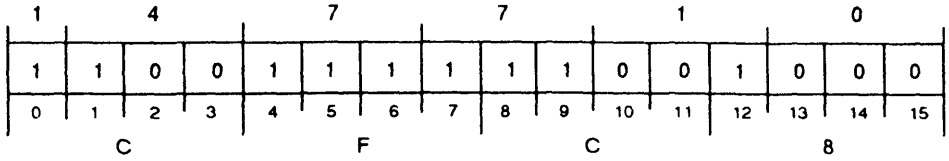
Example

```
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, in AC1 and AC2, and adds the 32-bit intermediate result to the signed 16-bit contents of AC0. The result is a signed 32-bit integer in AC0 and AC1.

Arguments

None

Registers, Flags, and Stacks

AC0(16-31)	Before execution, contains signed 16-bit integer to be added to intermediate 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

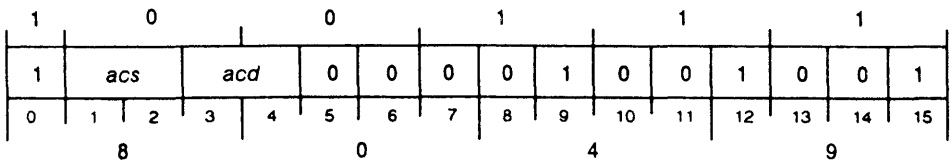
None

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

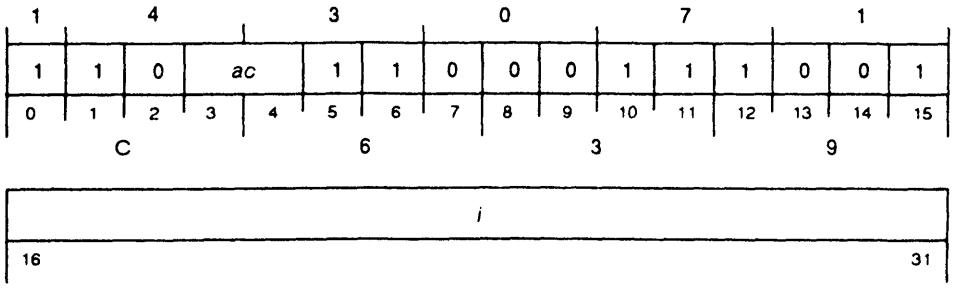
None

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: $i + ac \rightarrow ac$

Parameters: None

NADDI adds a signed 16-bit integer contained in the immediate field to the signed 16-bit integer in *ac* and stores the 32-bit sign-extended result in *ac*.

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

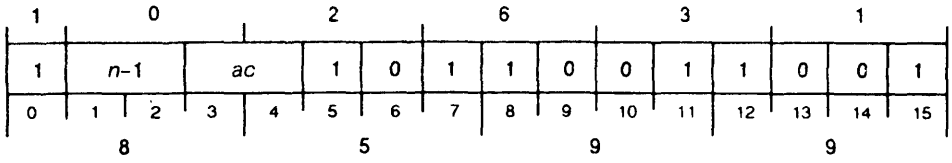
None

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: $n + ac \rightarrow ac$

Parameters: None

NADI adds an integer in the range of 1 to 4 to the signed 16-bit integer in *ac*, sign-extending the result 32 bits.

Arguments

n	Integer in range 1-4. Since Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field, you 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

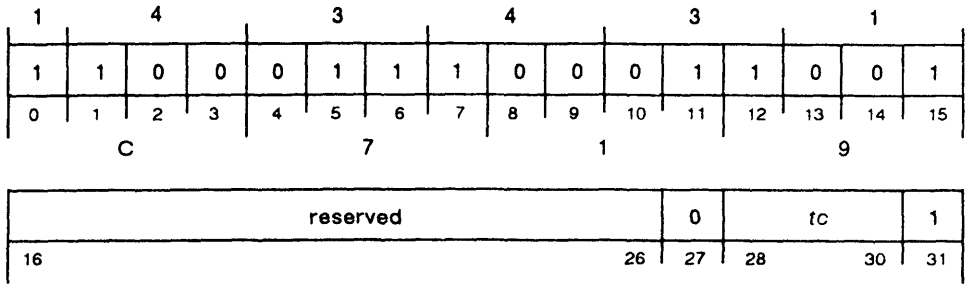
None

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 (in 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 (on the wide stack). 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 of instruction 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. After execution, 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 (only after unsuccessful search or another interrupt), 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, bits 16-31 of top wide stack doubleword 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

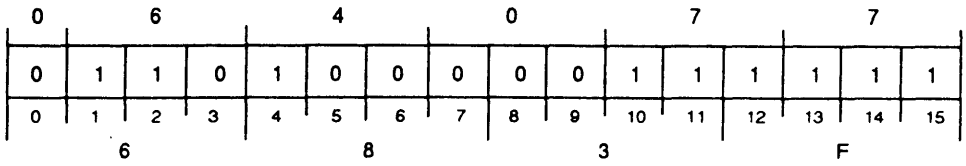
An invalid address (produced by the contents of either or both of AC1 and AC3) may cause a protection fault. The fault code returned to AC1 is dependent on the type of fault.

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 bits 16 through 31 of three accumulators (LEF mode and I/O protection must be disabled).

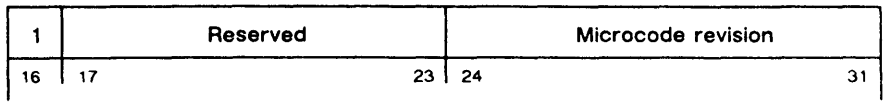
Arguments

None

Registers, Flags, and Stacks

AC0(16–31) After execution, contains model number (binary value of processor's allocated model number). Bits 0–15 are undefined.

AC1(16–31) After execution, contains current microcode revision. Bits 0–15 are undefined. Accumulator format is



Bits	Name	Contents or Function
16	1	Always set to 1
17–23	Reserved	Reserved for future use and returned as zeros.
24–31	Microcode revision	Current microcode revision

If AC1 contains 177777_8, load microcode.

AC2(16–31) After execution, contains memory size (amount of physical memory available, measured in 32-Kbyte modules with an origin of 0). Bits 0–15 are undefined.

Note that the actual memory size in bytes is equal to
$(AC0(16\text{–}31) + 1) * 32768_{10}$
For example, 32_8 indicates 1 Mbyte; 64_8 indicates 2 Mbytes..

NOTE: *Systems which contain 2 Gbytes or more of physical memory return 177777_8 to bits 16–31 of AC0.*

AC3 Unused

Carry Unchanged

Overflow 0

PC PC + 1

PSR Unchanged

Stack Unchanged

Related Instructions

ECLID, LCPID Return CPU identification information.

Exceptions

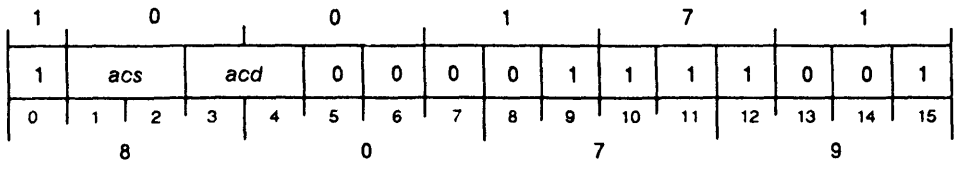
None

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 0, an overflow occurs; PSR(OVR) is set to 1, and *acd* remains unchanged.

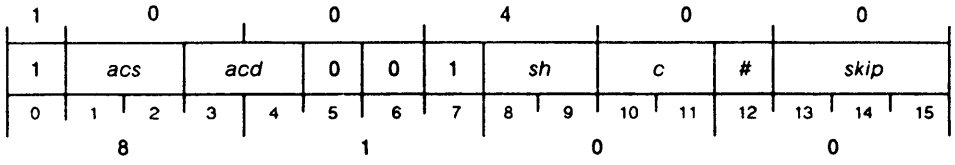
Example

```
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: $-acs \rightarrow acd$

Parameters: None

NOTE: If $\overline{acs} = 0$, CRY → CRY

NEG initializes Carry to the specified value and places the negative of the signed 16-bit integer in *acs* into the shifter. The instruction then performs the specified shift operation, placing the result in *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*, 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*, 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, 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 tests 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

A 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)

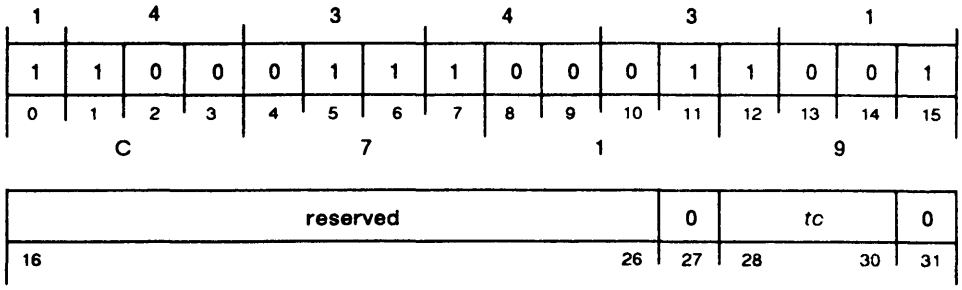
Do not specify **NEG** with no-load option (#) in combination with either never skip or always skip option. Instruction may not end in 1000_2 or 1001_2 (reserved for other instructions).

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 (in 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 (on the wide stack). 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 of instruction 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. After execution, 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 (only after unsuccessful search or another interrupt), 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, bits 16-31 of top wide stack doubleword 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

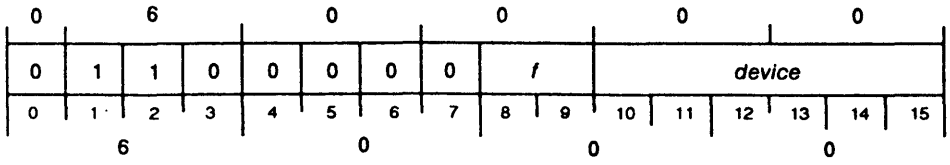
An invalid address (produced by the contents of either or both of AC1 and AC3) may cause a protection fault. The fault code returned to AC1 is dependent on the type of fault.

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 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 1	Set to 0
C	Set to 0	Set to 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

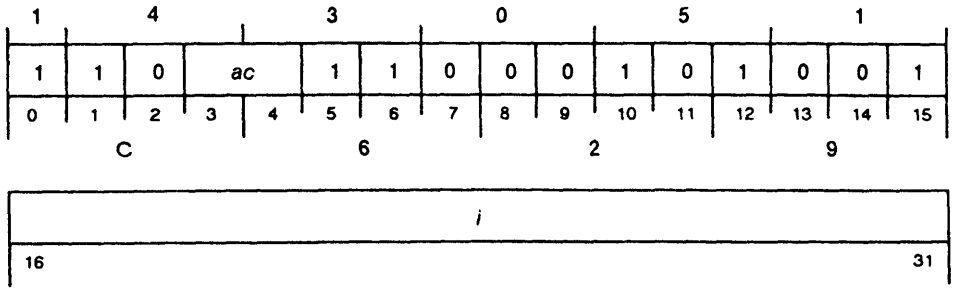
The NIO[*f*] CPU instructions are reserved or assigned specific functions. For instance, NIOS CPU is the Interrupt Enable (INTEN) instruction.

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: $i \rightarrow ac$

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 sign-extended 32-bit 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

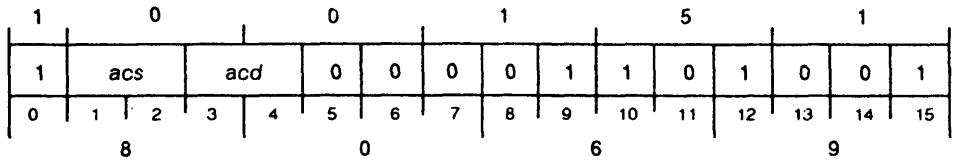
None

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 the 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

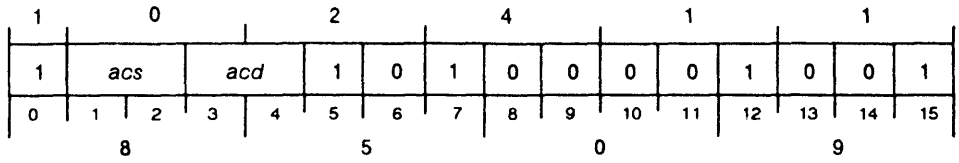
If result exceeds specified range (-32,768 to +32,767 inclusive), PSR(OVR) is set to 1.

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: $-acs \rightarrow acd$

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 0. 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 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

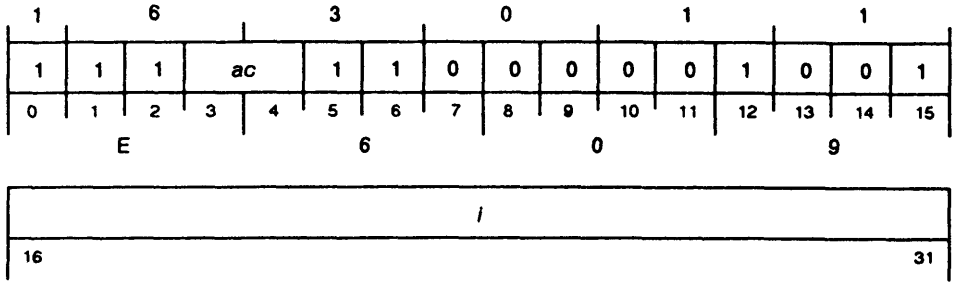
If largest negative 16-bit integer (100000_8) is negated, PSR(OVR) is set to 1.

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 = nonzero return)
(AND = zero return)



Function: If *i* AND *ac* = 0 then skip

Parameters: None

NSALA performs a logical AND of the contents of the 16-bit immediate field with the complement of the least significant 16 bits contained in *ac*. The instruction skips the next word if the result of the AND is 0.

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

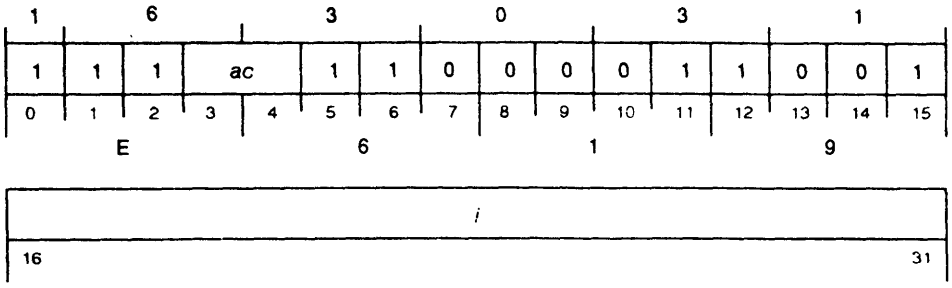
None

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 performs the logical AND of the contents of the 16-bit immediate field with the complement of the word addressed by *ac*. The instruction then skips the next sequential word if the result of the AND equals 0.

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

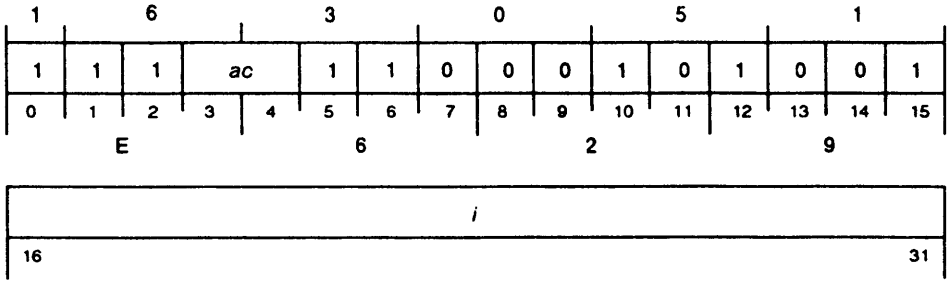
None

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 performs the logical AND of the contents of the 16-bit immediate field with the least significant 16 bits contained in *ac*. The instruction then skips the next sequential word if the result of the AND equals 0.

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

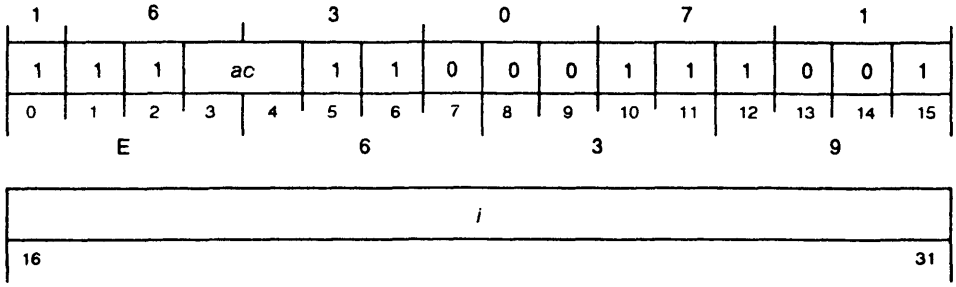
None

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 performs the logical AND of the contents of the 16-bit immediate field with the contents of the word addressed by *ac*. The instruction then skips the next sequential word if the result of the AND equals 0.

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

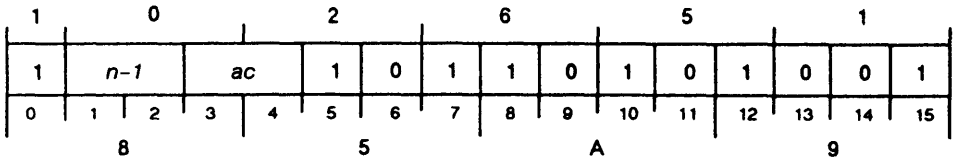
None

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 carry → CRY

Parameters: None

NSBI subtracts an integer in the range of 1 to 4 from the signed 16-bit integer contained in *ac*. Then it stores the result in *ac*, sign-extending it to 32 bits.

Arguments

n	Integer in range 1–4. Since Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field, you 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

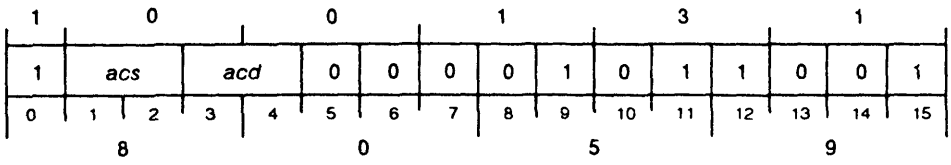
None

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

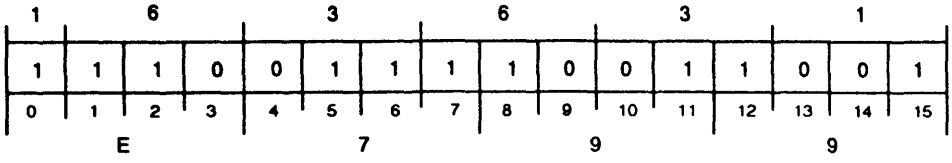
Example

```
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.
```

Purge the Address Translator — PATU

Privileged Instruction

PATU



Function: Purges address translator

Parameters: None

PATU purges the entire address translator of all entries. It should be used only when pagetable 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 Pagetable Entry

Exceptions

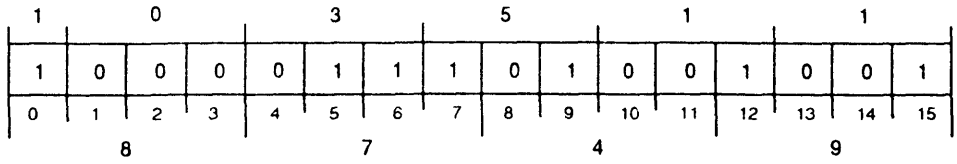
None

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 doublewords (wide return block) → registers, PSR, PC, CRY
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 refer to 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** instruction.

Finally, it temporarily replaces the **BKPT** instruction with the temporarily-saved opcode and continues normal program flow. (The "Subroutine" section in the chapter, "Program Flow Management," provides more information on the **PBX** and **BKPT** instructions.)

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, refers to **BKPT**, effectively substituting 16-bit instruction in AC0 (before pop) for **BKPT** referred to by PC after pop. After execution, initial PC + (opcode length of instruction in AC0).
PSR	If execution of AC0 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 doubleword of wide stack contains value referring to **BKPT** instruction. (If value popped off stack and loaded into PC does not refer to **BKPT** instruction, results undefined.)

Related Instructions

BKPT	Breakpoint

Exceptions

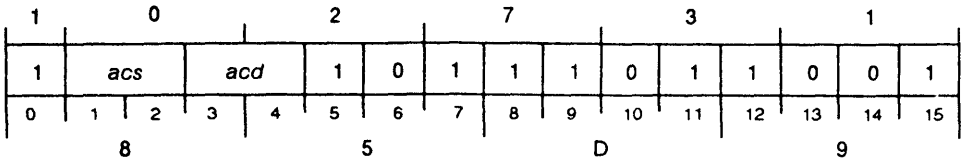
None

Example

```
    .ENT   START        ;Locations 10₈ - 11₈ contain 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 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 loop.
EXIT:     WSUB        2,2         ;End of Program - RETURN TO CALLER
          ?RETURN
    ;     BREAKPOINT HANDLER
BKPTH:    XNLDA       0,OPCOD     ;Load the opcode of the instruction
                                  ;WADD 2,1 into AC0. PBX will
                                  ;cause the execution of this
                                  ;instruction in place of 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. (67₈)
    ;
    ;NOTE: You cannot use the system debugger when you have your
    ;own BKPT handler.
    .END   START
```

Program I/O — PIO

PIO *acs,acd*



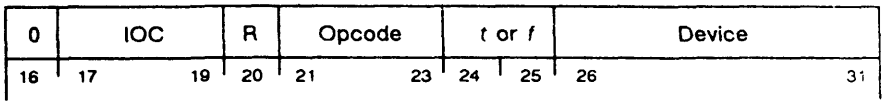
Function: command(*acs*) → device

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. Any data transferred 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	Contents	Function*
16	0	Must be set to 0.
17-19	IOC	I/O channel number (in single-IOC configurations, this value must be 0).
20	R	Reserved and must be set to 0.
21-23	Opcode	Code for I/O operation to be performed. The operations and their codes are DIA 001 DOA 010 DIB 011 DOB 100 DIC 101 DOC 110 NIO 000 SKP*t* 111
24,25	*t* or *f*	Device flag control.
26-31	Device	Device code (refer to appendix, "Standard I/O Device Codes" in machine-specific supplement for a list of these codes).

* 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 operation (read or write). When *acd* is specified as other than AC0, execution of an **NIO** or **SKP***t* 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.

NIO No I/O Transfer

SKP*t* I/O Skip

Exceptions

None

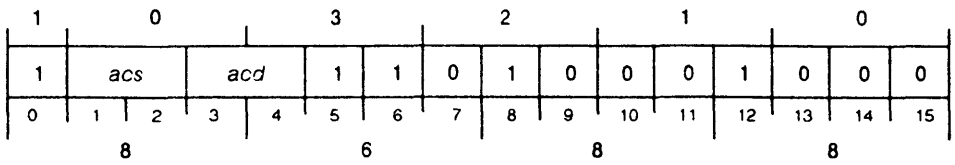
Example

```
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 loaded 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

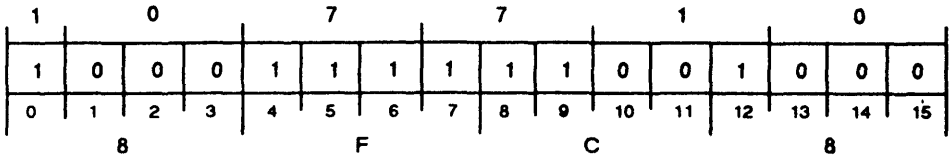
Example

```
PSH   2,0        ;Push bits 16-31 of AC2, AC3, and AC0 onto the
                 ;narrow stack.
. . .
POP   0,2        ;Pop words off stack and restore bits 16-31 of AC0,
                 ;AC2, and AC3 to their values at 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 Extended Operation (**XOP0**) instruction 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(16-31)
3	AC2(16-31)
4	AC1(16-31)
5	AC0(16-31)

Underflow is checked for 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 5 through 2 popped from stack (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.

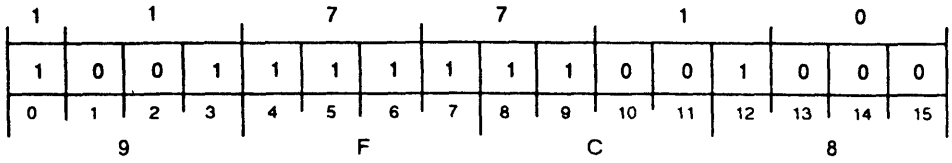
Example

```
POPB        ;Pop 5 words off the stack, loading AC0-AC3, CRY, and 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.

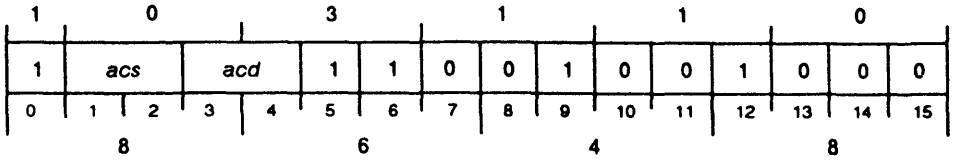
Example

```
    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

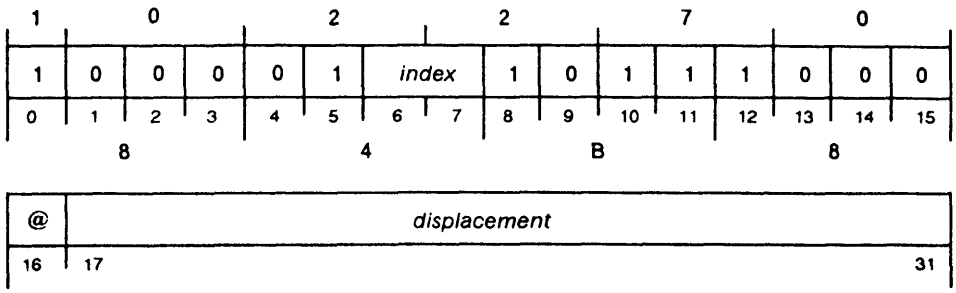
Example

```
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, AC3, and AC2 to their values at time of PSH.
```

Push Jump — PSHJ

ECLIPSE Instruction

PSHJ [@]*displacement*[,*index*]



Function: PC + 2 → 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.

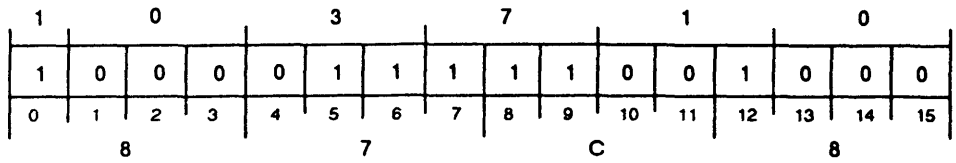
Example

```
    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 two to the current program counter value 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.

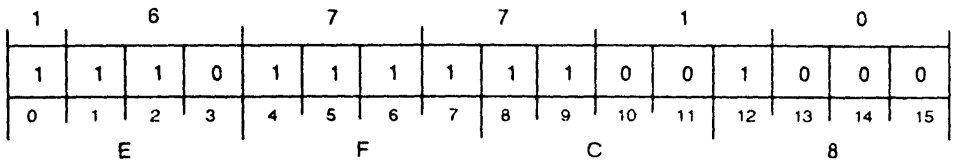
Example

```
    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.
```

Restore RSTR

ECLIPSE Instruction

RSTR



Function: stack → locations
-9 words → stack
1st-5th word → 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(16-31)
3	AC2(16-31)
4	AC1(16-31)
5	AC0(16-31)
6	Narrow stack fault address (page zero location 43_8)
7	Narrow stack limit
8	Narrow frame pointer
9	Narrow 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 5 through 2 popped from stack (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

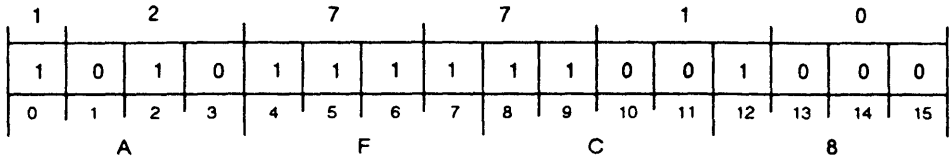
Example

```
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 five words off the narrow stack (narrow return block) and placing them into the following locations:

Word Popped	Destination
1	Bit 0 into Carry; bits 1-15 into the PC
2	AC3(16-31)
3	AC2(16-31)
4	AC1(16-31)
5	AC0(16-31)

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 equivalent).

Arguments

None

Registers, Flags, and Stacks

AC0-AC3	After execution, bits 16-31 contain words 5 through 2 popped from stack (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

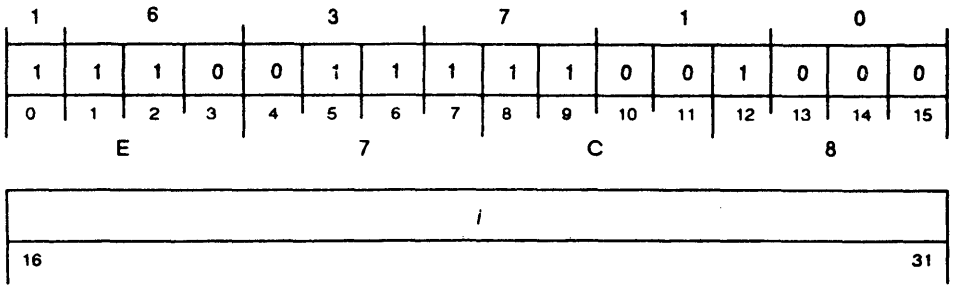
Example

```
SAVZ    ;Save all AC's on the stack
...     ;Perform routine.
RTN     ;Return to caller.
```

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 five-word return block onto the narrow stack. The instruction 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 or 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(16-31)
2	AC1(16-31)
3	AC2(16-31)
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 above 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 execution of the **SAVE** instruction will cause a stack overflow, a stack fault occurs, and the **SAVE** instruction is not executed.

Example

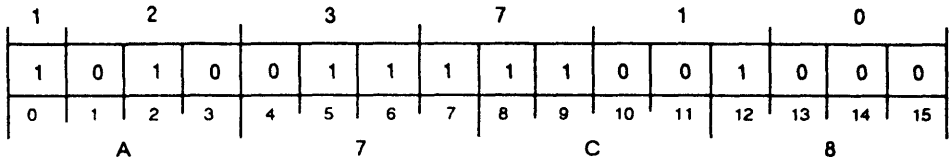
```
    .TITLE SAVE
    .ENT   START, RESULT, XTEL, STKER, RESOK, ARADD, ARAY0, ARAY1
    .ENABLE UWORD
    .LOC   14
    STKER                           ;Stack fault handler.
    .LOC   40
SP:       400                       ;Stack pointer.
FP:       400                       ;Frame pointer.
SL:       436                       ;Stack limit.
          .LOC          401
          .BLK 50                   ;Allocate 50₈ 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 0 then AC1 changed.
          ELDA          3,A2
          SUB#          3,2,SNR
          JMP RESOK                 ;If not 0 then AC2 changed.
                                    ;AC2 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 array lengths 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
```

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(16-31)
2	AC1(16-31)
3	AC2(16-31)
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

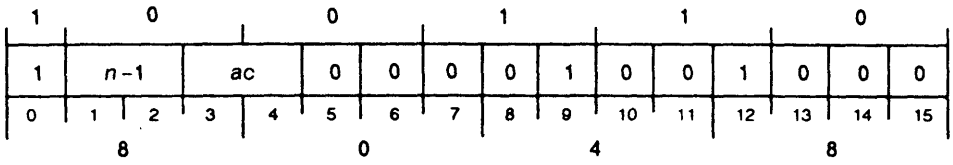
If the execution of the SAVZ instruction will cause a stack overflow, a stack fault occurs, and the SAVZ instruction is not executed.

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 50₈ 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 0 then AC1 changed.
            ELDA            3,A2
            SUB#            3,2,SNR
            JMP             RESOK           ;If not 0 then AC2 changed.
                                            ;AC2 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 (SP - 4).
            STA             1,LCNT          ;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 carry → 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. Since Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field, you 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	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

NSBI	Narrow Subtract Immediate
WSBI	Wide Subtract Immediate

Exceptions

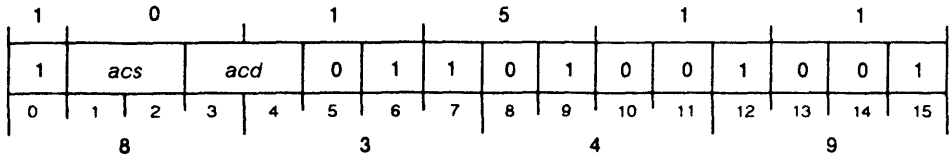
None

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

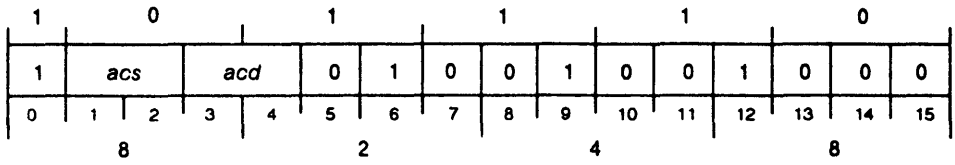
Example

```
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* return)
(*acs* >= *acd* return)



Function: If *acs* > = *acd* then skip

Parameters: None

NOTE: Compares signed numbers.

SGE compares two signed 16-bit integers in two accumulators and skips the next sequential word if the integer in *acs* is greater than or equal to the integer in *acd*.

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

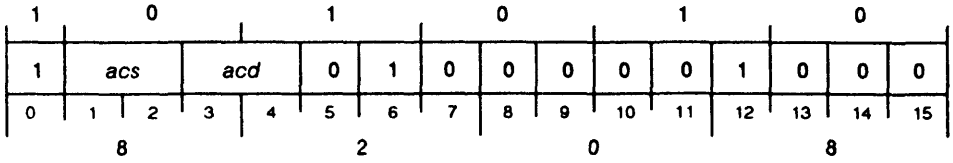
Example

```
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* return)
(*acs* > *acd* return)



Function: If *acs* > *acd* then skip

Parameters: None

NOTE: **Compares signed numbers.**

SGT compares two signed 16-bit integers in two accumulators and skips the next sequential word if the integer in *acs* is greater than the the integer in *acd*.

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

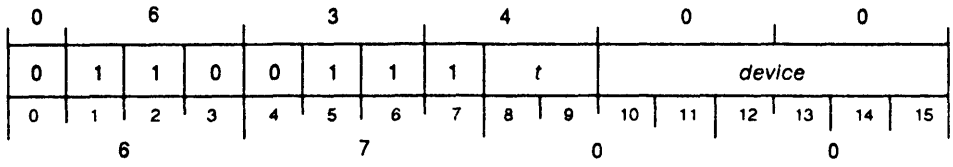
Example

```
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 — SKP*t*

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 word.

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 — Specifies 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

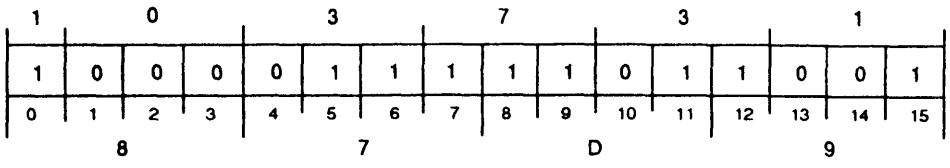
Example

```
        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 = page frame # → unchanged

SMRF stores new values into the modified and referenced bits of the specified page frame.

Arguments

None

Registers, Flags, and Stacks

AC0(30–31)	Before execution contains new values to be stored in modified (bit 30) and referenced (bit 31) bits.
	After execution, contents unchanged.
AC1(13–31)	Before execution specifies page frame number.
	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

The results are undefined if the address translator is not enabled or you specify either a nonexistent page frame or a page frame outside main memory.

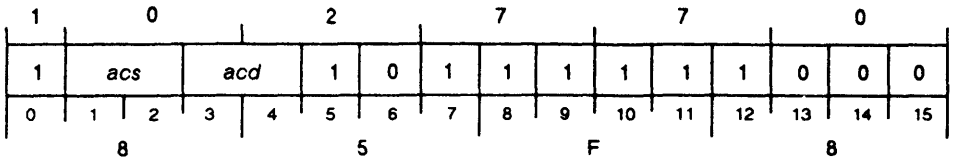
Example

```
WSUB   0,0          ;New modified and referenced bits both 0.
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 = 0 return)
(test bit = 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 the test result is true. The effective address generated by the instruction is confined to the first 64 Kbytes of the current segment.

Arguments

acs(16-31) Contains high-order word of 32-bit address. If *acs* and *acd* are the same accumulator, then the high-order word of the address is 0.

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 = 0)
PC + 2 (test bit = 1)

PSR Unchanged

Stack Unchanged

Related Instructions

SZB Skip on Zero Bit

SZBO Skip on Zero Bit and Set to One

Exceptions

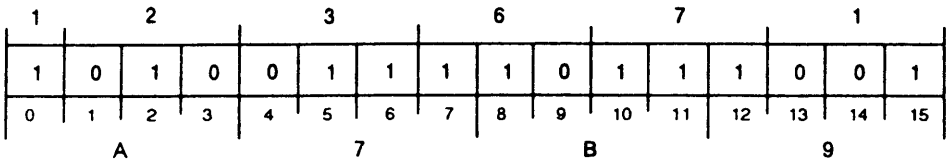
None

Example

```
        ELEF   0,FLAGS      ;Get word address of flags word.
        SUB    1,1          ;Get a 0 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) and skips the next sequential word if OVR is 0.

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

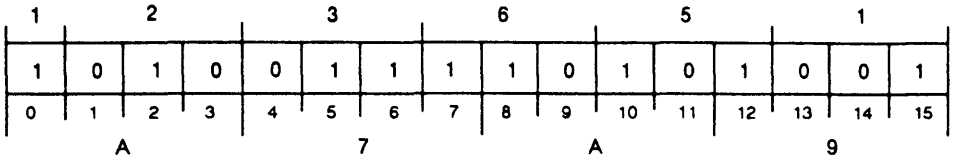
None

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

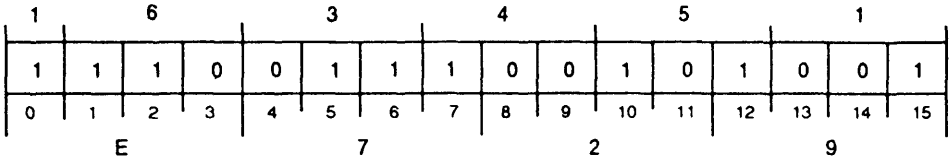
Example

```
XWLDA 0,PSR        ;Get saved PSR from memory.
SPSR               ;Store into the PSR register.
```

Store Pagetable 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 pagetable 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 pagetable entry data.
	After execution, contents unchanged.
AC1	Before execution, contains logical address to use new PTE data. This address 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 Pagetable Entry

LPHY Use this instruction to load a physical address into AC2 and the last resident PTE into AC0.

Exceptions

If the address translation unit is off, then no operation occurs.

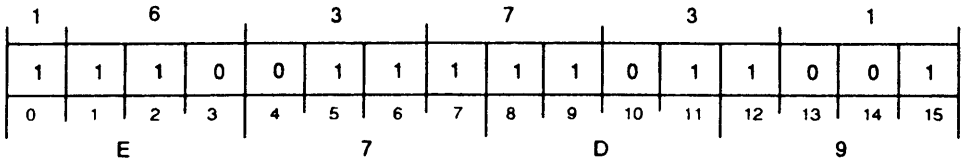
Example

```
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. (Refer to the section, "State Area," in the "System and Memory Management" chapter for further information.)

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, contains physical page-frame number of state area base to be moved to state pointer. (The maximum address is machine-specific.)

After execution, contents unchanged.

AC1 — Before execution, unused.

After execution, contents dependent upon initial value of AC0:

If AC0 contains -1, AC1 contains number of consecutive physical pages that operating system should reserve in memory.

If AC0 contains other than -1, AC1 contains number of state pages assigned.

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 (for instance, 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). The operating system may then perform the move and reload the state pointer by re-executing **SSPT**.

If the processor does not implement a state area, **SSPT** performs no operation.

If you use an **SSPT** instruction in NOVA mode for bootstrapping, bits 0-15 of AC0 and AC1 are undefined.

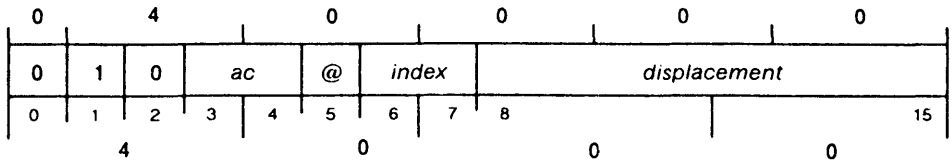
Example

```
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 the specified accumulator into memory.

Arguments

[@]*displacement*[,*index*]
Effective address generated by instruction confined to first 64 Kbytes of current segment.

ac(16-31) Before execution, contains 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
PSR	Unchanged
Stack	Unchanged

Related Instructions

ESTA, LNSTA, LWSTA, XNSTA, XWSTA
Store the contents of an accumulator into memory.

Exceptions

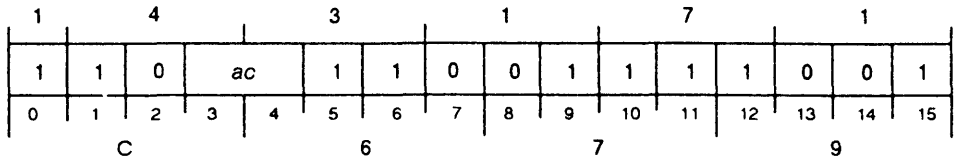
None

Example

```
    STA    0,@ACOUNT     ;Store AC0[16-31] into memory.
    .                    ;This 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 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

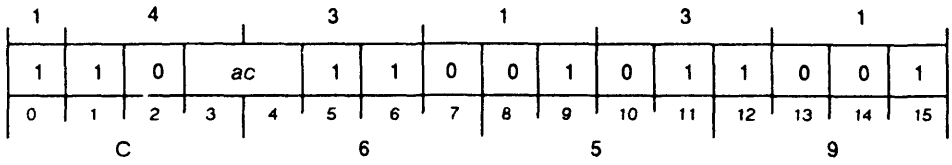
None

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 and 27_8 in page zero of the current segment.

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

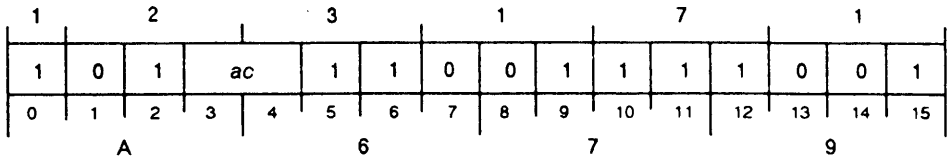
None

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 and 25_8 in page zero of the current segment.

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

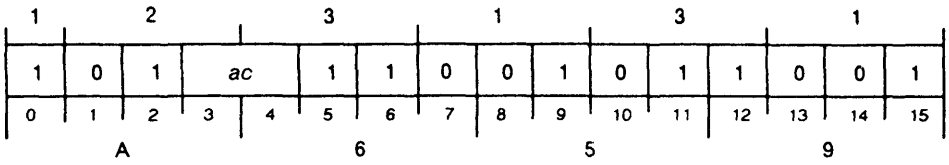
None

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

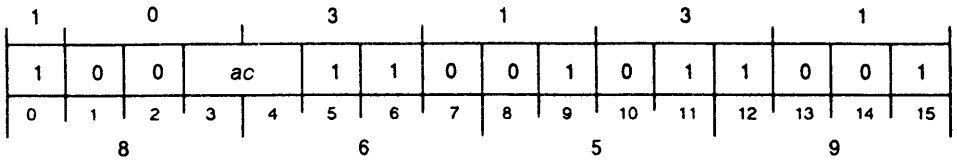
None

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 doubleword and stores the contents of *ac* at this address.

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 Doubleword

Exceptions

None

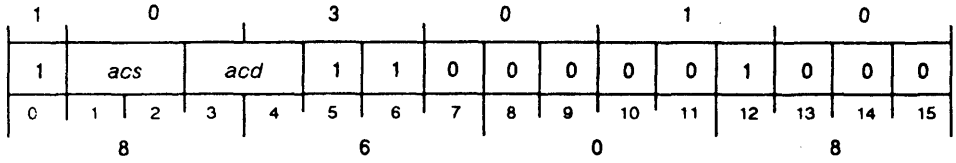
Example

```
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 → unchanged

STB stores a byte from *acd* into memory.

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 stored 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

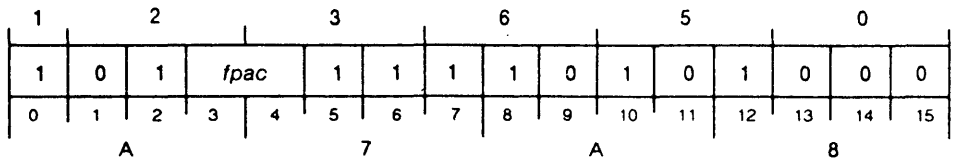
Example

```
    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 the 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 decimal/ASCII fault occurs.

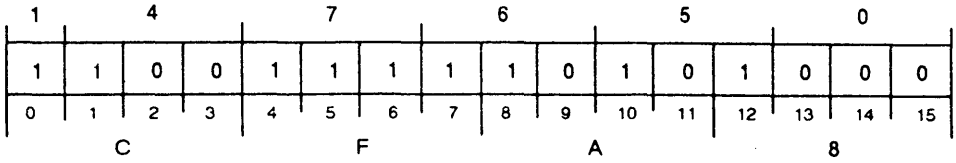
Example

```
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
                    ;an 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 → unchanged
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. It then 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 value to be converted; FPAC0 contains the high 8 digits; FPAC3 contains the low 8 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 decimal/ASCII fault occurs.

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.

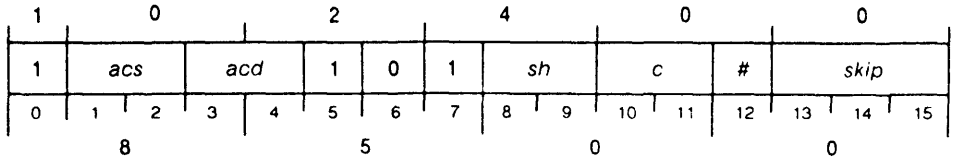
Example

```
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 an 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 initial value of Carry before performing operation (opcode). Following table gives values of *c*, 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*, 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, 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 tests 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

A 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).

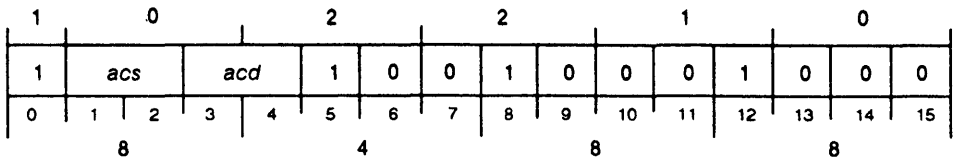
Example

```
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 = 1 return)
(bit = 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 0. 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 *acs* and *acd* are the same accumulator, then high-order word of address is 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 = 1) PC + 2 (test bit = 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

SZBO	Skip on Zero Bit and Set to One
SNB	Skip on Nonzero Bit

Exceptions

None

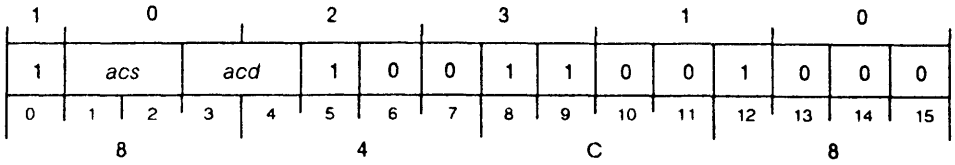
Example

```
      ELEF   0,FLAGS      ;Get word address of flags word.
      SUB    1,1          ;Get a 0 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 = 1 return)
(bit = 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. If the bit is

0, **SZBO** sets the bit to 1 and skips the next sequential word.

1, it remains unchanged and the next sequential word is executed.

SZBO is useful for creating bit maps for such purposes as allocating facilities (memory blocks, I/O devices, etc.) to several processes (or tasks) that may interrupt one another, or in a multiprocessor environment. **SZBO** atomically tests the bit and sets it to 1.

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 *acs* and *acd* are the same accumulator, then high-order word of address is 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 = 1) PC + 2 (test bit = 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

SZB	Skip on Zero Bit
SNB	Skip on Nonzero Bit

Exceptions

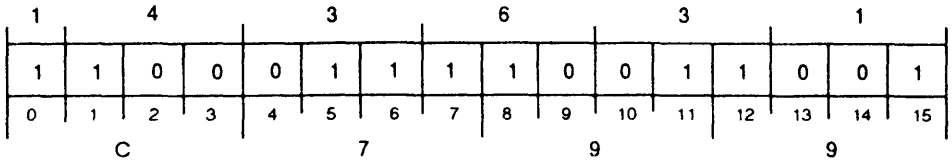
None

Example

```
    ELEF   0,FLAGS      ;Get word address of flags word.
    SUB    1,1          ;Get a 0 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 be 0.
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

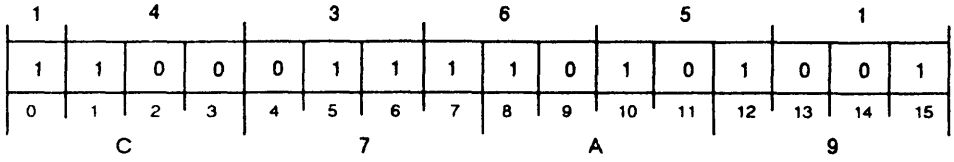
An invalid access (read, write, or execute) generates a protection violation.

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 → resolved address
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 higher segment.
- 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	Before execution, contains 31-bit, indirectable word pointer. After execution, contains resolved address.
AC1(1-3)	Contains segment number; all other bits must be 0.
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

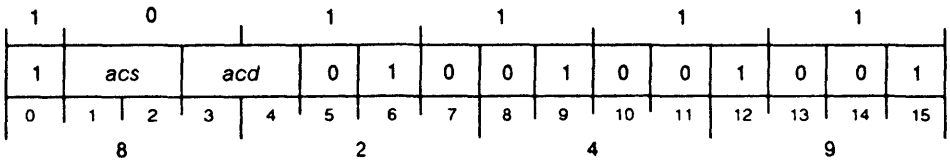
An invalid access (read, write, or execute) or more than the allowable number of indirect addresses generate a protection violation.

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: $\overline{acs} + acd \rightarrow acd$

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 signed 32-bit 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

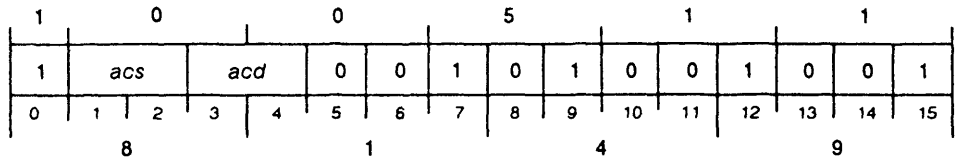
None

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 signed 32-bit 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

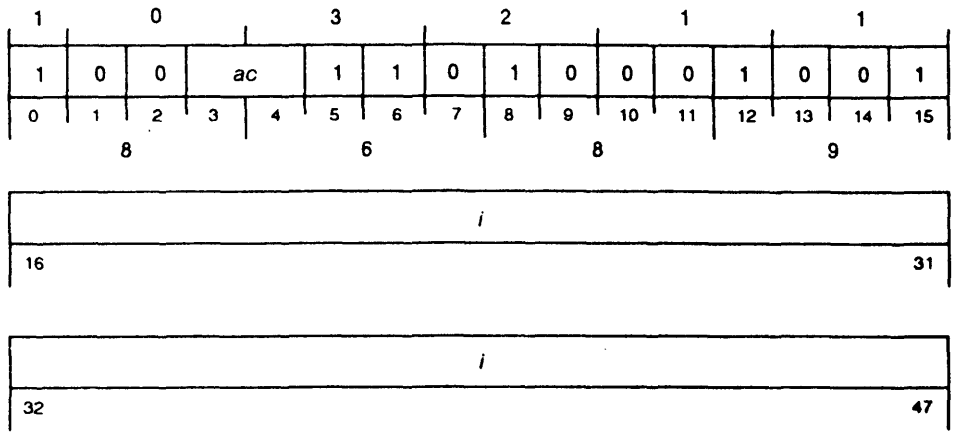
None

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 0.
        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: $i + ac \rightarrow ac$

Parameters: None

WADDI adds the signed 32-bit integer in the immediate field to the signed 32-bit integer in *ac*.

Arguments

i	Contains signed 32-bit integer.
ac	Before execution, contains signed 32-bit integer.
	After execution, contains signed 32-bit 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

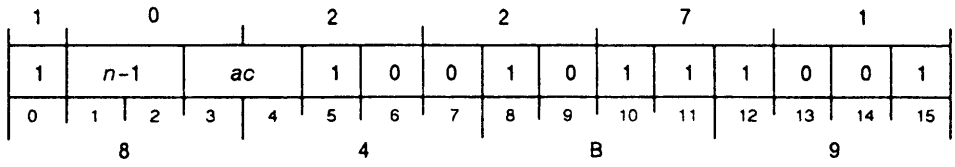
None

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: $n + ac \rightarrow ac$

Parameters: None

WADI adds an integer in the range 1–4 to the signed 32-bit integer contained in *ac*.

Arguments

n	Integer in range 1–4. Since Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field, you should code exact value to be added.
ac	Before execution, contains signed 32-bit integer. After execution, contains signed 32-bit 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

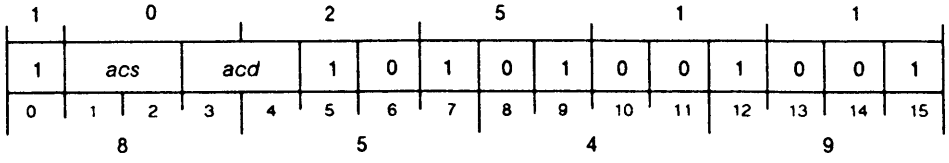
None

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 performs a logical AND of this value with the contents of *acd*, placing the result 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 value.
	After execution, contains 32-bit 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

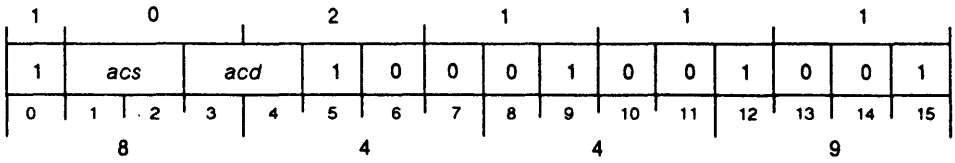
None

Example

```
XWLDA  0,FLAGS              ;Get the flags doubleword.
WLDAI  1B3+1B27+1B29,1      ;Get doubleword with bits 3, 27, 29 set.
WANC   0,1                  ;AND with complement of flags doubleword.
WSEQ   1,1                  ;If the result is 0, 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

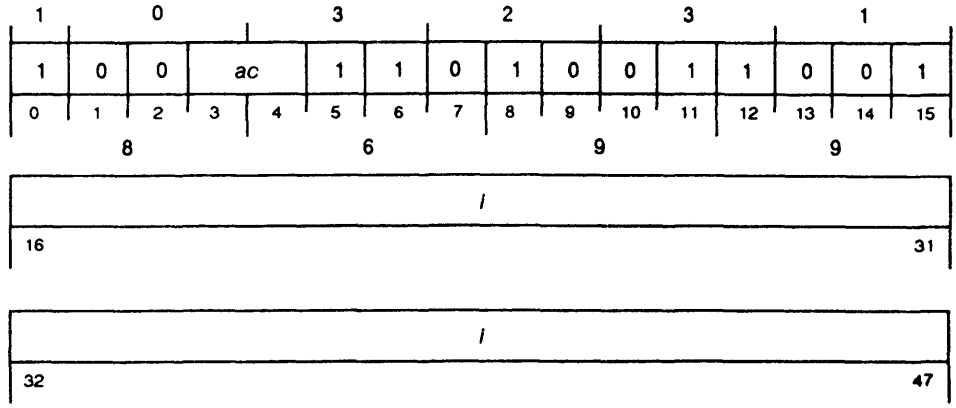
None

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 *ac* and the value in the immediate field, placing the result in *ac*.

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

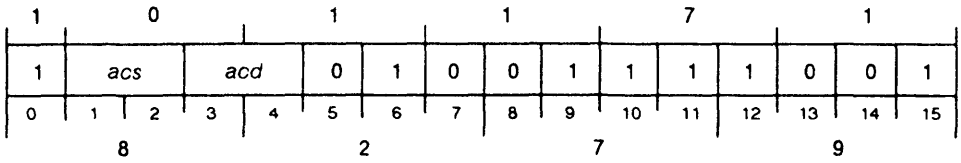
None

Example

```
    ;Convert lowercase input to uppercase.
    ;
    ;       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 lowercase
        "A+40                   ;Lower limit for compare
        "Z+40                   ;Upper limit for compare
        WBR NOTLOW              ;Not lowercase
        WNADI       -40,2       ;Yes, lowercase, convert to uppercase.
```

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*. (This instruction provides the capability of multiplying or dividing by a power of two.)

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 0, 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 (the sign remains unchanged following a shift).

Negative values shifted right are rounded towards 0. For instance, -3 shifted right one position is -1.

Registers, Flags, and Stacks

AC0-AC3	Can be individually specified as *acs* and *acd*; otherwise unused.
Carry	Unchanged
Overflow	0
PC	PC + 1
PSR	OVR is set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

WMOVR	Wide Move Right
WHLV	Wide Halve
WASHI	Wide Arithmetic Shift With Narrow Immediate

Exceptions

In a left shift, if the bit whose value is the complement of the sign bit of *acd* is shifted out, an overflow occurs, and PSR(OVR) is set to 1; the contents of *acd* are undefined, but the lower bits (16–31) contain a correct result.

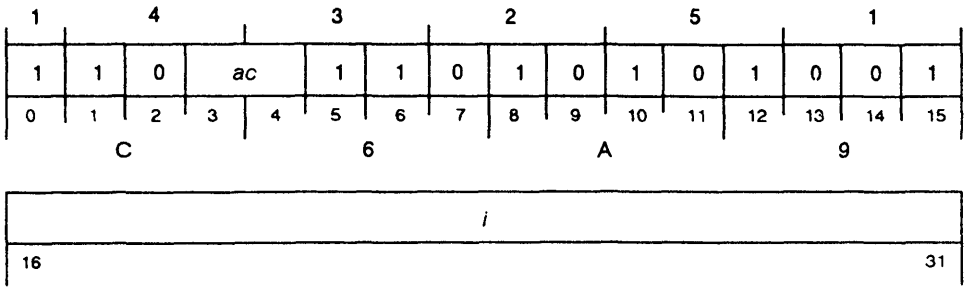
If an arithmetic overflow occurs, the contents of *acd* are undefined.

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 $ac(i[+ = left, - = right]) \rightarrow ac$

Parameters: None

WASHI shifts the contents of *ac* 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 0), and sign-bit fills vacated bit positions.

If *i* is 0, no shifting occurs.

ac Before execution, contains value to be shifted.

After execution, contains result (the sign remains unchanged following a shift).

Negative values shifted right are rounded towards 0. For instance, −3 shifted right one position is −1.

Registers, Flags, and Stacks

AC0–AC3	Can be individually specified as *ac*; otherwise unused.
Carry	Unchanged
Overflow	0
PC	PC + 2
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

WMOVR	Wide Move Right
WHLV	Wide Halve
WASH	Wide Arithmetic Shift

Exceptions

In a left shift, if the bit whose value is the complement of the sign bit of *acd* is shifted out, an overflow occurs, and PSR(OVR) is set to 1; the contents of *acd* are undefined, but the lower bits (16–31) contain a correct result.

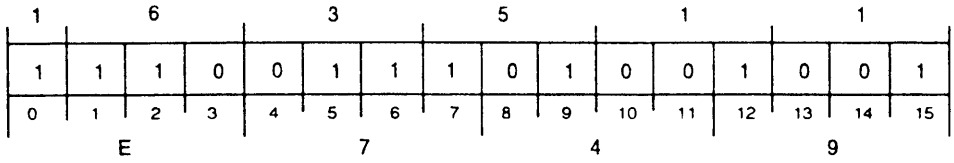
If an arithmetic overflow occurs, the contents of *ac* are undefined.

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 moved.

WBLM moves a number of memory words in consecutive (ascending or descending) order from a source location to a 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. If 0, no words moved. With each word moved, count increments (if negative number) or decrements (if positive number). After execution, contains 0.
AC2	Before execution, specifies source location in memory. With each word moved, value increments (if ascending) or decrements (if descending) by 1. 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 (if ascending) or decrements (if descending) by 1. After execution, contains pointer to next word after last word moved.
Carry	Unchanged
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

BLM	Block Move
BAM	Block Move and Add

Exceptions

Because **WBLM** may have a long execution time, it is interruptible. If the instruction is interrupted, the program counter is decremented by 1 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.

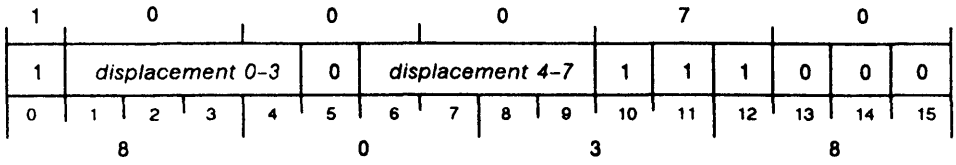
If the contents of AC2 or AC3 produce an invalid address, a protection fault may occur (even if no words are to be moved — AC1 contains 0).

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 refer to a location in current segment).

PSR Unchanged

Stack Unchanged

Related Instructions

JMP Jump

Exceptions

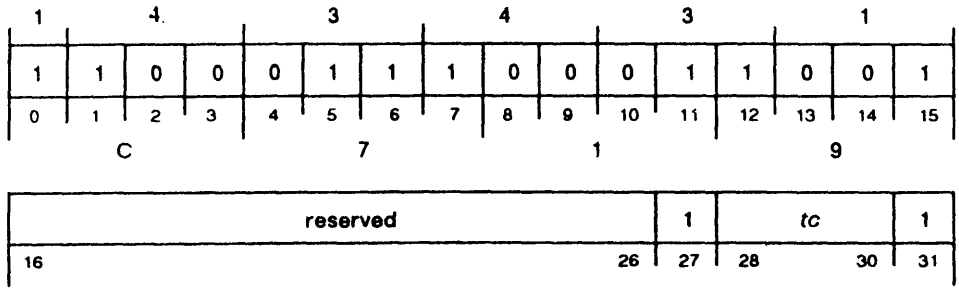
None

Example

```
;This subroutine removes an element from a linked list queue. It is the
;responsibility of the caller to set the transition bit, if necessary.
;
;Calling conventions:          XJSR PDEQ
;                              <return>
;      AC1 = Queue descriptor address.
;      AC2 = Element to be removed.
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 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) → 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

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 (in 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 (on the wide stack). 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 of instruction 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 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. After execution, 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 (only after unsuccessful search or another interrupt), 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 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 doubleword 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

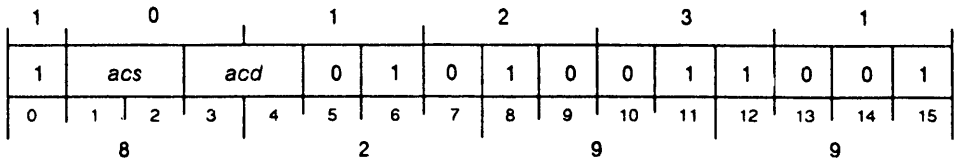
An invalid address (produced by the contents of either or both of AC1 and AC3) may cause a protection fault. The fault code returned to AC1 is dependent on the type of fault.

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 the same accumulator, the processor assumes the word address is 0 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

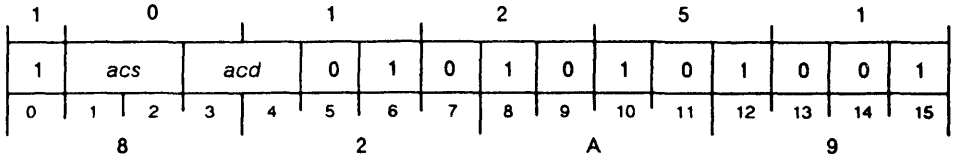
None

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 the same accumulator, the processor assumes the word address is 0 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

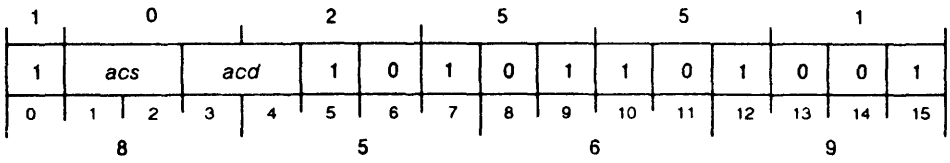
None

Example

```
    ;This subroutine removes an element from a linked list queue. It is
    ;the responsibility of the caller to set the transition bit, if
    ;necessary.
    ;
    ;Calling conventions:          XJSR PDEQ
    ;                              <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# → unchanged

NOTE: If *acs* is not *acd*:
@(*acd*) = 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 executes.

The specification of *acd* determines the location of L and H.

Arguments

acs — Contains signed 32-bit integer for comparison.

acd — Specification of *acd* determines location of L and H.

If specification is different from *acs*, *acd* contains address of lower limit doubleword value, L; higher limit value, H, is contained in next doubleword location following L.

If specification is same as *acs*, limits L and H are in next two respective doubleword 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

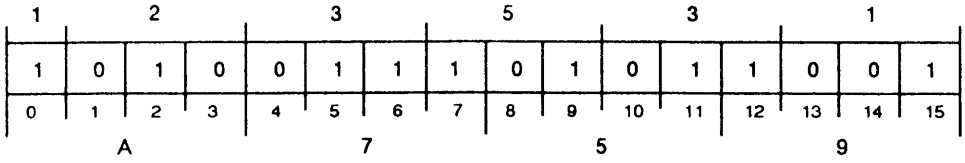
None

Example

```
    ;Convert lowercase input to uppercase.
    ;
    ;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 lowercase.
         WNADI       -40,2        ;Yes, lowercase, convert to uppercase.
         . . .
NOTLOW:  . . .
```

Wide Character Compare WCMP

WCMP



Function: String 1 ?=? string 2
Result → 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 the instruction 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 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 a length of 0, 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 the remaining bytes of the longer string.

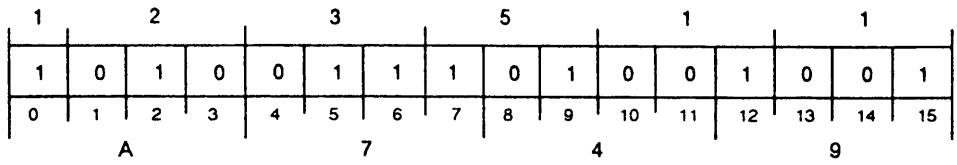
If the addresses are not valid byte-pointers within the user's address space, a protection fault may occur, even if no bytes are to be compared.

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:    LWLDA       0,LEN2      ;Get the lengths of the strings.
          LWLDA       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 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 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 $0\text{-}255_{10}$) 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; it is copied from the source string into the destination string.

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, contains positive value of number of bytes in source string.

If descending order, 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 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 AC1 containing error code 4.

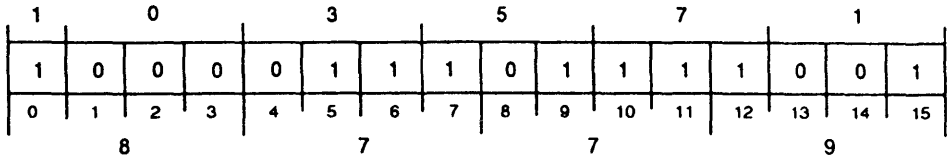
If AC2=AC3, no bytes are written, but the string is scanned for a delimiter.

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.
          NLDAI       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 → 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:
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	Set to 1 if source number of bytes is > destination number of bytes; otherwise 0.
Overflow	0
PC	PC + 1
PSR	Unchanged
Stack	Unchanged

Related Instructions

BLM	Block Move (Use this instruction, for performance improvement, when moving a relatively small number of bytes or words.)
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).

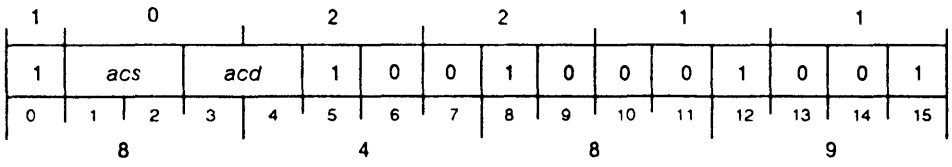
If a backward move would cause an inward ring crossing, a protection fault occurs before WCMV begins executing.

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 bits set to 1 in *acs* and adds the number to the signed 32-bit integer in *acd*.

Arguments

acs — Before execution, contains value 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

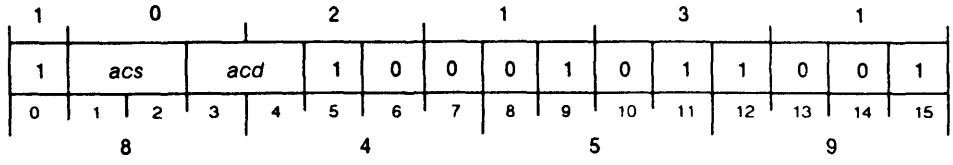
None

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: $\overline{acs} \rightarrow acd$

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 32-bit 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

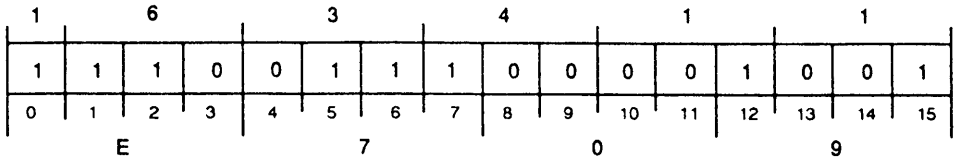
None

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 [+ = acsending, - = descending] → 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

0, the byte is not a delimiter, and the instruction processes the next byte.

1, 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 0.

After execution, contains number of bytes (or two's complement of number of bytes) not scanned plus one.

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 these instructions to place the word address of the delimiter table into AC0 and a 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.

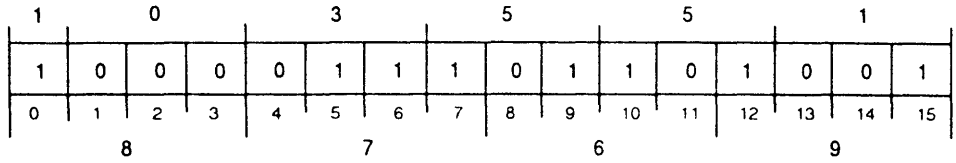
The original contents of AC0 and AC3 must be valid pointers to an area in the user's address space. If they are invalid, a protection fault may occur, even if no bytes are to be scanned, with AC1 containing error code 4.

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 → address of byte pointer to table
AC1 = # bytes [-2#] → 0
AC2 = destination byte pointer → last byte pointer + 1
AC3 = source byte pointer → last byte pointer + 1

-OR-

Function: source1 @(AC3) (translates) ?=? source2 @(AC2)
result code → AC1

Parameters: AC0 = address of translation table byte pointer → address of byte pointer to table
AC1 = # bytes [+#] → result:
-1 (source1 < source2)
0 (source1 = source2)
+1 (source1 > source2)
AC2 = source2 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*.

For either mode, **WCTR** processes one byte at a time from the source string, translating it from one data representation to another. **WCTR** performs this translation by using the source byte as an 8-bit index into a 256-byte *translation table*. The byte addressed by the index then becomes the *translated value*.

- For *translate-and-move* mode, the translated value is deposited in the next byte position in the destination string. This process continues until the specified number of bytes has been translated and moved.

- In the *translate and compare* mode, the translated value from one source string is compared with the translated value from a second source string. In performing this translation, **WCTR** treats the translated values as unsigned 8-bit integers. The process continues until either the specified number of bytes has been translated and compared, or a pair of unequal values is found. The string for which the byte has the smaller numerical value is defined as the lower-valued string. When processing is completed, **WCTR** returns a result code in AC1.

In both modes, the strings are processed from the specified starting addresses upward. The source strings for translations remain unchanged after execution. 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 memory doubleword containing byte pointer to first byte in 256-byte translation table.

After execution, contains address of doubleword containing byte pointer to translation table.

AC1 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.

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 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	source1 byte < source2 byte
0	source1 byte = source2 byte
+1	source1 byte > source2 byte

AC2 Before execution, contains 32-bit byte pointer to first byte in destination string (or source2). With each byte accessed, address incremented by 1.

After execution, contains either byte pointer to byte following destination (or source2), or, if inequality found in *translate-and-compare* mode, byte pointer to failing byte in source2.

AC3 Before execution, contains 32-bit byte pointer to first byte in source string (or source1). With each byte accessed, address incremented by 1.

After execution, contains either byte pointer to byte following source (or source1), or, if inequality found in *translate-and-compare* mode, byte pointer to failing byte in source1.

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 0, the *translate-and-compare* mode returns a 0 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 AC1 containing error code 4.

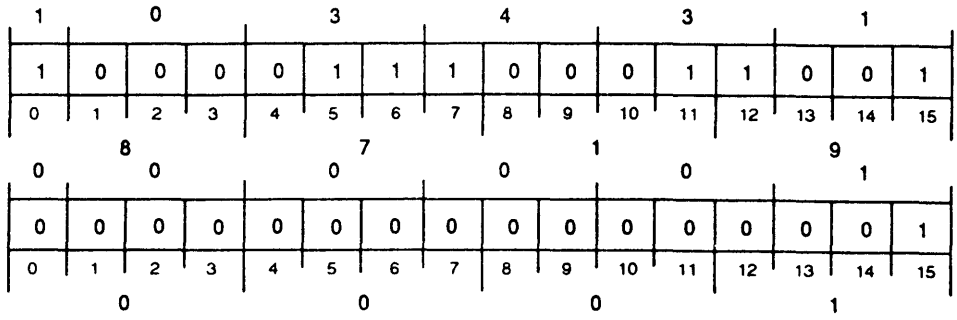
The final byte pointer (to the translation table) must specify an address in the same (or higher) segment as the original address contained in AC0. For example, if **WCTR** executes in segment 6 and AC0 contains an address in segment 7, the final byte pointer must not specify an address in segment 6.

Example

```
    .TITLE    WCTR
    .NREL
    .ENT  START, SOURCE, DEST
    ;Translate all nonalphanumeric characters in a string to blanks.
START;    LLEF        0,TBLPTR     ;Address of translate table.
          NLDAI       -100.,1      ;Two's complement of length of
                                   ;source string - translate and move.
          LLEFB       2,2*DEST     ;Point to source and destination.
          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 string 1.
    ;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: String1 @(AC2)[dec#] ?=? string2 @(AC3)[dec#]

Parameters:
AC0 = string1 data type indicator → unchanged
AC1 = string2 data type indicator → result
-1 (string1 < string2)
0 (string1 = string2)
+1 (string1 > string2)
AC2 = string1 byte pointer → unchanged
AC3 = string2 byte pointer → unchanged
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 1. **WDCMP** does not use the scale factor in the data type indicator.

After execution, contents unchanged.

AC1 — Contains data-type indicator describing decimal string 2. **WDCMP** does not use the scale factor in the data type indicator.

After execution, AC1 holds return code as follows:

-1	(string1 < string2)
0	(string1 = string2)
+1	(string1 > string2)

AC2 — Contains byte pointer to high-order byte of decimal string 1 in memory.

After execution, contents unchanged.

AC3 — Contains byte pointer to high-order byte of decimal string 2 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

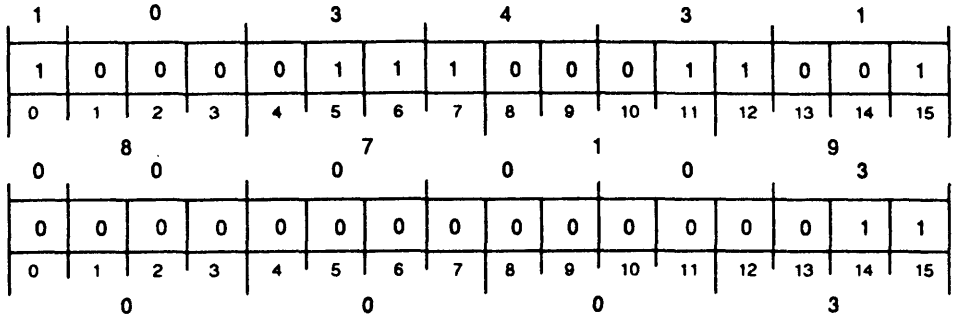
Any digit position not actually present in either argument but needing to be considered in the comparison is treated as zero.

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 → unchanged
AC3 = byte pointer to dec# → unchanged

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 1 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 unchanged.
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 low-order result is stored, and the high-order result is ignored.

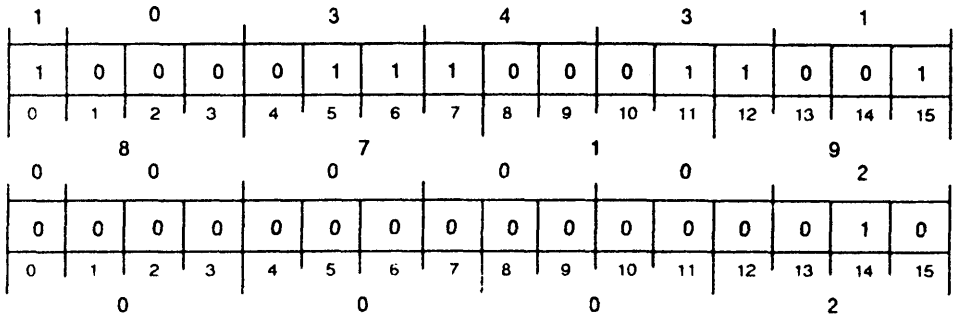
If the result is negative and the data-type is 4 (unsigned), any negative sign of the result is ignored, and the absolute value of the result is stored.

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 integer.
```

Wide Decimal Increment WDINC

WDINC



Function: @(AC3)[dec#] + 1 → @(AC3)[dec#]

Parameters: AC1 = data type indicator → unchanged
AC3 = byte pointer to dec# → unchanged

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 1 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 unchanged.
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

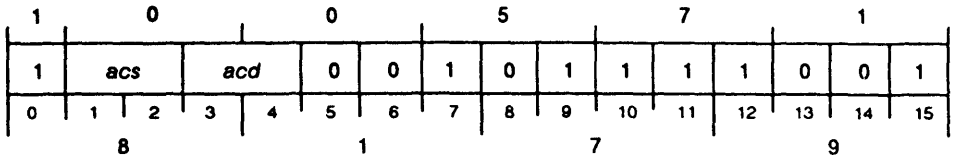
If the increment overflows the decimal string, Carry is set to 1, the low-order result is stored, and the high-order result is ignored.

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 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* = unchanged.

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*, placing the result in *acd*.

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, DIVS, DIVX, NDIV, WDIVS
Divide the contents of one or more accumulators by an accumulator.

Exceptions

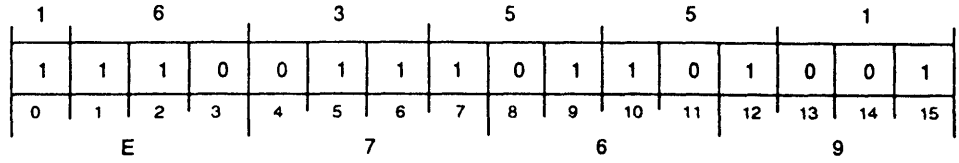
If quotient is outside the range, -2,147,483,648 to +2,147,483,647, or if *acs* contains 0, an *overflow* occurs, PSR(OVR) is set to 1, and *acd* is unchanged.

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. The instruction places the quotient in AC1 and the remainder in AC0. 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 dividend. 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, DIVS, DIVX, NDIV, WDIV
Divide the contents of one or more accumulators by an accumulator.

Exceptions

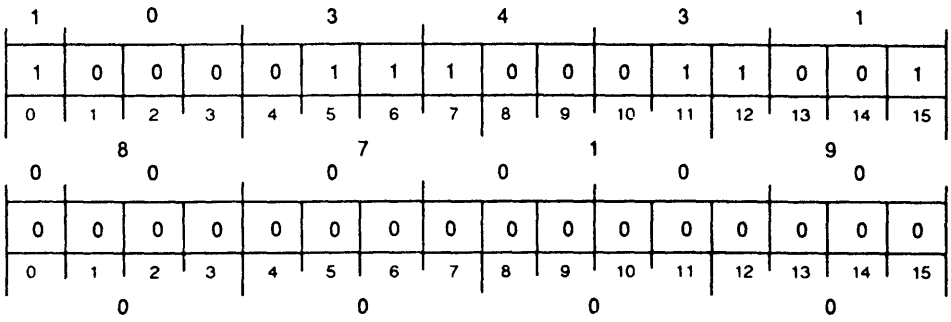
If quotient is outside the range, -2,147,483,648 to +2,147,483,647, or if AC2 contains 0, an overflow occurs, PSR(OVR) is set to 1, and AC0 and AC1 are unchanged.

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 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 the destination string 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.

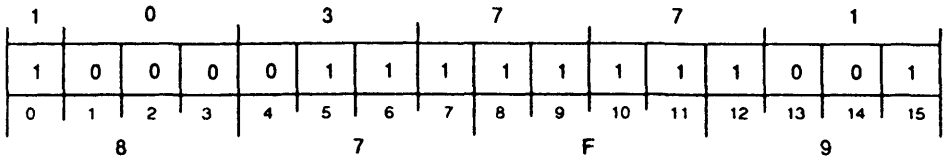
Example

```
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 and 33_8 in page zero of segment 0) to restore the processor 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

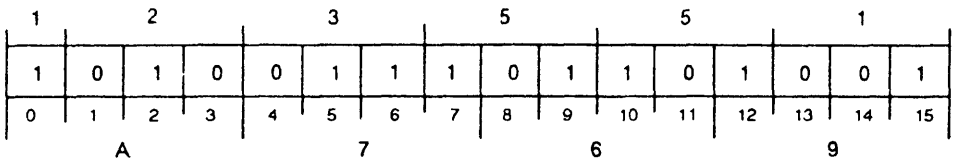
None

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, that can be tested and modified by the subprogram. The flags are the significance Trigger (T), and the Sign flag (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 instructions, however, process characters one at a time, so certain types of overlap may produce unusual side effects.

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 Wide Edit subopcode descriptions, 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 *wide stack pointer + 2 + 2*j*.

The Wide Edit operations which process numeric data (**DMVF**, **DMVN**, **DMVO**, and **DMVS**) use the following algorithm to access each source digit:

- 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.
- 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.
- 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.
- 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 Wide Edit 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 the section, "Decimal and Byte Operations," of the chapter, "Fixed-Point Computing." 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 doublewords available for use by **WEDIT**, plus six additional doublewords for interrupt use.
T	Initially, set to 0.
	Set to 1 when first nonzero 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

WEDIT considers the subprogram as data and does not check for execute protection.

If **WEDIT** is interrupted, restart information is placed on the wide stack and PSR(IRES) is set to 1.

If the sign of the source integer is invalid, a decimal/ASCII fault occurs, and **WEDIT** terminates.

If the data type indicator in AC1 specifies that the source integer is data type 6 or 7, a decimal/ASCII fault occurs, and the instruction terminates.

See also the exceptions for the individual subopcodes.

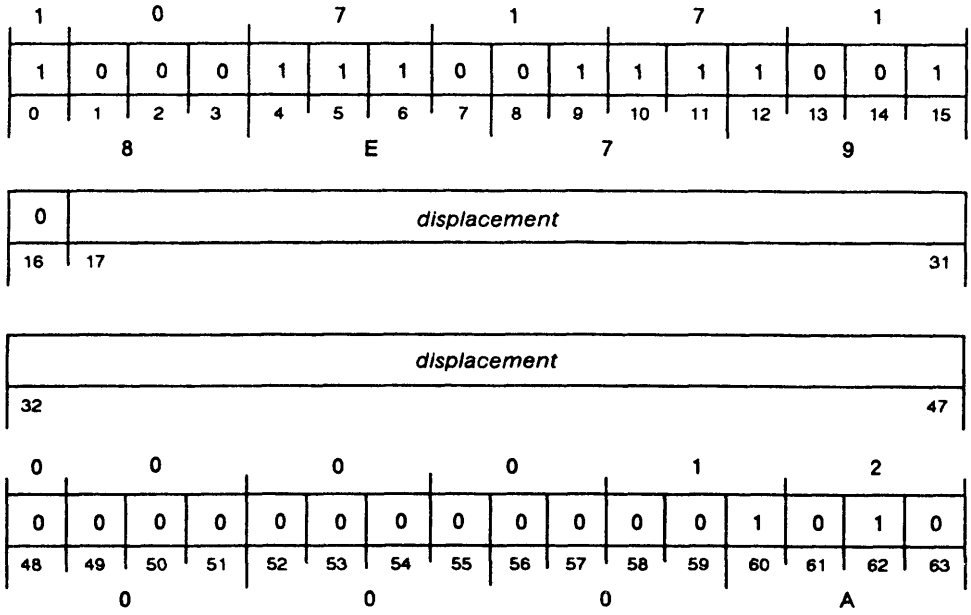
Example

```
    .TITLEW
    .ENT START,SRCSTRNG,DESTRING,EDITSUB
    ;User symbols defined in this source module may be referred to 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 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), FPAC0-3 = undefined.

WFACOSD computes the arccosine of the double-precision floating-point value in FPAC0 and places the result (in radians) in FPAC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime 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 1, the processor sets FPSR(INV) to 1, returns error code 3 to FPSR(INP), and leaves the contents of all FPACs undefined.

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 runtime routine that performs the arccosine calculation. The **WPOPJ** instruction exits from this software emulator.

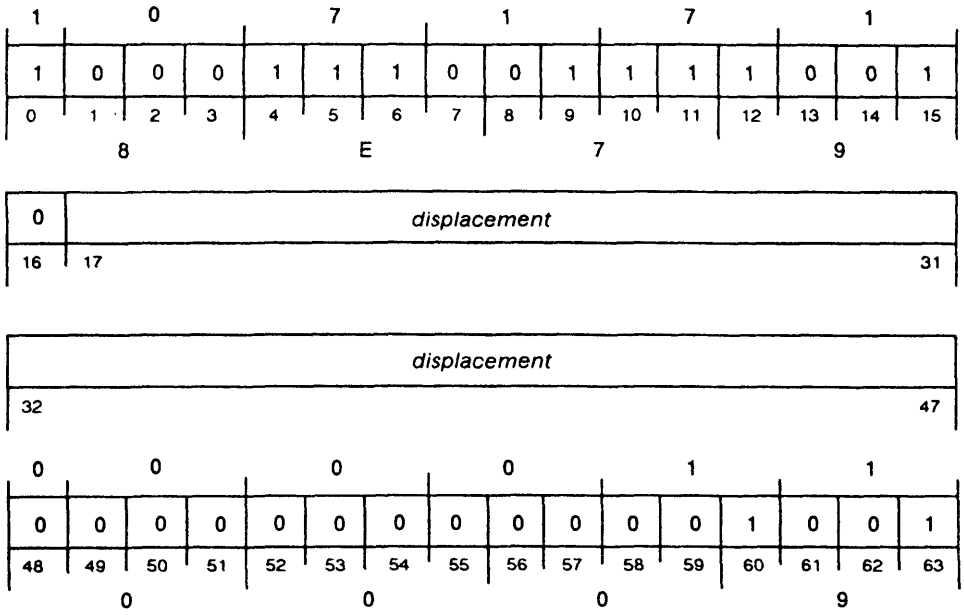
Example

```
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), FPAC0-3 = undefined.

WFACOSS computes the arccosine of the single-precision floating-point value in FPAC0, and places the result (in radians) in FPAC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime 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 (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 1, the processor sets FPSR(INV) to 1, returns error code 3 to FPSR(INP), and leaves the contents of all FPACs undefined.

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 runtime routine that performs the arccosine calculation. The **WPOPJ** instruction exits from this software emulator.

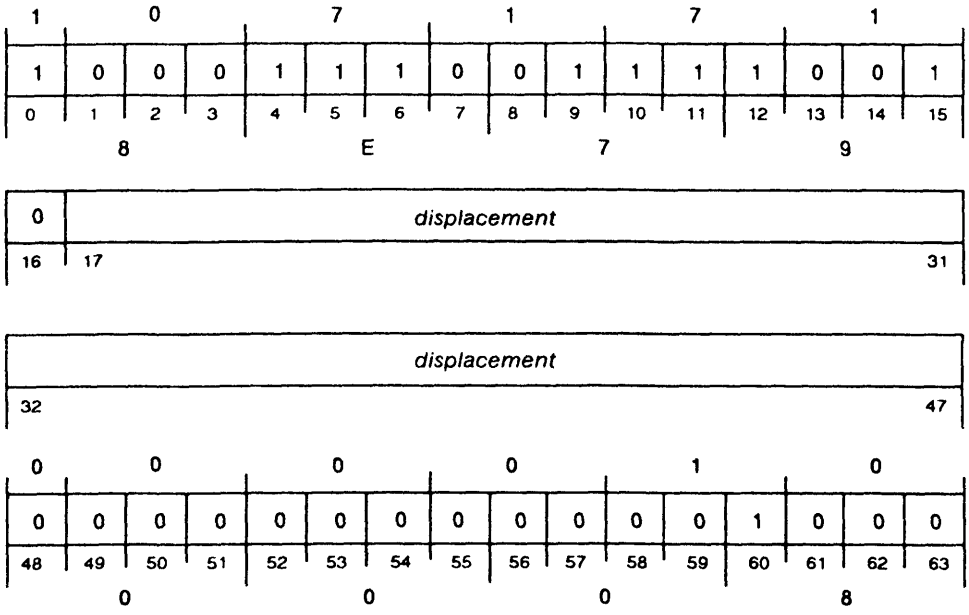
Example

```
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), FPAC0-3 = undefined.

WFASIND computes the arcsine of the double-precision floating-point value in FPAC0 and places the result (in radians) in FPAC0.

Arguments

displacement — Nonindirectable PC-relative offset to runtime 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 1, the processor sets FPSR(INV) to 1, returns error code 3 to FPSR(INP), and leaves the contents of all FPACs undefined.

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 runtime routine that performs the arcsine calculation. The **WPOPJ** instruction exits from this software emulator.

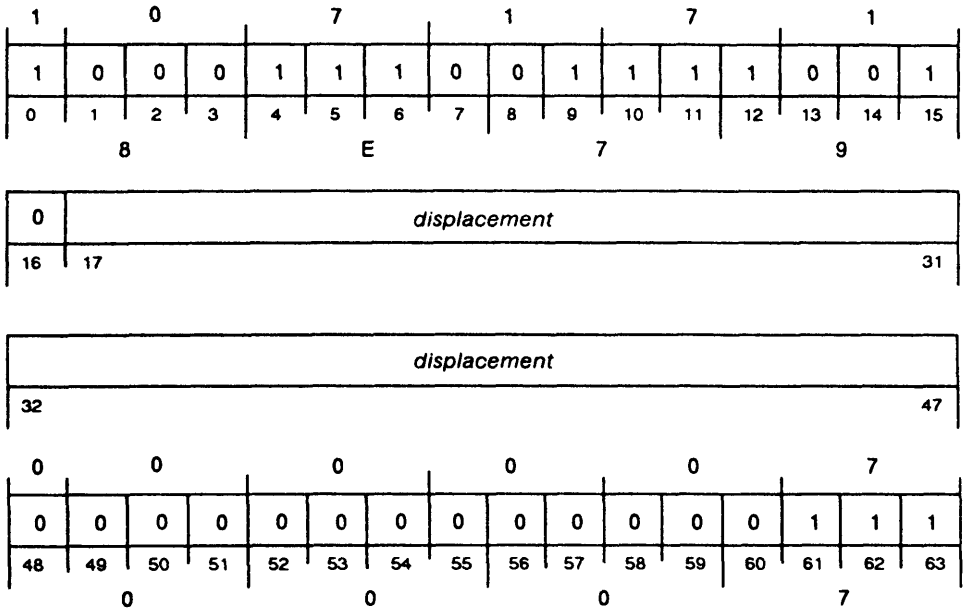
Example

```
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), FPAC0-3 = undefined.

WFASINS computes the arcsine of the single-precision floating-point value in FPAC0 and places the result (in radians) in FPAC0.

Arguments

displacement — Nonindirectable PC-relative offset to runtime 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 (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 1, the processor sets FPSR(INV) to 1, returns error code 3 to FPSR(INP), and leaves the contents of all FPACs undefined.

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 runtime routine that performs the arcsine calculation. The **WPOPJ** instruction exits from this software emulator.

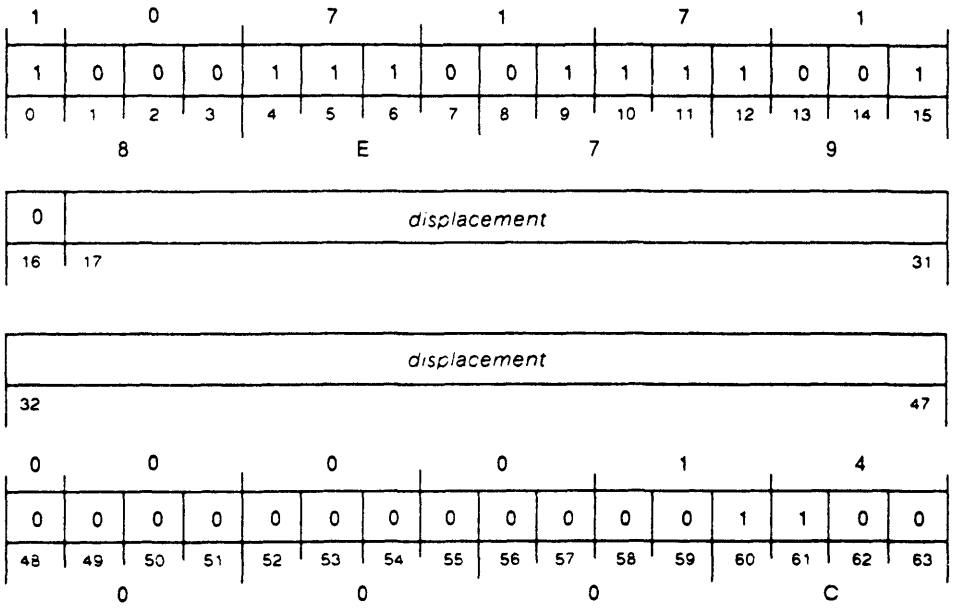
Example

```
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 double-precision floating-point value in FPAC0 and places the result (in radians) in FPAC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime 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 nonindirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a runtime routine that performs the arctangent calculation. The **WPOPJ** instruction exits from this software emulator.

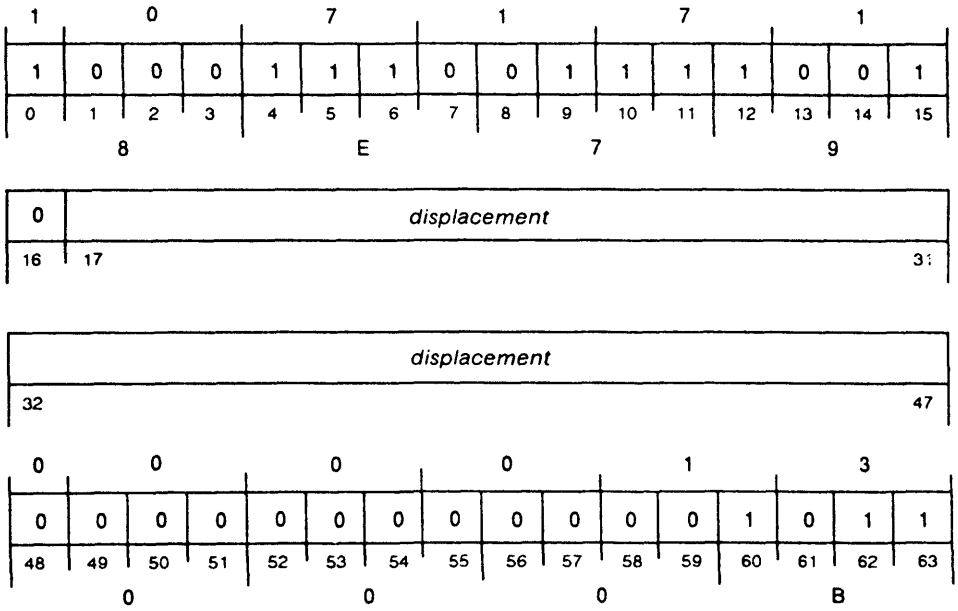
Example

```
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 single-precision floating-point value in FPAC0 and places the result (in radians) in FPAC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime 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 (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 nonindirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a runtime routine that performs the arctangent calculation. The **WPOPJ** instruction exits from this software emulator.

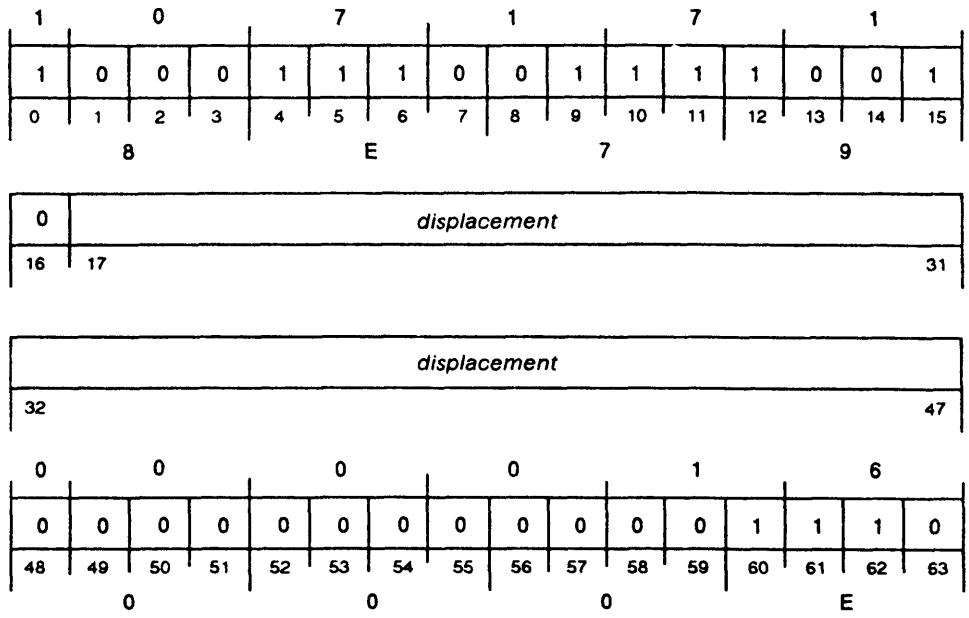
Example

```
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 FPAC0 = 0 and FPAC1 = 0, then 1 → FPSR(3), code 7 → FPSR(28-31), FPAC0-3 = undefined.

WFATN2D computes the arctangent of the quotient of two double-precision floating-point values (FPAC0 divided by FPAC1), and places the result with correct quadrature (in radians) in FPAC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime 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 values in both FPAC0 and FPAC1 are 0, the processor sets FPSR(INV) to 1, returns error code 7 to FPSR(INP), and leaves the contents of all FPACs undefined.

If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction function 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 runtime routine that performs the arctangent function. The **WPOPJ** instruction exits from this software emulator.

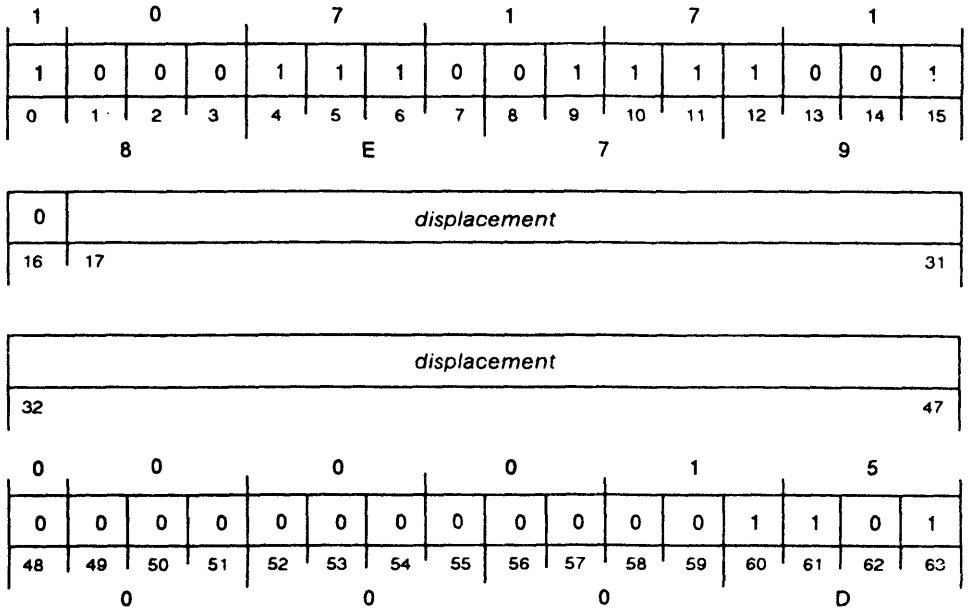
Example

```
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 FPAC0 = 0 and FPAC1 = 0, then 1 → FPSR(3), code 7 → FPSR(28-31), FPAC0-3 = undefined.

WFATN2S computes the arctangent of the quotient of two single-precision floating-point values (FPAC0 divided by FPAC1) and places the result with correct quadrature (in radians) in FPAC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime 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 (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 values in both FPAC0 and FPAC1 are 0, the processor sets FPSR(INV) to 1, returns error code 7 to FPSR(INP), and leaves the contents of all FPACs undefined.

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 runtime routine that performs the arctangent calculation. The **WPOPJ** instruction exits from this software emulator.

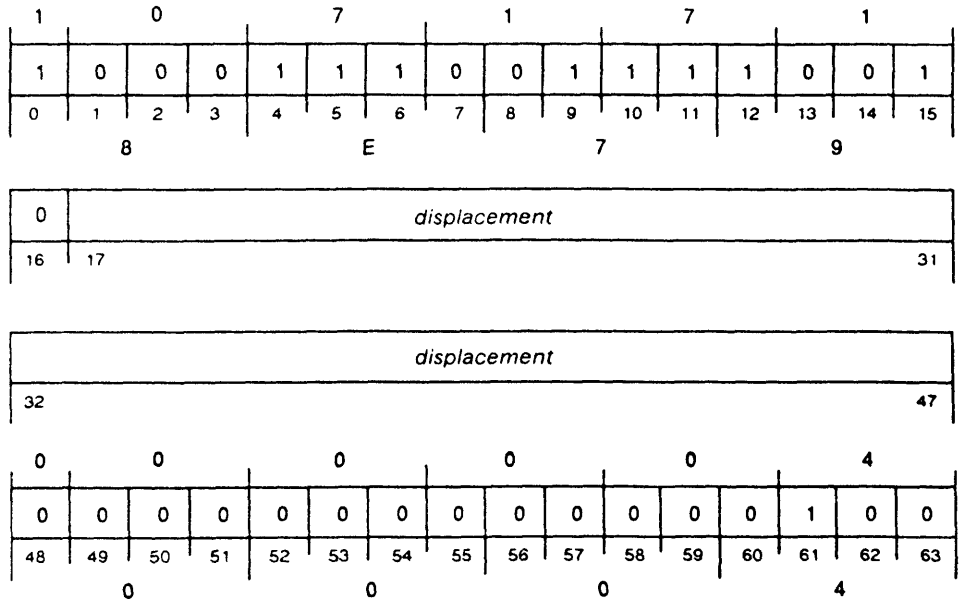
Example

```
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 double-precision floating-point value (in radians) in FPAC0, and places the result in FPAC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime 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 nonindirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a runtime routine that performs the cosine calculation. The **WPOPJ** instruction exits from this software emulator.

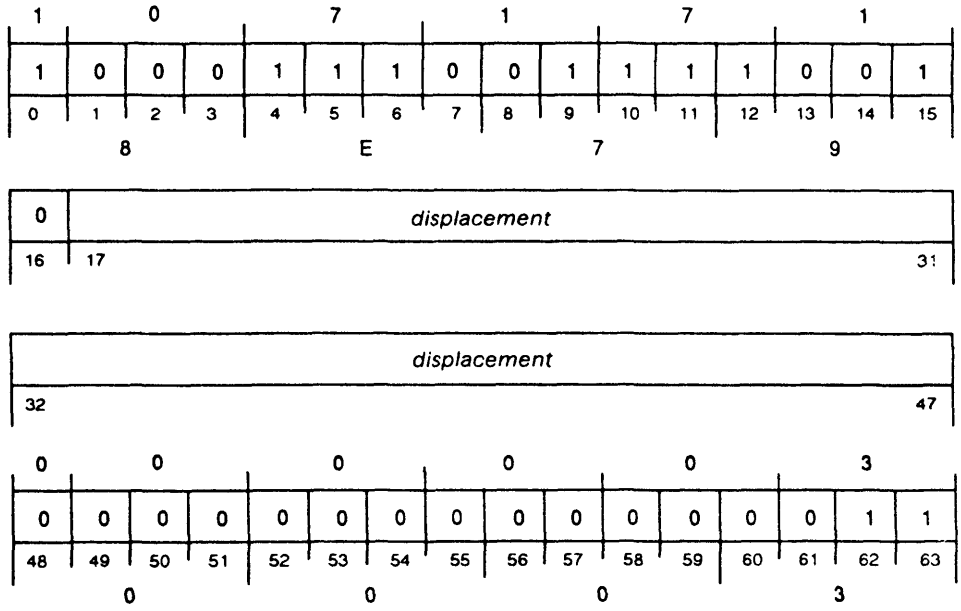
Example

```
LFLDD 0,FLOATX      ;Calculate the double-precision cosine
WFCOSDCOSD          ;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 single-precision floating-point value (in radians) in FPAC0 and places the result in FPAC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime 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 (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 nonindirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a runtime routine that performs the cosine calculation. The **WPOPJ** instruction exits from this software emulator.

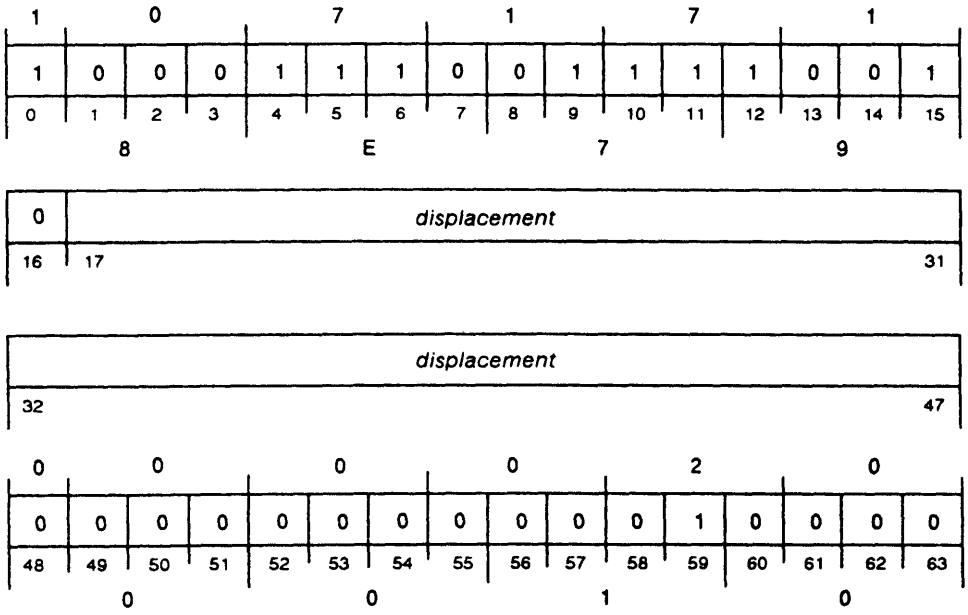
Example

```
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), FPAC0-3 = undefined.
If result will underflow, then 1 → FPSR(3), code 10_8 → FPSR(28-31), FPAC0-3 = undefined.

WFEXPD raises the value e to the double-precision floating-point power contained in FPAC0 and places the result in FPAC0.

Arguments

displacement — Nonindirectable PC-relative offset to runtime 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^{63}), the processor sets FPSR(INV) to 1, returns error code 5 to FPSR(INP), and leaves the contents of all FPACs undefined.

If the input value in FPAC0 produces a result that underflows, the processor sets FPSR(INV) to 1, returns error code 10_8 to FPSR(INP), and leaves the contents of all FPACs undefined.

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 runtime routine that performs the exponential calculation. The **WPOPJ** instruction exits from this software emulator.

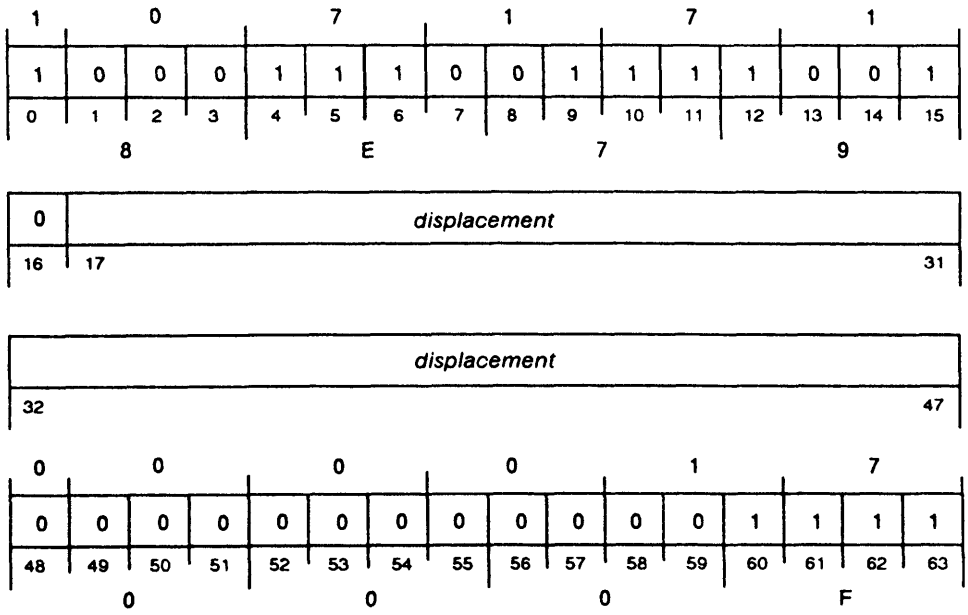
Example

```
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), FPAC0-3 = undefined.
If result will underflow, then 1 → FPSR(3), code 10_8 → FPSR(28-31), FPAC0-3 = undefined.

WFEXPS raises the value e to the single-precision floating-point power contained in FPAC0 and places the result in FPAC0.

Arguments

displacement — Nonindirectable PC-relative offset to runtime 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 (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^{63}), the processor sets FPSR(INV) to 1, returns error code 5 to FPSR(INP), and leaves the contents of all FPACs undefined.

If the input value in FPAC0 produces a result that underflows, the processor sets FPSR(INV) to 1, returns error code 10_8 to FPSR(INP), and leaves the contents of all FPACs undefined.

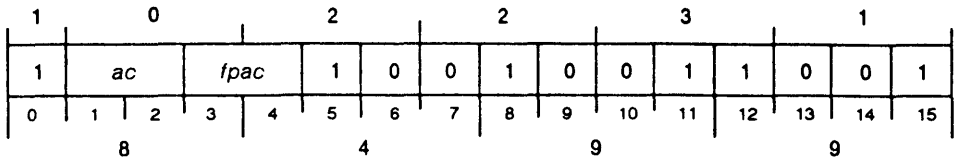
If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address, E (using the *displacement* as a nonindirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a runtime routine that performs the exponential calculation. The **WPOPJ** instruction exits from this software emulator.

Example

```
FMOV   2,0          ;Calculate the single-precision exponential
WFEXPS EXPS         ;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 *fpac* to a signed 32-bit integer. It does this by taking the absolute value of the integer portion of the number contained in *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 *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

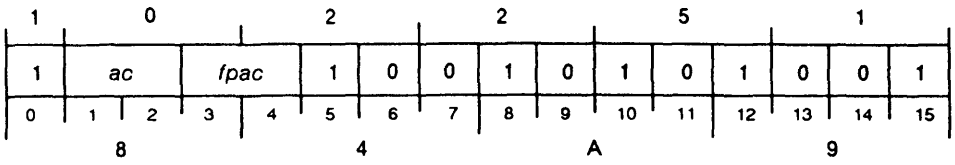
If the integer portion of the number contained in *fpac* is less than −2,147,483,648 or greater than +2,147,483,647, FPSR(MOV) is set to 1.

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 *ac* into a double-precision floating-point number and places the result into *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

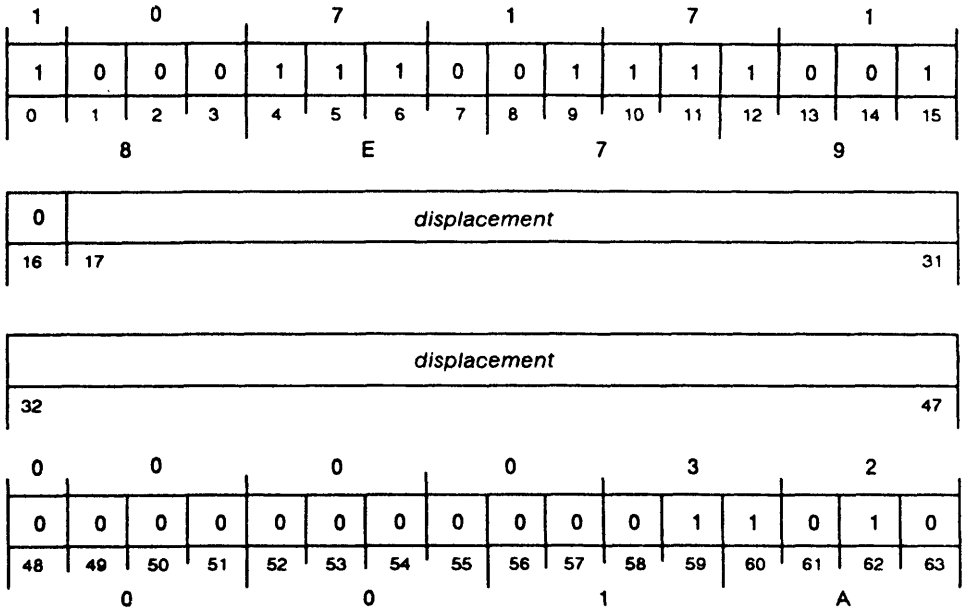
Example

```
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), FPAC0-3 = undefined.

WFLG2D computes the binary logarithm ($\log_2$) of the double-precision floating-point number contained in FPAC0 and places the result in FPAC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime 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 FPSR(INV) to 1, returns error code 1 to FPSR(INP), and leaves the contents of all FPACs undefined.

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 runtime routine that performs the binary logarithm calculation. The **WPOPJ** instruction exits from this software emulator.

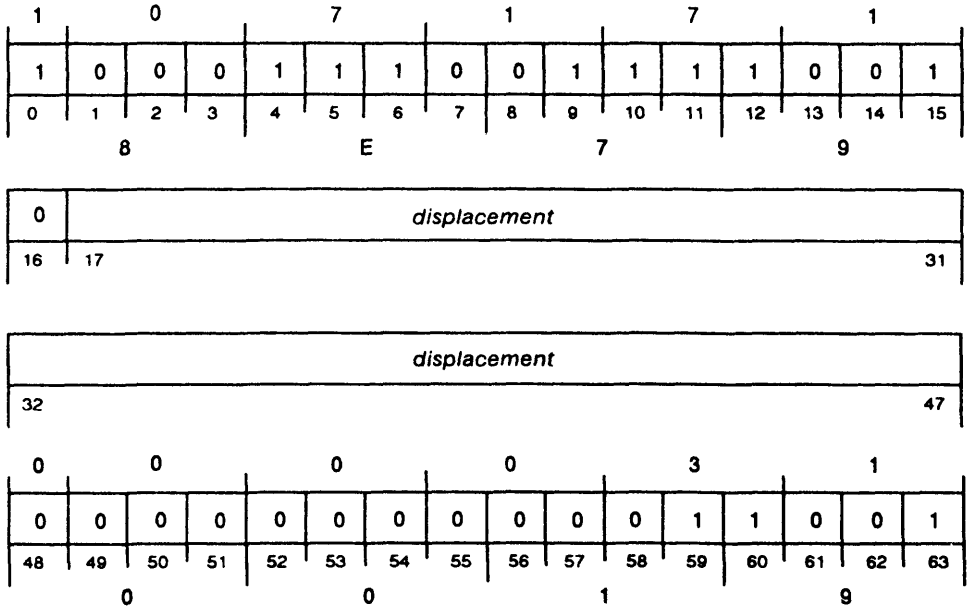
Example

```
LFLDD 0,X          ;Calculate the double-precision log2 of the
WFLG2D LOG2D       ;floating-point number at location X, and store
LFSTD 0,LOG2X      ;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), FPAC0-3 = undefined.

WFLG2S computes the binary logarithm ($\log_2$) of the single-precision floating-point number contained in FPAC0 and places the result in FPAC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime 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 (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

WFLG2D Floating-Point Binary Logarithm Double

Exceptions

If the input value in FPAC0 is less than or equal to 0, the processor sets FPSR(INV) to 1, returns error code 1 to FPSR(INP), and leaves the contents of all FPACs undefined.

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 runtime routine that performs the binary logarithm calculation. The **WPOPJ** instruction exits from this software emulator.

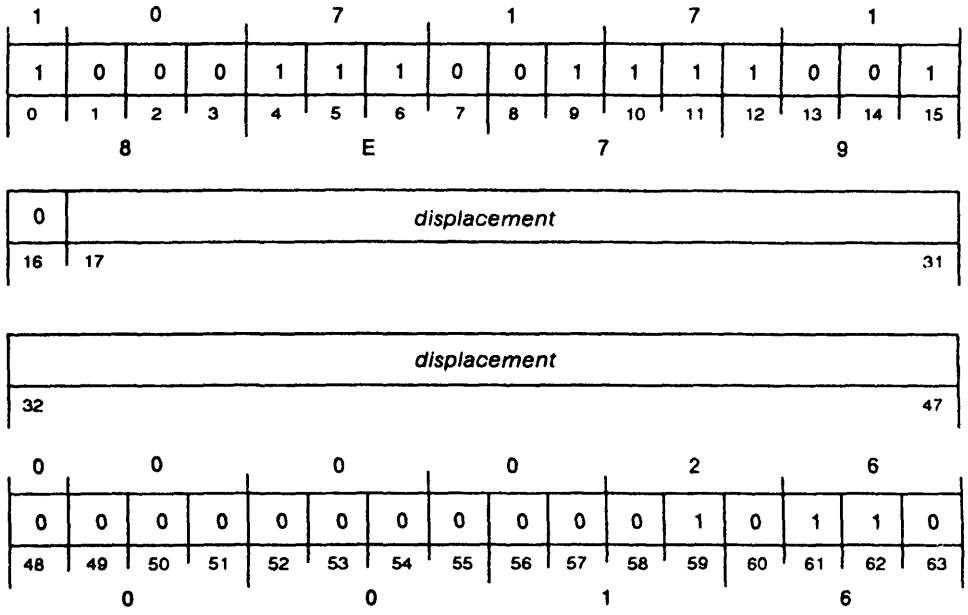
Example

```
FMOV  1,0          ;Calculate the single-precision log2
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: loge 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), FPAC0-3 = undefined.

WFLNGD computes the natural logarithm ($\log_e$) of the double-precision floating-point number contained in FPAC0 and places the result in FPAC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime 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. After execution, contents undefined.
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 FPSR(INV) to 1, returns error code 1 to FPSR(INP), and leaves the contents of all FPACs undefined.

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 runtime routine that performs the natural logarithm calculation. The **WPOPJ** instruction exits from this software emulator.

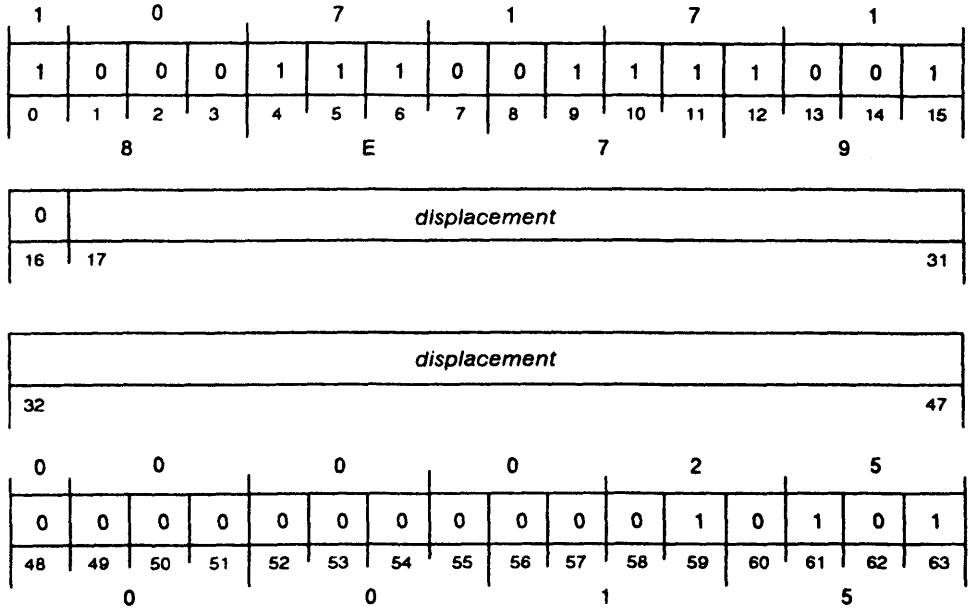
Example

```
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: loge 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), FPAC0-3 = undefined.

WFLNGS computes the natural logarithm (loge) of the single-precision floating-point number contained in FPAC0 and places the result in FPAC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime 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

FPAC0(0-31) Before execution, contains 32-bit floating-point value.
After execution, contains 32-bit result (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

WFLNGD Floating-Point Natural Logarithm Double

Exceptions

If the input value in FPAC0 is less than or equal to 0, the processor sets FPSR(INV) to 1, returns error code 1 to FPSR(INP), and leaves the contents of all FPACs undefined.

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 runtime routine that performs the natural logarithm calculation. The **WPOPJ** instruction exits from this software emulator.

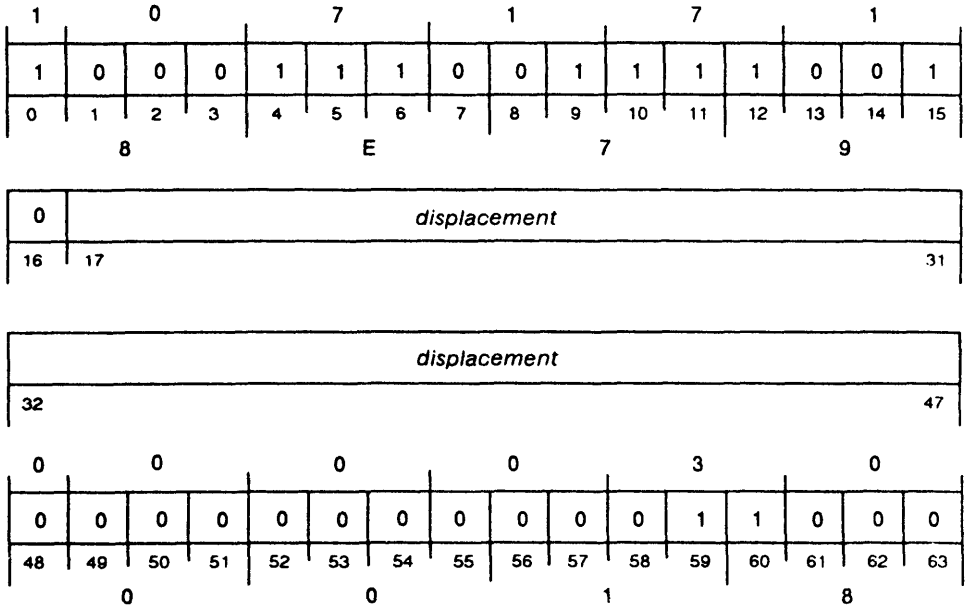
Example

```
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), FPAC0-3 = undefined.

WFLOGD computes the common logarithm ($\log_{10}$) of the double-precision floating-point number contained in FPAC0 and places the result in FPAC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime 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 FPSR(INV) to 1, returns error code 1 to FPSR(INP), and leaves the contents of all FPACs undefined.

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 runtime routine that performs the common logarithm calculation. The **WPOPJ** instruction exits from this software emulator.

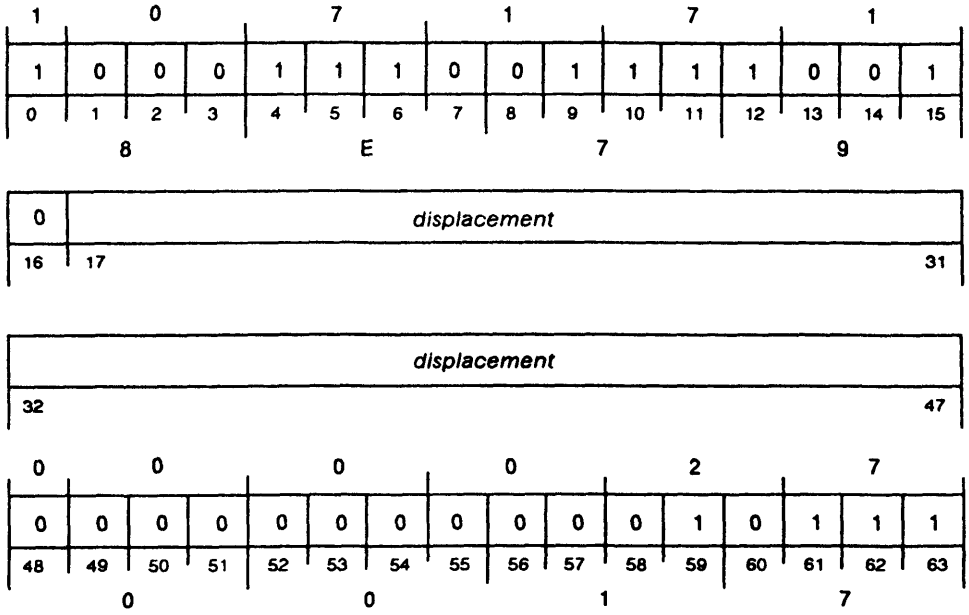
Example

```
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), FPAC0-3 = undefined.

WFLOGS computes the common logarithm ($\log_{10}$) of the single-precision floating-point number contained in FPAC0 and places the result in FPAC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime 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 (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 FPSR(INV) to 1, returns error code 1 to FPSR(INP), and leaves the contents of all FPACs undefined.

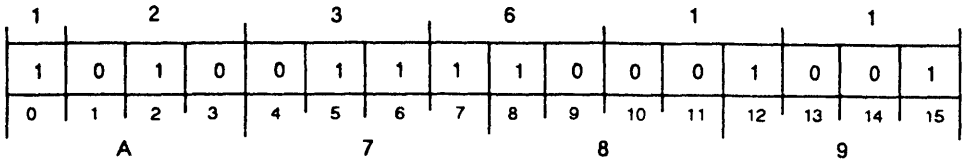
If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address, E (using the *displacement* as a nonindirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a runtime routine that performs the common logarithm calculation. The **WPOPJ** instruction exits from 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

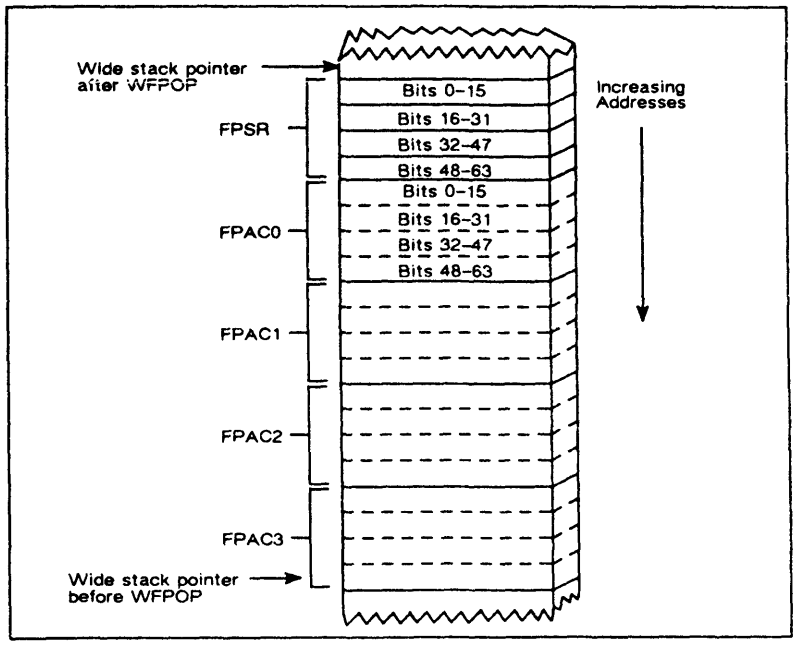


Function: 10 stack doublewords → registers
1st two doublewords → FPAC3
2nd two doublewords → FPAC2
3rd two doublewords → FPAC1
4th two doublewords → FPAC0
last two doublewords → FPSR

Parameters: None

NOTE: Certain FPSR bits are not set by this instruction.

WFPOP pops the state of the floating-point unit, a 10-doubleword block, off the wide stack and loads the contents into the four floating-point accumulators and the floating-point status register (FPSR). Unnormalized data is moved without change. The format of the 10-doubleword block is as follows:



The first eight doublewords popped are loaded into the four floating-point accumulators; the last two doublewords popped, a 64-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(1-8) receive memory(1-8).

FPSR(9-11) must be loaded as zeros.

FPSR(12-15) are not set from memory. These bits contain floating-point identification code, are set by the processor, and cannot be changed.

FPSR(16-21) are not loaded from memory. The processor sets these bits to 0.

FPSR(22) is loaded from memory only if the system supports a parallel floating-point unit; otherwise the processor sets this bit to 0.

FPSR(23-27) are set by the processor to 0.

FPSR(28-31) are set according to the state of ANY:

If ANY = 0, FPSR(28-31) are undefined.

If ANY = 1, loaded from memory(28-31).

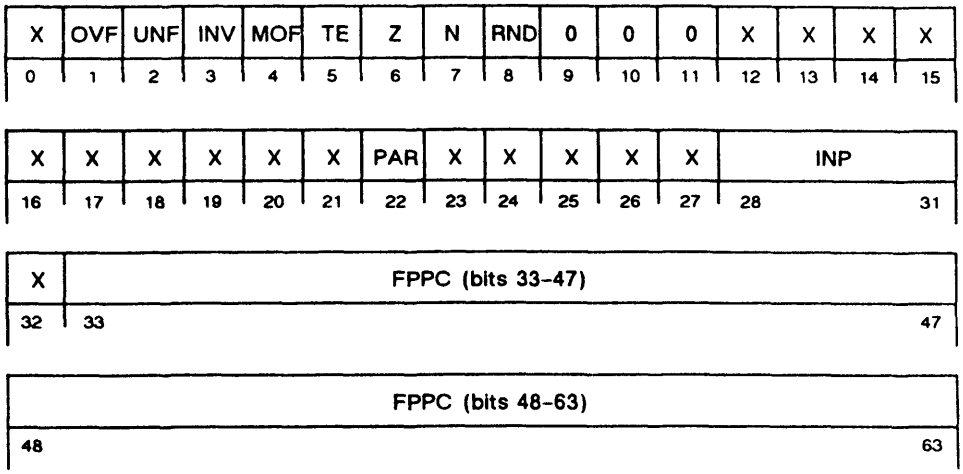
FPSR(32) set to 0.

FPSR(33-63) set according to the state of ANY:

If ANY = 0, FPSR(33-63) are undefined.

If ANY = 1, FPSR(33-63) loaded from memory(33-63).

The contents of the stack doublewords (as they correspond to the bits in the FPSR) are



X = Processor ignores this bit

Arguments

None

Registers, Flags, and Stacks

FPAC0-FPAC3	After execution, contain data from stack.
PC	PC + 1
FPSR	After execution, contains data from stack.
Stack	Wide stack pointer decremented by 10 doublewords.

Related Instructions

FPOP	Floating-Point Pop
FPSH, WFPSH	Push floating-point state onto stack.

Exceptions

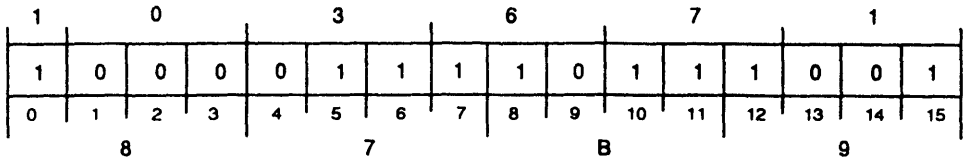
WFPOP initiates a floating-point trap if FPSR(ANY) and FPSR(TE) are both 1 after FPSR(FPPC) is loaded.

Example

```
WFPSH              ;Save the floating-point state before calling this
LCALL  ROUTINE     ;routine, since it may destroy some of the FPACs.
WFPOP              ;Restore the floating-point state after returning.
```

Wide Floating-Point Push WFPSH

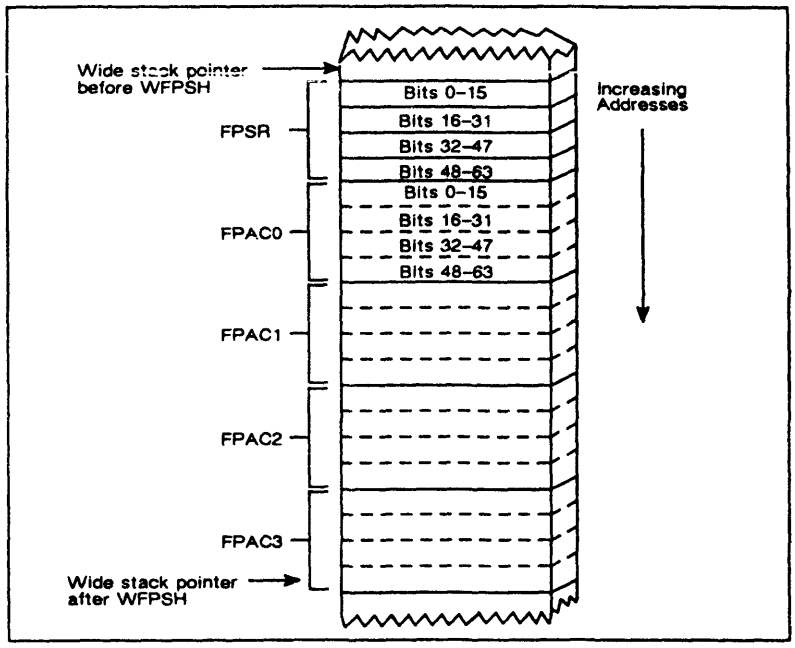
WFPSH



Function: registers → 10 stack doublewords
FPSR → 1st two doublewords
FPAC0 → 2nd two doublewords
FPAC1 → 3rd two doublewords
FPAC2 → 4th two doublewords
FPAC3 → 5th two doublewords

Parameters: None

WFPSH pushes the state of the floating-point unit, a 10-doubleword block, onto the wide stack, leaving the contents of the floating-point accumulators and the floating-point status register unchanged. Unnormalized data is moved without change. The format of the 10 doublewords pushed is as follows:



The first two doublewords pushed onto the stack, a 64-bit operand, are taken from the floating-point status register as follows:

Memory(0-31) contain FPSR(0-31).

Memory(32) is set by the processor to 0.

Memory(33-63) contents dependent on the state of FPSR(ANY):

If ANY is 0, memory(33-63) are undefined.

If ANY is 1, memory(33-63) contain FPSR(33-63).

The next eight doublewords pushed on the stack are taken from the four floating-point accumulators.

Arguments

None

Registers, Flags, and Stacks

FPAC0–FPAC3	Provide data to the stack. After execution, contents unchanged.
PC	PC + 1
FPSR	Provides data to the stack. After execution, contents unchanged.
Stack	Wide stack pointer incremented by 10 doublewords.

Related Instructions

FPSH Floating-Point Push

FPOP, WFPOP Load floating-point accumulators and floating-point status register with contents of stack.

Exceptions

WFPSH will not store an FPSR value with any combination of bit 5 (TE) and bits 1–4 concurrently set.

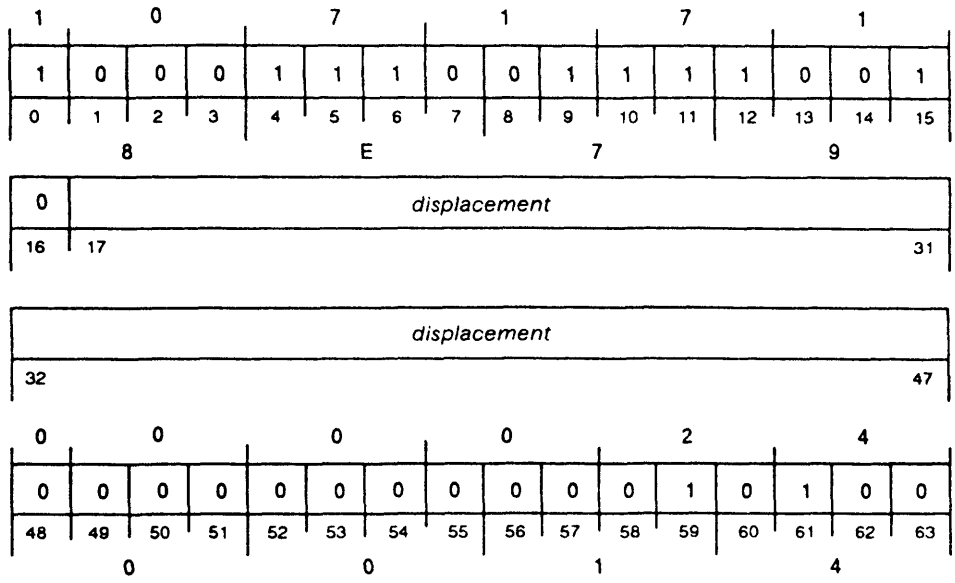
Example

```
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), FPAC0-3 = undefined.
If result will overflow, then 1 → FPSR(3), code 5 → FPSR(28-31), FPAC0-3 = undefined.
If result will underflow, then 1 → FPSR(3), code 10_8 → FPSR(28-31), FPAC0-3 = undefined.

WFPWRD raises the double-precision floating-point number contained in FPAC0 to the double-precision floating-point power contained in FPAC1 and places the result in FPAC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime 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 FPSR(INV) to 1, returns error code 4 to FPSR(INP), and leaves the contents of all FPACs undefined.

If the values in either FPAC0 or FPAC1 will produce a result that overflows, the processor sets FPSR(INV) to 1, returns error code 5 to FPSR(INP), and leaves the contents of all FPACs undefined.

If the values in either FPAC0 or FPAC1 will produce a result that underflows, the processor sets FPSR(INV) to 1, returns error code 10_8 to FPSR(INP), and leaves the contents of all FPACs undefined.

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 runtime routine that performs the power calculation. The **WPOPJ** instruction exits from this software emulator.

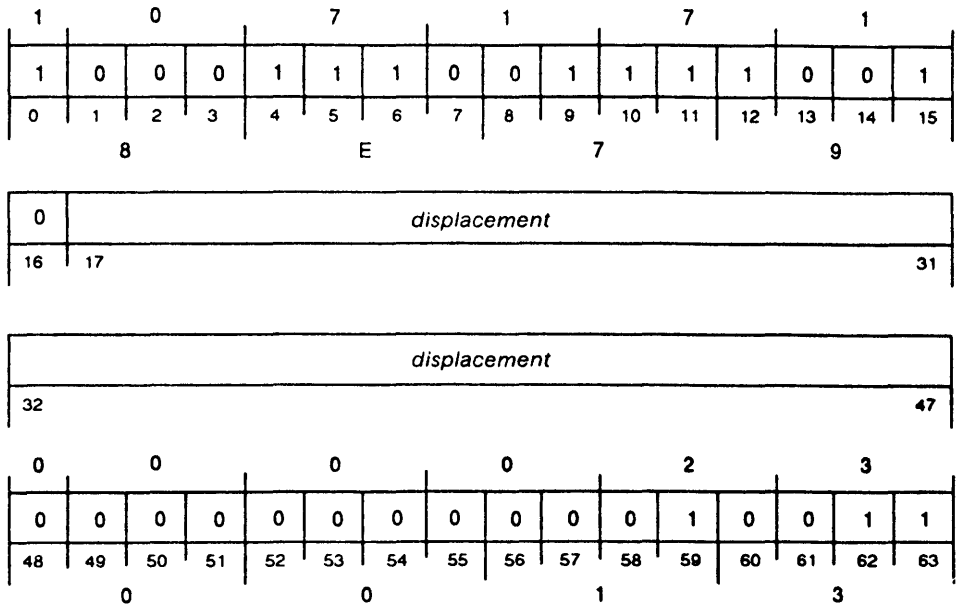
Example

```
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), FPAC0-3 = undefined.
If result will overflow, then 1 → FPSR(3), code 5 → FPSR(28-31), FPAC0-3 = undefined.
If result will underflow, then 1 → FPSR(3), code 10_8 → FPSR(28-31), FPAC0-3 = undefined.

WFPWRS raises the single-precision floating-point number contained in FPAC0 to the single-precision floating-point power contained in FPAC1 and places the single-precision result in FPAC0.

Arguments

displacement — Nonindirectable PC-relative offset to runtime 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 (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 FPSR(INV) to 1, returns error code 4 to FPSR(INP), and leaves the contents of all FPACs undefined.

If the values in either FPAC0 or FPAC1 will produce a result that overflows, the processor sets FPSR(INV) to 1, returns error code 5 to FPSR(INP), and leaves the contents of all FPACs undefined.

If the values in either FPAC0 or FPAC1 will produce a result that underflows, the processor sets FPSR(INV) to 1, returns error code 10_8 to FPSR(INP), and leaves the contents of all FPACs undefined.

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 runtime routine that performs the power calculation. The **WPOPJ** instruction exits from this software emulator.

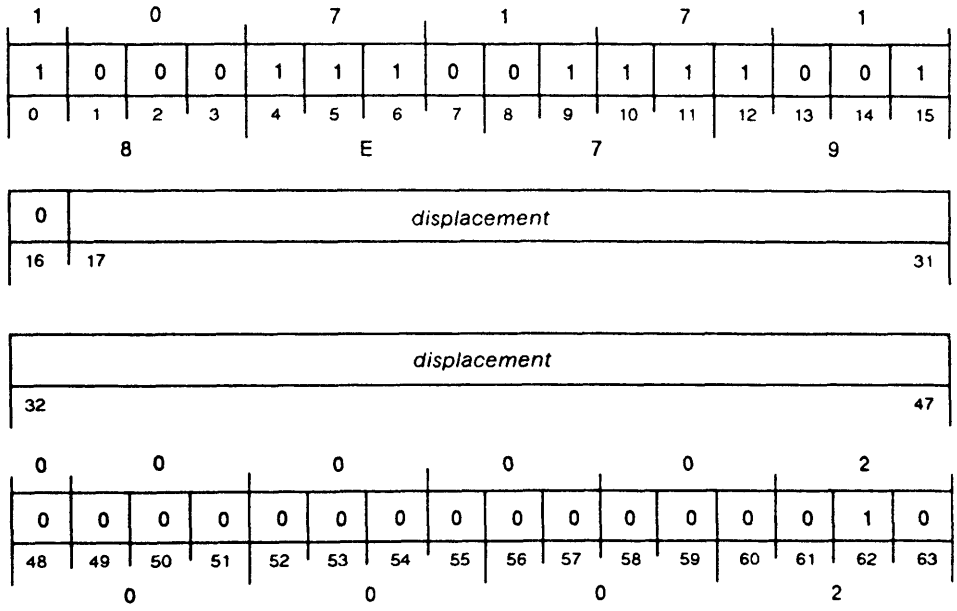
Example

```
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 double-precision floating-point value (in radians) in FPAC0 and places the result in FPAC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime 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 nonindirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a runtime routine that performs the sine calculation. The **WPOPJ** instruction exits from this software emulator.

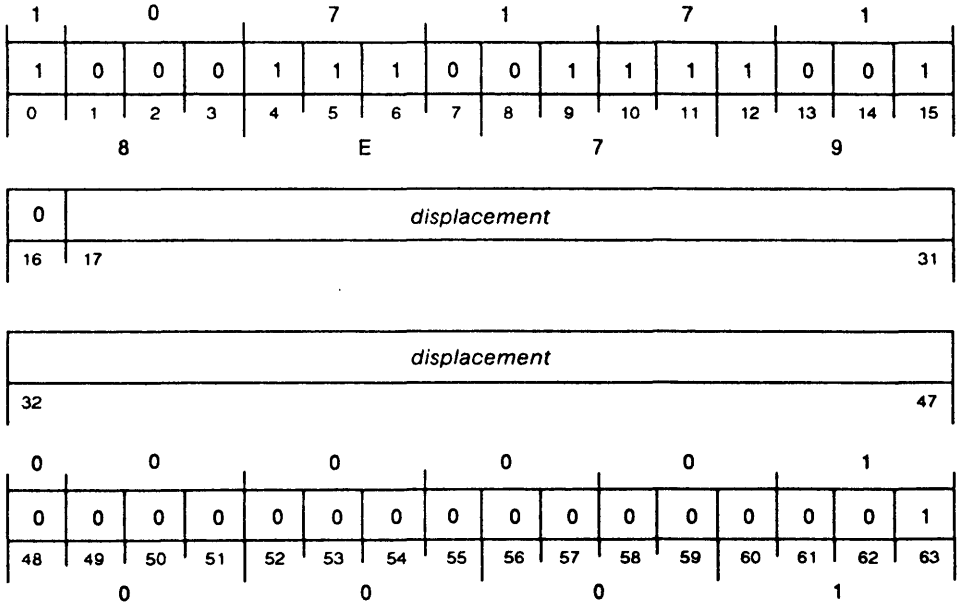
Example

```
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 single-precision floating-point value (in radians) in FPAC0, and places the result in FPAC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime 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 (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 nonindirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a runtime routine that performs the sine calculation. The **WPOPJ** instruction exits from this software emulator.

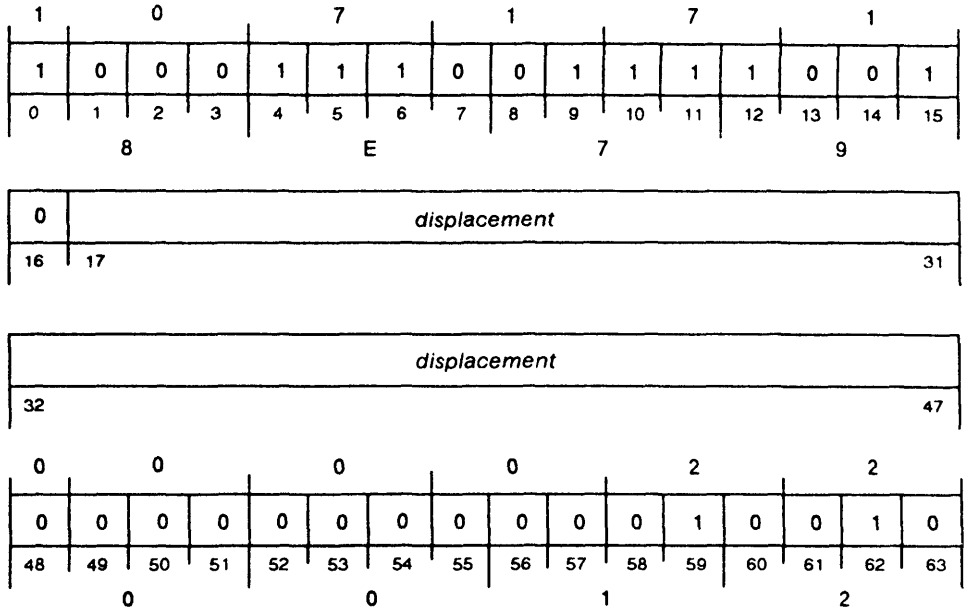
Example

```
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: $\overline{\text{FPAC0}} \rightarrow$ 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), FPAC0-3 = undefined.

WFSQRD computes the square root of the double-precision floating-point number contained in FPAC0 and places the result in FPAC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime 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 FPSR(INV) to 1, returns error code 2 to FPSR(INP), and leaves the contents of all FPACs undefined.

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 runtime routine that performs the square root calculation. The **WPOPJ** instruction exits from this software emulator.

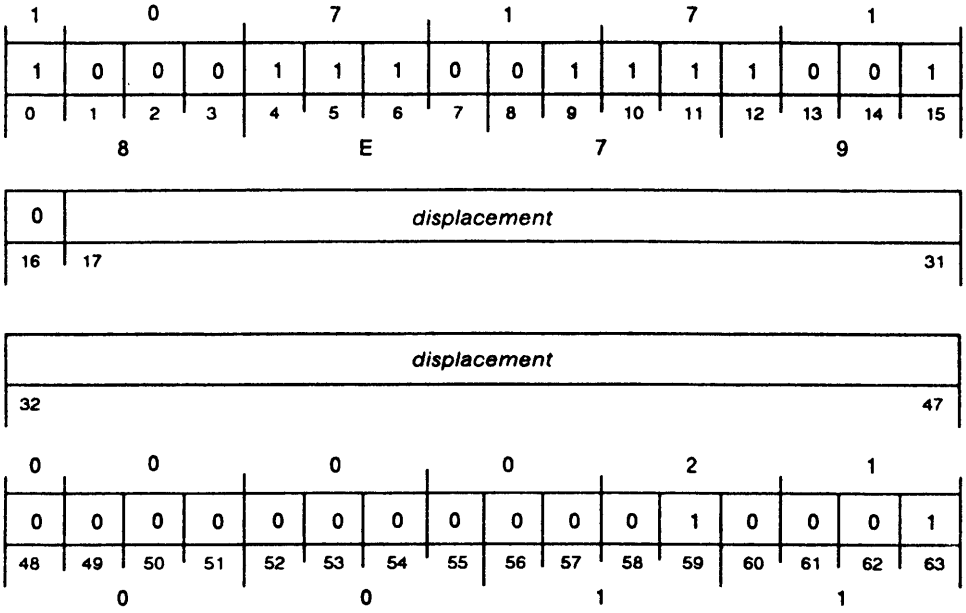
Example

```
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: $\overline{\text{FPAC0}} \rightarrow$ 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), FPAC0-3 = undefined.

WFSQRS computes the square root of the single-precision floating-point number contained in FPAC0 and places the result in FPAC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime 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 (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 0, the processor sets FPSR(INV) to 1, returns error code 2 to FPSR(INP), and leaves the contents of all FPACs undefined.

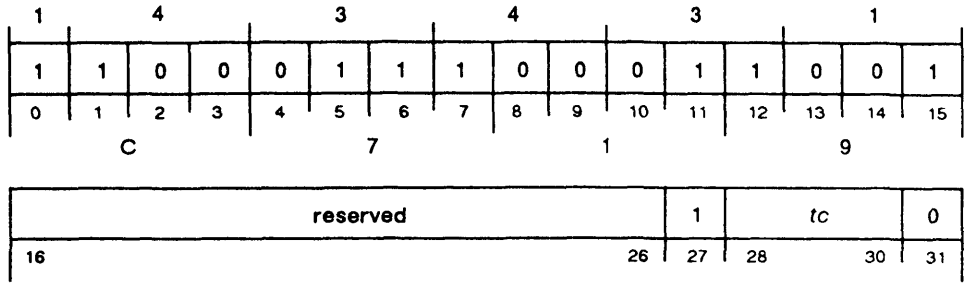
If hardware support is currently unavailable, the processor performs an **LPSHJ** instruction and computes the effective address, E (using the *displacement* as a nonindirectable PC-relative offset). The effective address must be in the current ring. The processor uses E as the address of a runtime routine that performs the square root calculation. The **WPOPJ** instruction exits from 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 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) → 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

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 (in 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 (on the wide stack). 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 of instruction 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 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.

After execution,

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 (only after unsuccessful search or another interrupt), 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 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 doubleword 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

An invalid address (produced by the contents of either or both of AC1 and AC3) may cause a protection fault. The fault code returned to AC1 is dependent on the type of fault.

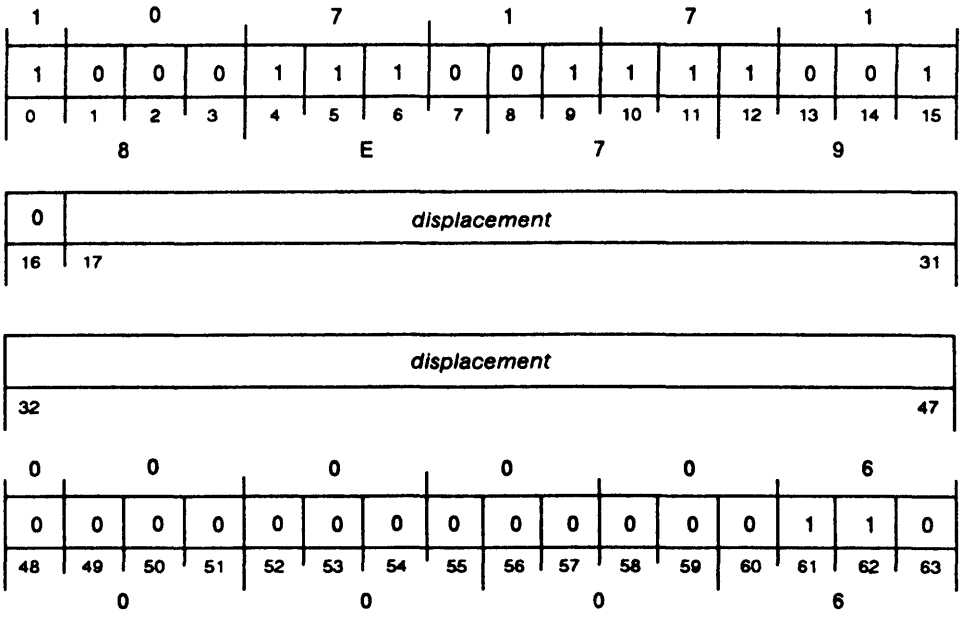
Example

```
    ;This example searches through a queue, finds all elements with a
    ;value of 6 at offset 4, and changes the value of these elements 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), FPAC0-3 = undefined.

WFTAND computes the tangent of the double-precision floating-point value (in radians) in FPAC0, and places the result in FPAC0.

Arguments

displacement Nonindirectable PC–relative offset to runtime 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 FPSR(INV) to 1, returns error code 6 to FPSR(INP), and leaves the contents of all FPACs undefined. (If the integer multiples of values is even, FPSR(INP) contains 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 runtime routine that performs the tangent calculation. The **WPOPJ** instruction exits from this software emulator.

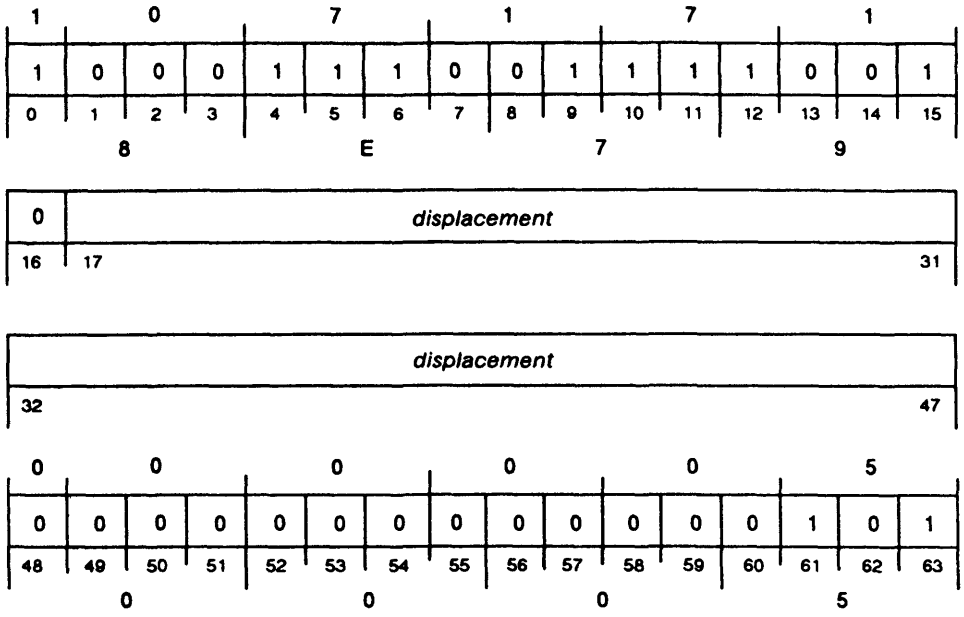
Example

```
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), FPAC0-3 = undefined.

WFTANS computes the tangent of the single-precision floating-point value (in radians) in FPAC0 and places the result in FPAC0.

Arguments

displacement Nonindirectable PC-relative offset to runtime 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 (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 FPSR(INV) to 1, returns error code 6 to FPSR(INP), and leaves the contents of all FPACs undefined. (If the integer multiples of values is even, FPSR(INP) contains 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 runtime routine that performs the tangent calculation. The **WPOPJ** instruction exits from this software emulator.

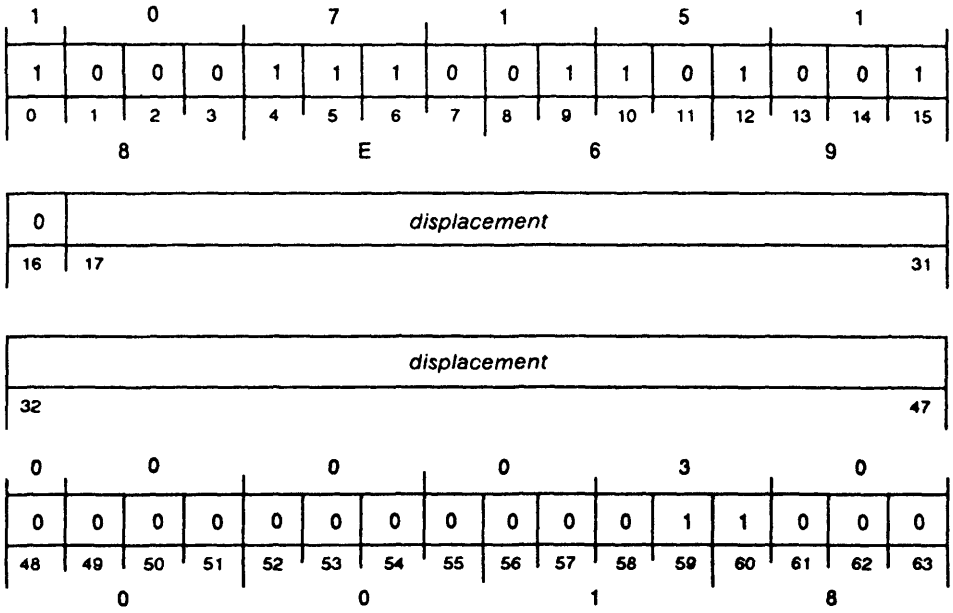
Example

```
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 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:

Doubleword #	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 0, **WGBITBLT** has no effect.

If an overdraw condition occurs, PSR(OVR) is set to 1.

WGBITBLT microcode is designed to optimize its speed if certain conditions are met. For the fastest possible execution, you should ensure that the following is true: for virtual to physical bitmap transfers, if the pixels are left justified to a doubleword boundary, **WGBITBLT** transfers an integral number of whole doublewords for each line.

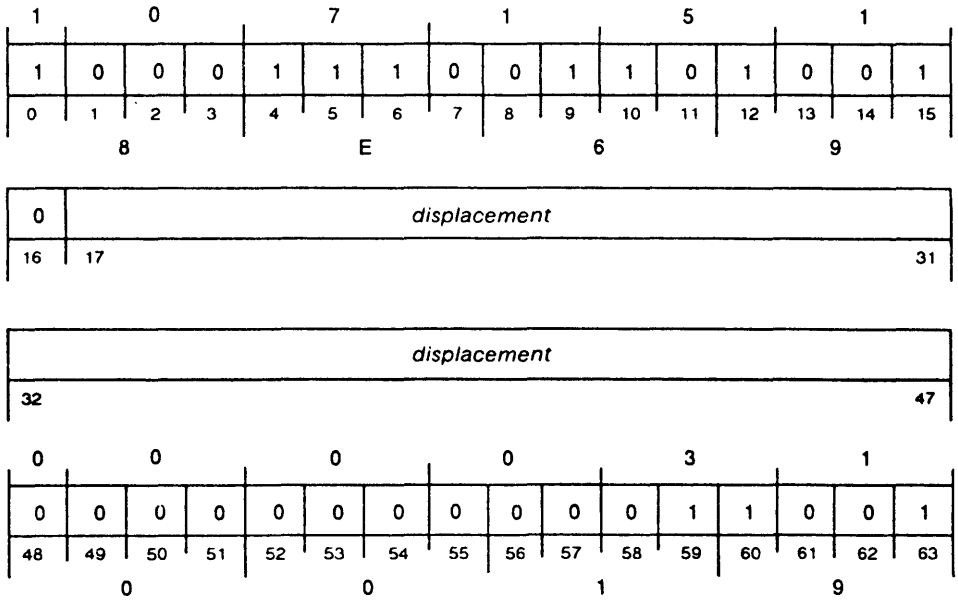
Example

```
;The following subroutine allows access to the WGBITBLT instruction
;from any common code compiler.

;$WGBITBLT (source_form, destination_form, packet)
$WGBITBLT:
      WSAVR       0
      XWLDA       0,@ARG1,3
      XWLDA       1,@ARG2,3
      XLEF        2,@ARG3,3
      WGBITBLT    GI_GIS2_TRAP
      WRTN
```

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 1 indicates foreground color; a character bit set to 0 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:

Doubleword #	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 0, **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.

Unless the source form (character) is one-bit per pixel and on a virtual bitmap, an invalid **WGCHRBLT** source fault occurs.

If the specified character rectangle does not lie entirely within one of the forms, **WGCHRBLT** clips the character rectangle 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, PSR(OVR) is set to 1.

WGCHRBLT microcode is designed to optimize its speed if certain conditions are met. For the fastest possible execution, you should ensure that the following 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.

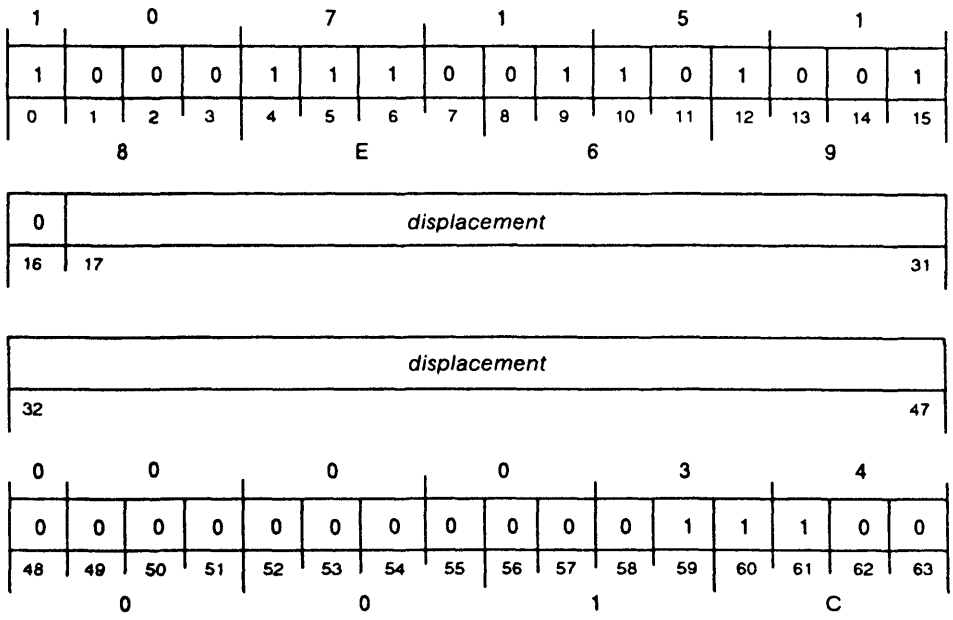
Example

```
;The following subroutine allows access to the WGCHRBLT instruction
;from any common code compiler.

;$WGCHRBLT (source_form, destination_form, packet)
$WGCHRBLT:
      WSAVR       0
      XWLDA       0,@ARG1,3
      XWLDA       1,@ARG2,3
      XLEF        2,@ARG3,3
      WGCHRBLT    GI_GIS2_TRAP
      WRTN
```

Load Cursor Descriptor — WGLDCURS

Privileged Graphics Instruction

WGLDCURS *displacement*



Function: Write updated cursor block to cursor descriptor memory
Update 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 doublewords moved is determined by the new cursor type. If the cursor is invisible, **WGLDCURS** will move only one doubleword. The cursor descriptor defines the size and position of the cursor. Note that the Load Form instruction (**WGLFORM**) loads the original cursor block into memory.

Arguments

displacement Nonindirectable PC-relative offset to runtime routine that performs the **WGLDCURS** function (if hardware support is currently unavailable). Effective address generated by instruction confined to current segment.

Registers, Flags, and Stacks

AC0 Before execution, contains either physical address of cursor-drawing database or -1.

If processor does not support hardware cursor drawing, then AC0 should contain -1. If AC0 is not -1, then **WGLDCURS** takes an unimplemented instruction trap after moving the cursor descriptor packet.

If processor supports hardware cursor drawing, then AC0 should contain physical address which points to the cursor-drawing database. This database describes the shape and color of the cursor and its format. If AC0 contains -1, then the processor assumes the cursor-drawing database has 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
PC	PC + 4
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 nonindirectable PC-relative offset). The processor uses E as the address of a runtime routine that performs the **WGLDCURS** function. The **WPOPJ** instruction exits from this software emulator.

WGLFORM
: Use the Load Form instruction to load the initial cursor block into memory.

Exceptions

If cursor drawing is not implemented, this instruction takes an unimplemented instruction trap when AC0 is not -1.

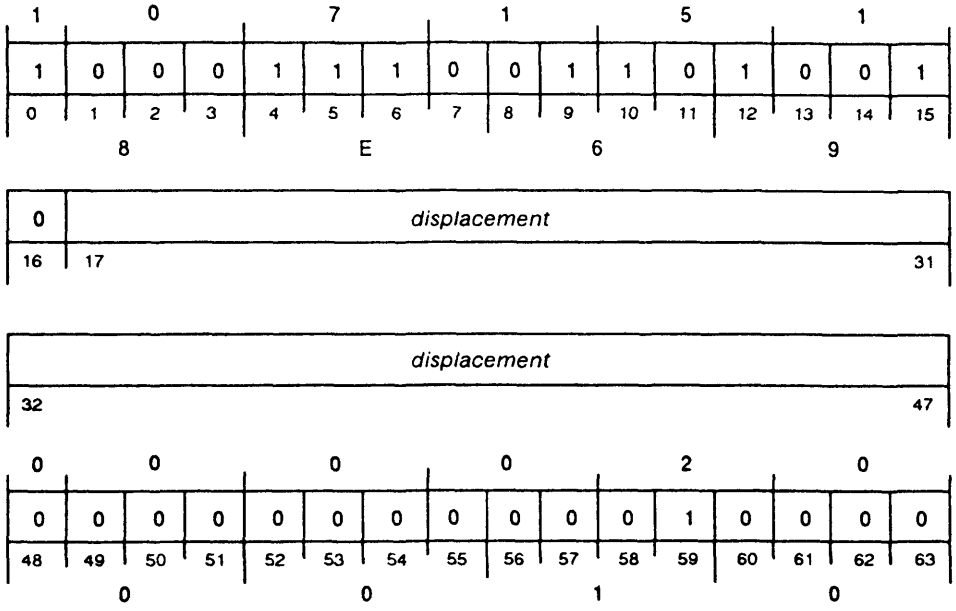
Example

```
WGLDCURS
```

Load Form WGLFORM

Privileged Graphics Instruction

WGLFORM *displacement*



Function: form → form cache memory

Parameters: AC0 = operating system key → unchanged
AC1 = user 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 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 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 user form ID of 0 in AC1 (invalid form).

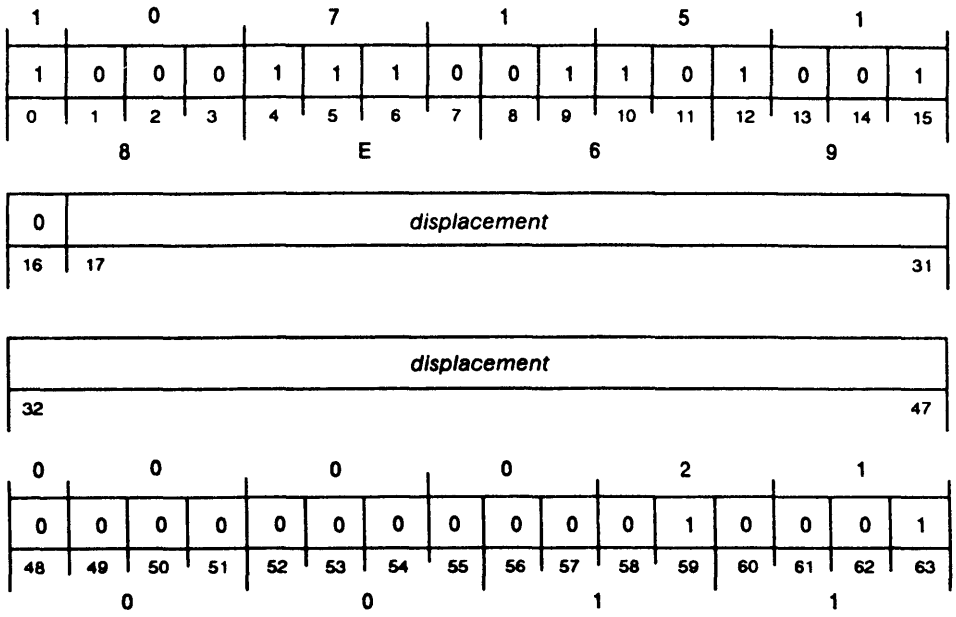
Example

```
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 form ID → unchanged
AC2 = ? → unchanged

WGPFORMS removes one or more forms from the form cache memory. The instruction 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 0.

if purging all forms, contains nonzero.

After execution, contents unchanged.

AC1 Before execution,

if AC0 = 0, contains 32-bit user 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

WSUB Use **WSUB** 0,0 to place a 0 in AC0.

Load with immediate
Use these instructions to place the user 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.

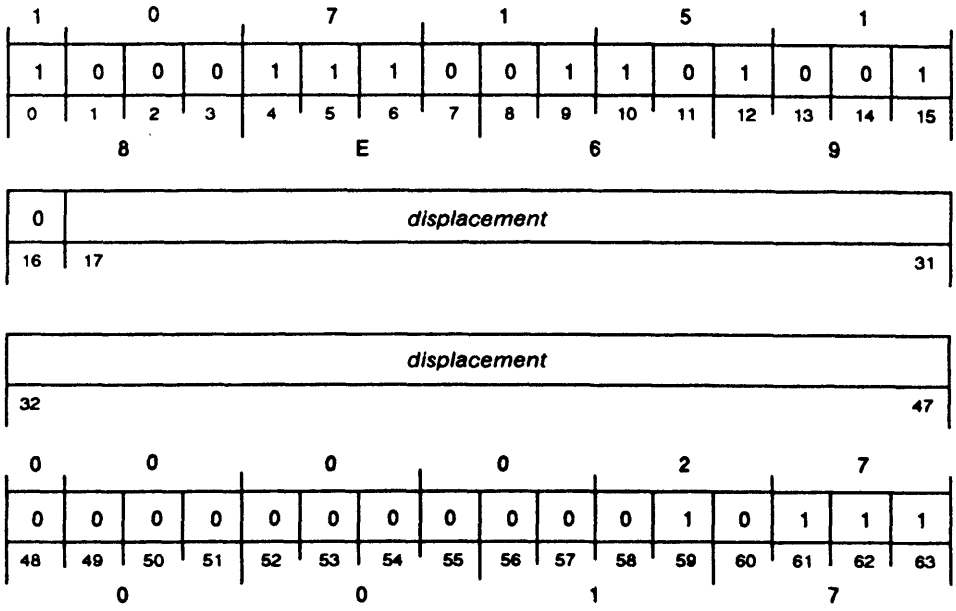
Example

```
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.

The setting of bits 4 and 5 of the line control doubleword (LINE_CTRL) in the attribute block specify whether the contents of the **WGPLINE** packet or the form's attribute block define the:

- foreground and background colors (packet or form's foreground and background color attributes),
- line style — dotted, dashed, or other nonsolid lines (packet or form's attribute block line style doubleword).
- contiguous or noncontiguous line segments and attributes for each segment (packet or attribute block's line control doubleword).

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 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.

The format of the packet depends on bits 4 and 5 of the line control doubleword in the attribute block. Coordinates are signed 32-bit integers (N is the number in AC0):

Line Control Bits
4 5

0 0 Contiguous line, use previously defined attributes. Packet consists of 2*N+2 doublewords.

Doubleword # (decimal)	Contents
1	X coordinate of initial endpoint.
2	Y coordinate of initial endpoint.
3	X coordinate of first endpoint.
4	Y coordinate of first endpoint.
.	...
2N+1	X coordinate of Nth (final) endpoint.
2N+2	Y coordinate of Nth (final) endpoint.

0 1 Noncontiguous line, use previously defined attributes. Packet consists of 4*N doublewords.

Doubleword # (decimal)	Contents
1	X coordinate of first initial endpoint.
2	Y coordinate of first initial endpoint.
3	X coordinate of first final endpoint.
4	Y coordinate of first final endpoint.
.	...
4N-3	X coordinate of Nth initial endpoint.
4N-2	Y coordinate of Nth initial endpoint.
4N-1	X coordinate of Nth final endpoint.
4N	Y coordinate of Nth final endpoint.

1 0 Contiguous line, contains new attributes. Packet consists of 5*N+2 doublewords.

Doubleword # (decimal)	Contents
1	X coordinate of initial endpoint.
2	Y coordinate of initial endpoint.
3	Foreground color for first line segment.
4	Background color for first line segment.
5	Line style for first line segment.
6	X coordinate of first endpoint.
7	Y coordinate of first endpoint.
. . .	(Repetition of foreground color, background color,
. . .	line style, and X and Y endpoint coordinates)
5N-2	Foreground color for Nth line segment.
5N-1	Background color for Nth line segment.
5N	Line style for Nth line segment.
5N+1	X coordinate of Nth (final) endpoint.
5N+2	Y coordinate of Nth (final) endpoint.

1 1 Noncontiguous line, contains new attributes. Packet consists of 7*N doublewords.

Doubleword # (decimal)	Contents
1	X coordinate of first initial endpoint.
2	Y coordinate of first initial endpoint.
3	Foreground color for first line segment.
4	Background color for first line segment.
5	Line style for first line segment.
6	X coordinate of first final endpoint.
7	Y coordinate of first final endpoint.
. . .	(Repetition of initial endpoint X and Y coordinates,
. . .	foreground color, background color,
. . .	line style, and final endpoint X and Y coordinates)
7N-6	X coordinate of Nth initial endpoint
7N-5	Y coordinate of Nth initial endpoint
7N-4	Foreground color for final line segment.
7N-3	Background color for final line segment.
7N-2	Line style for final line segment.
7N-1	X coordinate of Nth final endpoint.
7N	Y coordinate of Nth final endpoint.

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 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

If bit 4 of the line control word is 1, then the attribute block may be modified depending on the contents of the **WGPLINE** packet. The three attributes are therefore considered undefined during and immediately after execution of the **WGPLINE** instruction. Executing **WGPLINE** on the same form in a nonsequential manner produces undesirable results.

If bit 4 of the line control word is 0, the line style for the first line segment in the packet is LINE_STYLE in the attribute block. Each successive line segment uses a rotated version of the LINE_STYLE based on the length of the previous line segment. The attributes for the vertex of contiguous line segments are derived from the attributes for the trailing line segment.

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 PSR(OVR) is set to 1.

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.

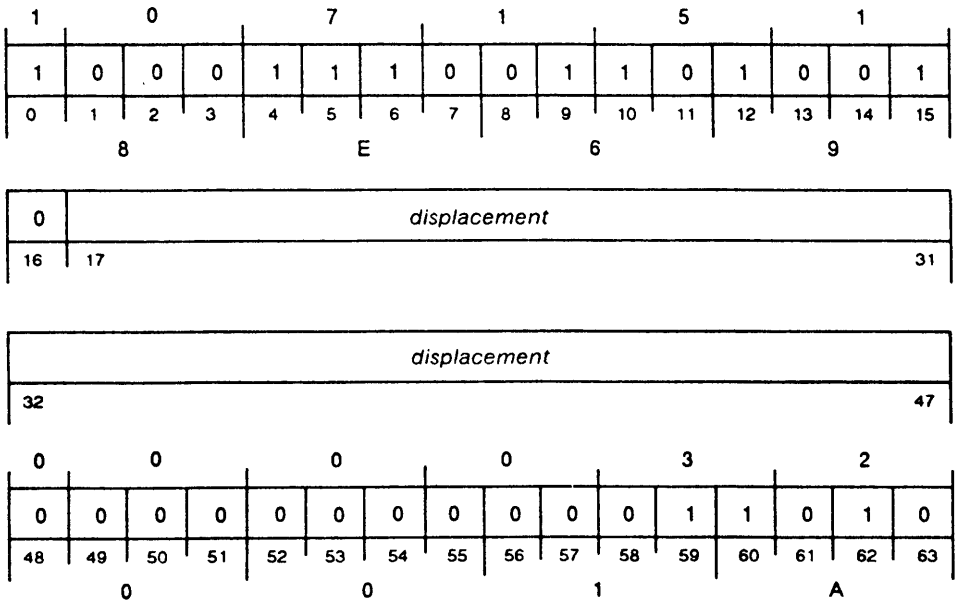
Example

```
NLDAI    2,AC0           ;Place the number of line segments (2)
XWLDA    AC1,DEST_FORM   ;into AC0. Load the contents of
XLEF     AC2,PACKET      ;DEST_FORM into AC1. Put the packet
WGPLINE  NOGIS           ;address into AC2. Execute 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 current segment.

Registers, Flags, and Stacks

AC0 Before execution, contains one of the following attribute numbers:

Attribute Number (decimal)	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.

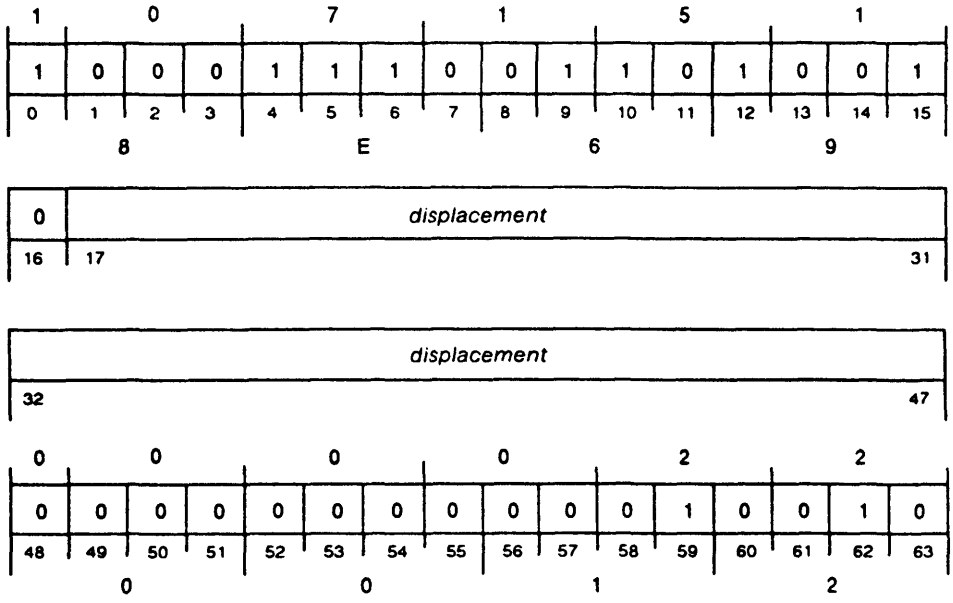
Example

```
;The following subroutine allows access to the WGRDATTR instruction
;from any common code compiler.

;attr = $WGRDATTR (index, destination_form)
$WGRDATTR:
        WSAVR       0
        XWLDA       0,@ARG1,3
        XWLDA       1,@ARG2,3
        XLEF        2,@SAVEAC0,3
        WGRDATTR    GI_GIS2_TRAP
        WRTN
```

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 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 a set of eight unsigned 32-bit integers, left justified, as follows:

Doubleword #	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 SCP command 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 Write Palette (**WGWRPAL**) instruction.

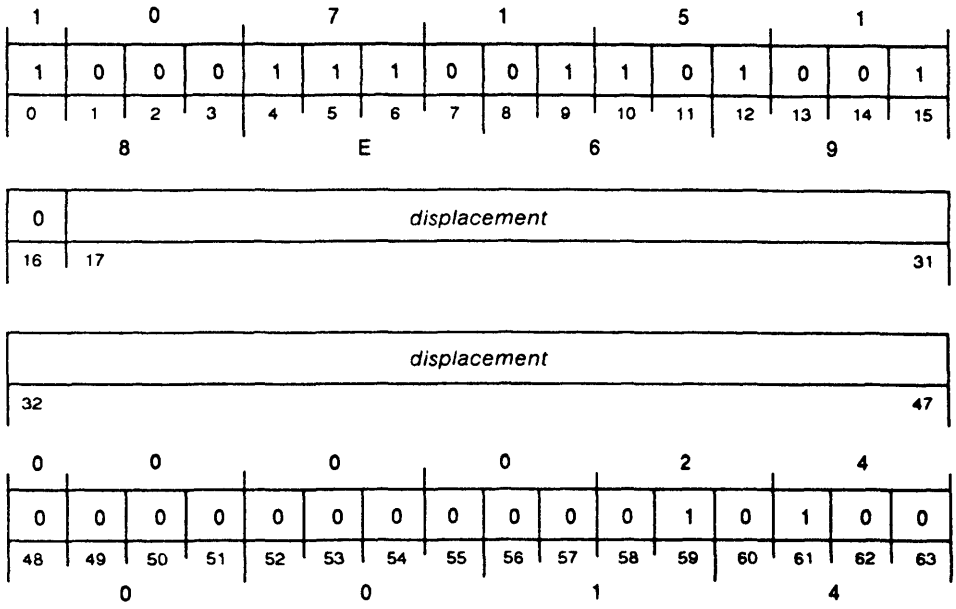
Example

```
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 instruction performs a logical AND of this pixel value with the form's form mask, and places the result 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 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:

Doubleword #	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 values are outside the range of the form, AC0 is unchanged.

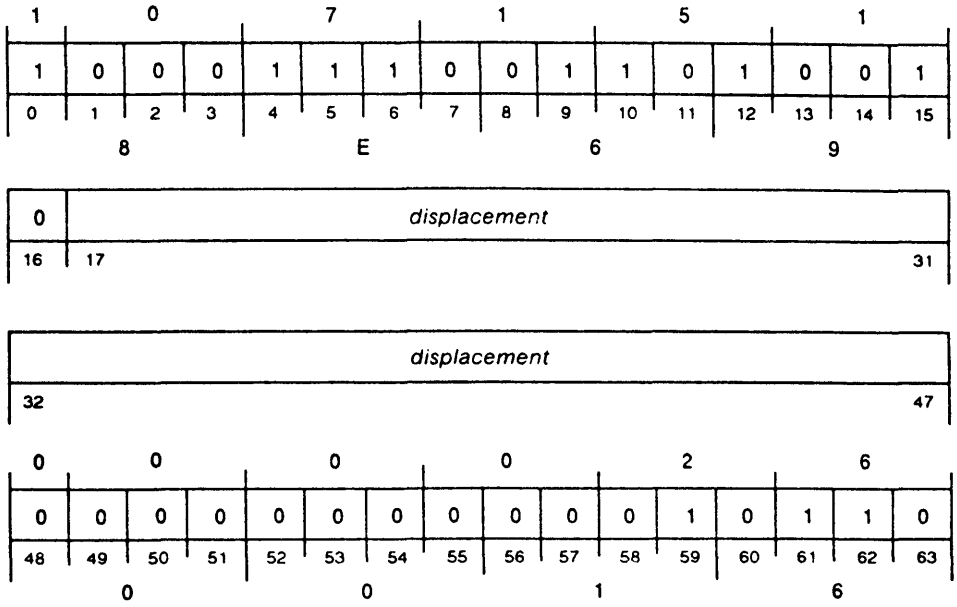
Example

```
;The following subroutine allows access to the WGRDPIXL instruction
;from any common code compiler.

;pixel = $WGRDPIXL (destination_form, packet)
$WGRDPIXL:
      WSAVR       0
      XWLDA       1,@ARG1,3
      XLEF        2,@ARG2,3
      WGRDPIXL    GI_GIS2_TRAP
      XWSTA       0,@SAVEAC0,3
      WRTN
```

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 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:

Doubleword #	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 (doublewords 3 or 4 in the packet) is specified as 0, **WGRFLOOD** has no effect.

If an overdraw condition occurs, PSR(OVR) is set to 1.

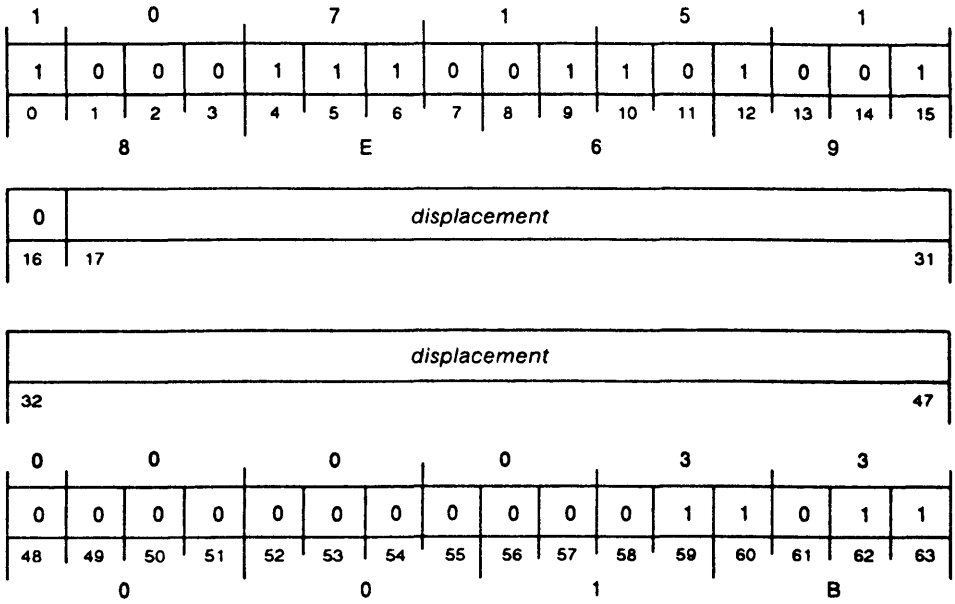
Example

```
;The following subroutine allows access to the WGRFLOOD instruction
;from any common code compiler.

;$WGRFLOOD (pixel, destination_form, packet)
$WGRFLOOD:
        WSAVR       0
        XWLDA       0,@ARG1,3
        XWLDA       1,@ARG2,3
        XLEF        2,@ARG3,3
        WGRFLOOD    GI_GIS2_TRAP
        WRTN
```

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 current segment.

Registers, Flags, and Stacks

AC0 Before execution, contains one of the following attribute numbers:

Attribute Number (Decimal)	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 occurs.

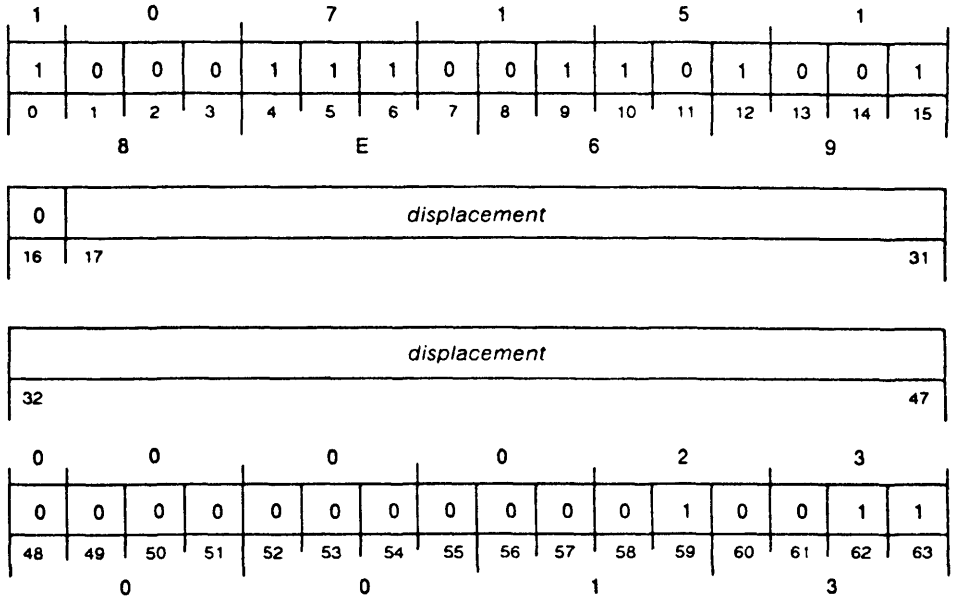
Example

```
;The following subroutine allows access to the WGWRATTR instruction
;from any common code compiler.

;$WGWRATTR (index, destination_form, attribute)
$WGWRATTR:
        WSAVR       0
        XWLDA       0,@ARG1,3
        XWLDA       1,@ARG2,3
        XLEF        2,@ARG3,3
        WGWRATTR    GI_GIS2_TRAP
        WRTN
```

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 a 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:

Doubleword #	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 SCP command to place the microcode ID into AC1.

LLEF Use this instruction to place the 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

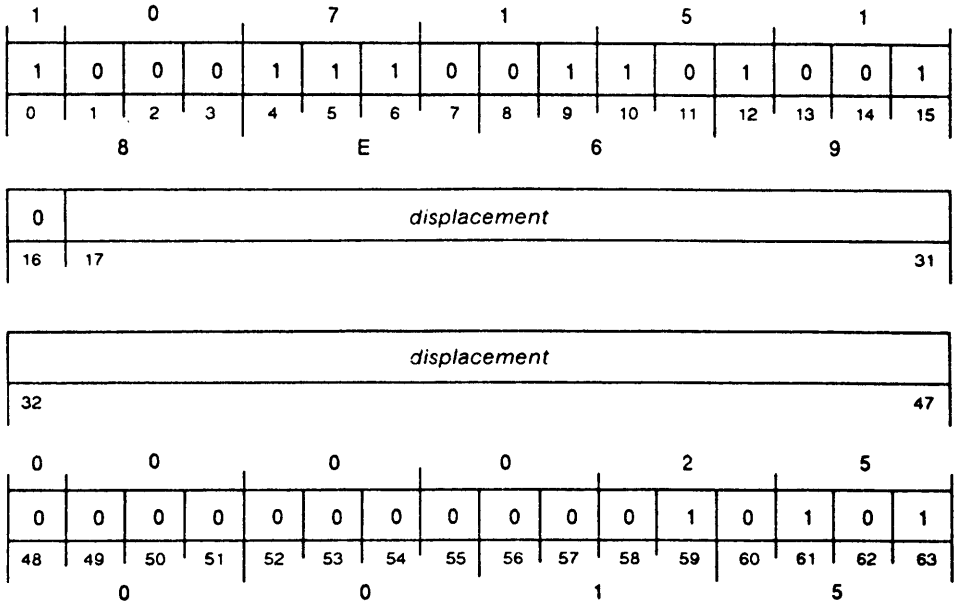
Example

```
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.

Packet consists of two signed, 32-bit integers as follows:

Doubleword #	Contents
1	X coordinate of pixel.
2	Y coordinate of pixel.

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 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

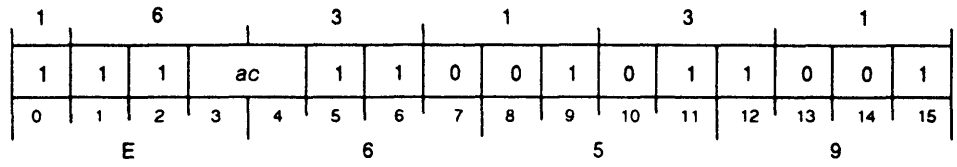
If the pixel specified is write inhibited or lies outside the form, **WGWRPIXL** has no effect.

Example

```
;The following subroutine allows access to the WGWRPIXL instruction
;from any common code compiler.

;$WGWRPIXL (pixel, destination_form, packet)
$WGWRPIXL:
        WSAVR       0
        XWLDA       0,@ARG1,3
        XWLDA       1,@ARG2,3
        XLEF        2,@ARG3,3
        WGWRPIXL    GI_GIS2_TRAP
        WRTN
```

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 2 and rounds the result toward 0.

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

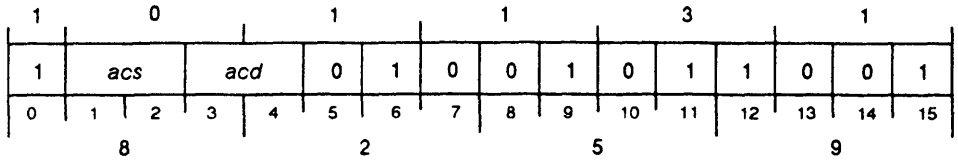
Example

```
;Subroutine to compare two strings.
;
;Strings are assumed to be word aligned and 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 overflows.
PC	PC + 1
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

INC	Increment

Exceptions

None

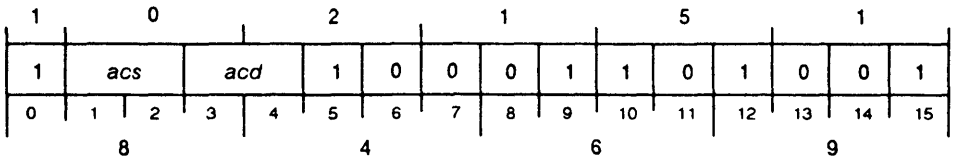
Example

```
;Subroutine to compare two strings.
;
;Strings are assumed to be word aligned and 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*. The instruction sets a bit position in the result to 1 if the corresponding bit position in one or both operands contains a 1; otherwise the result bit is set 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

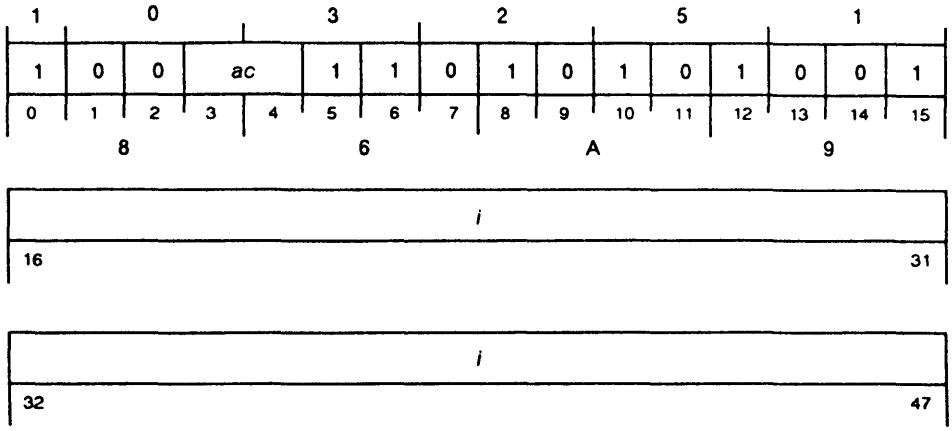
None

Example

```
WIOR  0,1          ;Inclusively 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 *ac*, placing the result in *ac*.

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

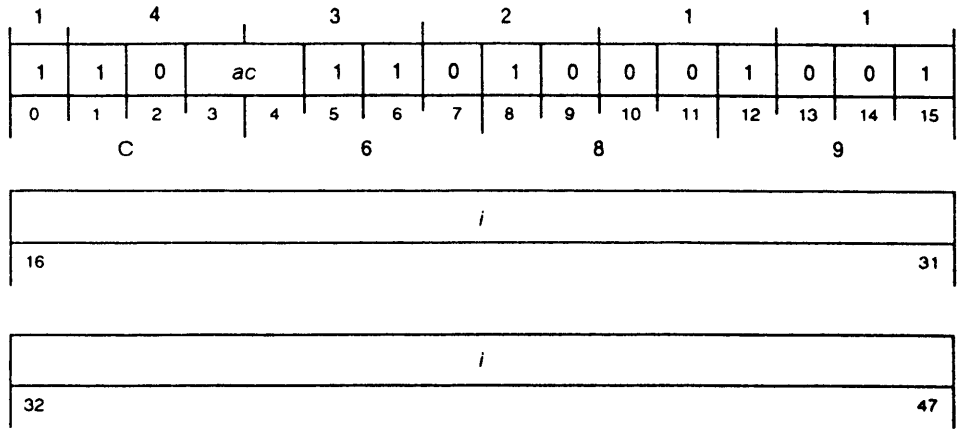
None

Example

```
WIORI 016000000000,0          ;Force ring bits in AC0 to be ring 7.
```

Wide Load with Wide Immediate — WLDAI

WLDAI *i,ac*



Function: $i \rightarrow ac$

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

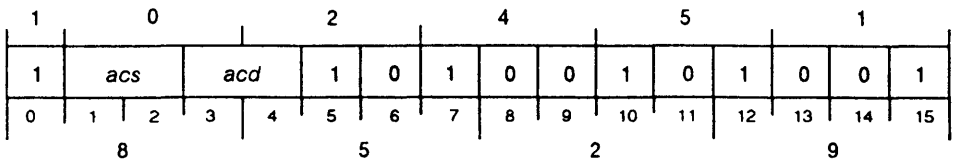
None

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 0-23 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 (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

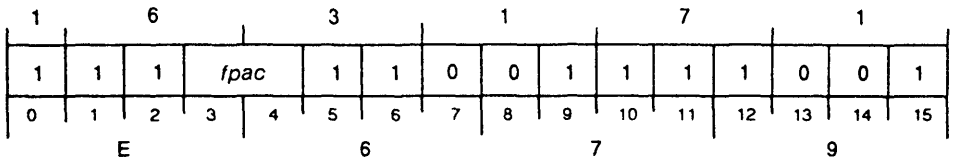
None

Example

```
;Convert lowercase input to uppercase.
;
;       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 lowercase.
.DWORD              "A+40       ;Lower limit for compare.
.DWORD              "Z+40       ;Upper limit for compare.
        WBR         NOTLOW      ;Not lowercase.
        WNADI       -40,2       ;Yes, lowercase, convert to uppercase.
```

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,. The instruction 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 the mantissa of *fpac*.

It then sets to 0 each low-order byte in *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 positions 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

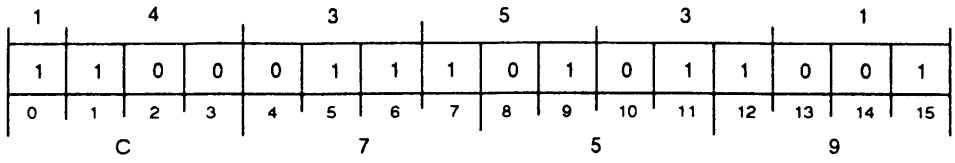
An attempt to load a –0 sets *fpac* to true zero.

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 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 memory. The instruction expands the integer to 32 digits, divides the result into four 8-digit integers, and converts the integers to floating-point numbers. **WLDIX** then loads the floating-point numbers into individual floating-point accumulators.

The sign of the 32-bit integer is stored in each floating-point accumulator unless the integer in that FPAC consists of all zeros, in which case the FPAC 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 instruction expands the integer 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

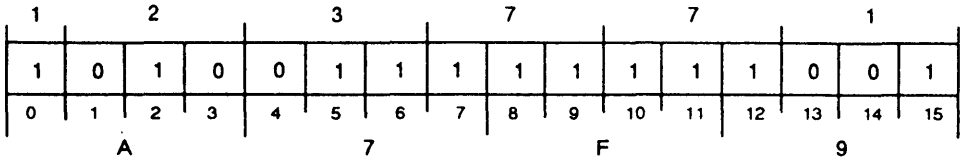
Example

```
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 integer
                    ;to four 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 doublewords 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; $2000_8–2777_8$ indicate DCH slots.

If machine implements more than one I/O channel, AC0(18–20) contains channel number.

For each map slot loaded, AC0(21–31) is incremented by 1.

After execution, refers to map slot following last slot loaded.

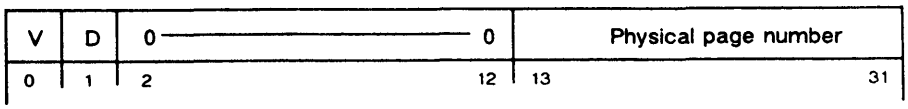
AC1(16–31) Before execution, contains unsigned 16-bit integer specifying number of map slots to be loaded.

For each map slot loaded, AC1 is decremented by 1.

After execution, contains 0.

AC2 Before execution, contains effective address of first doubleword to be loaded into slots.

Contents of doublewords are in following format:



Bits	Name	Contents or Function
0	V	Valid Set to 0: valid access. 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 doubleword 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 2.

After execution, contains address of word following last doubleword 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 performs no operation.

If an active device causes the detection of an invalid map entry,

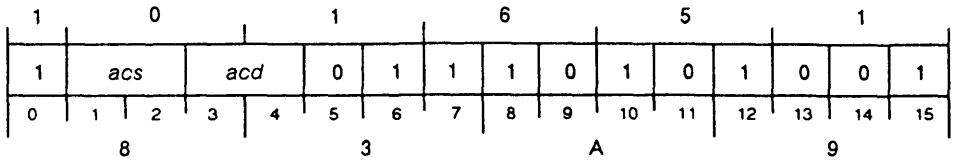
for BMC: active BMC requesting device flagged.
for DCH: bit 4 of IOC Status Register set to 1.

Example

```
XWLDA 0,SLOT_NUM              ;Get the starting slot number.
NLDAI 4,1                     ;Load four map slots.
XLEF  2,SLOT_CONTENTS         ;Get address of data to load.
WLMP                          ;Load all four map slots.
      ...
SLOT_CONTENTS:
      .DWORD      0           ;Data for four 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 this 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

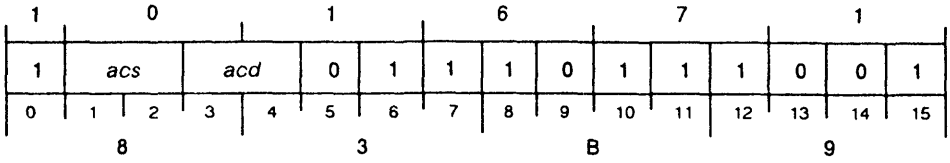
None

Example

```
NLDAI  0777,1        ;Load a bit pattern into AC1.
WSUB   0,0           ;Set AC0 to zero.
WLOB   1,0           ;Add 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 this count to the 32-bit integer in *acd*. The instruction 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

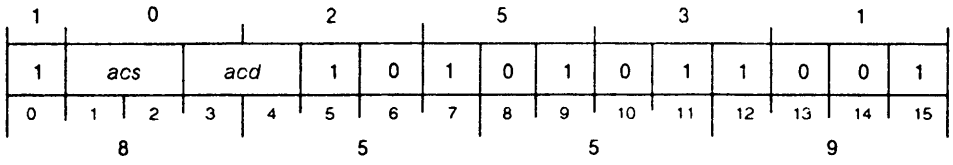
None

Example

```
NLDAI  0777,1        ;Load a bit pattern into AC1.
WSUB   0,0           ;Set AC0 to zero.
WLRB   1,0           ;Add 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

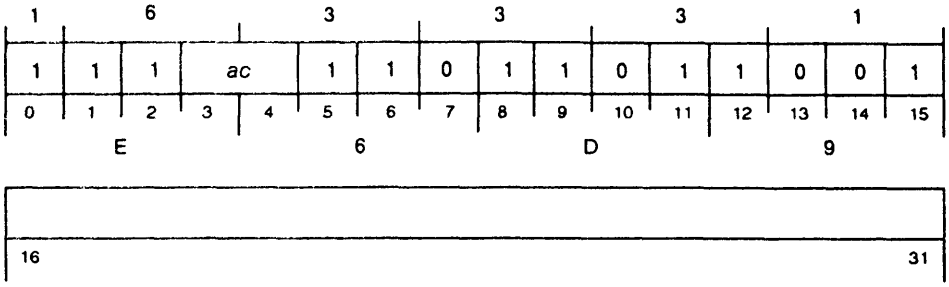
None

Example

```
NLDAI  -4,3          ;Load a -4 into AC3, and then
WLSH   3,1           ;divide contents of AC1 by 16.
```

Wide Logical Shift with Narrow Immediate **WLSHI**

WLSHI *i,ac*



Function: shift $ac(i[+ = \text{left}, - = \text{right}]) \rightarrow ac$

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

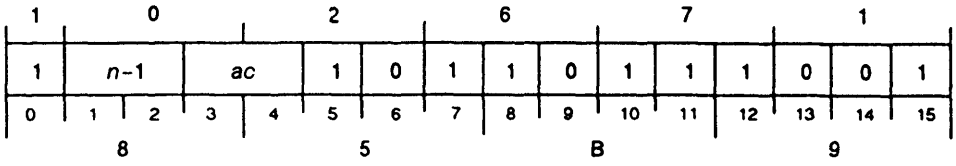
None

Example

```
XLEF    0,VARIABLE      ;Get the address of the variable.
WANDI   01777777777,0   ;Mask to just the offset portion.
WLSHI   -10.,0          ;Shift right 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*). Note that **WLSI** 1,*ac* converts a word pointer in *ac* to a byte pointer.

Arguments

n	Integer in range 1–4. Since Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field, you 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 (converts a byte pointer to a word pointer).

Exceptions

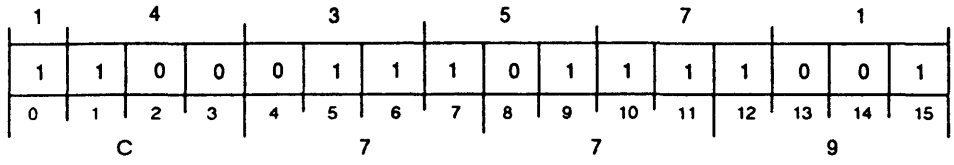
None

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 memory and returns a code to AC1 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 code as follows:

Code	Value of Number
+1	Positive nonzero
-1	Negative nonzero
0	Positive zero
-2	Negative zero

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

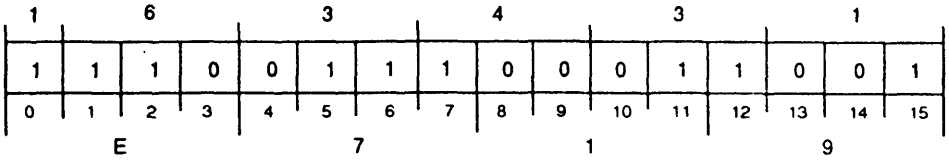
None

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 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 doubleword in memory.

The instruction reads the doubleword addressed by AC2 and performs an exclusive OR of this doubleword with the contents of AC0. **WMESS** then performs a logical AND of this value with the contents of AC3. If the final result

equals 0, the instruction swaps the values in AC1 and memory, and executes the second word following the **WMESS** instruction (successful return).

does not equal 0, the instruction places the memory value into AC1, and exectues the word following the **WMESS** instruction (unsuccessful return).

WMESS is guaranteed to be indivisible when the memory value is doubleword 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 doubleword 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)	Final result ≠ 0.
	PC + 2	(successful return)	Final result = 0.
PSR	Unchanged		
Stack	Unchanged		

Related Instructions

LLEF, XLEF Use a load effective address instruction to calculate and load the doubleword memory address into AC2.

WBR Use the Wide Branch instruction to jump to the code for handling the unsuccessful return.

Exceptions

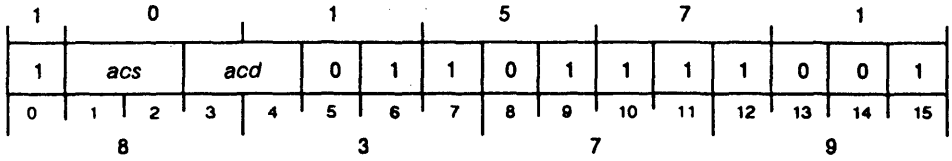
None

Example

```
;In this example, 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.

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 doubleword
                    ;to test.
XWLDA 3,MASK        ;Load AC3 with mask bits for the AND
WMESS               ;comparison.
WBR   NOGOOD        ;Invalid comparison.
      .             ;Valid comparison.
      ...           ;

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

;The following example shows how WMESS can be used to indivisibly add
;a specified value to memory and return the updated value.
;
        XLEF 2,COUNT ;Load AC2 with address of doubleword 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*. If *acs* and *acd* are specified as the same accumulator, **WMOV** performs no operation.

Arguments

acs	Before execution, contains 32-bit value. After execution, contents unchanged.
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

MOV	Move

Exceptions

None

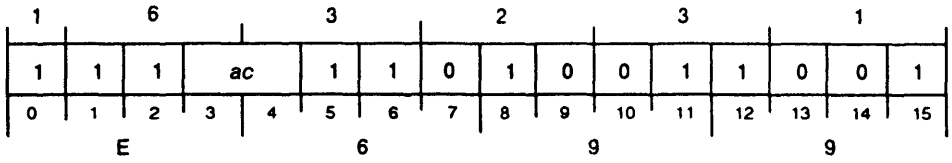
Example

```
;This subroutine removes an element from a linked list queue. It is
;the responsibility of the caller to set the transition bit, if
;necessary.
;
;Calling conventions:          XJSR PDEQ
;                    <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 0 into bit 0. This instruction can convert a byte pointer into a word pointer.

Arguments

ac	Before execution, contains 32-bit value.
	After execution, contains 32-bit 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

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

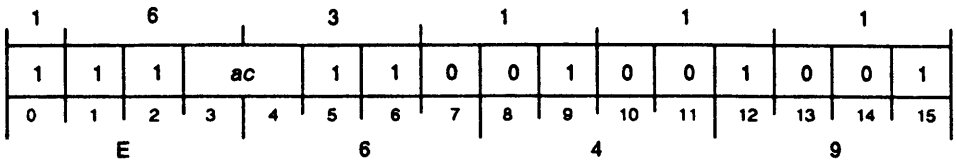
None

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.

The instruction 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. **WMSP** then checks for a fixed-point overflow resulting from the shift. If no overflow occurs, the temporary value is stored as the new value of the stack pointer.

Arguments

ac	Before execution, contains signed 32-bit integer specifying number of doublewords 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

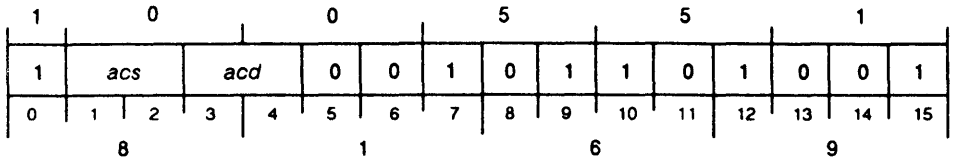
If the shifted value would produce a fixed-point overflow, the stack pointer is not modified. Instead, a return block is pushed, using the original stack pointer as the reference, and the processor jumps to the wide stack fault handler routine. Upon return from the fault handler, processing resumes with the **WMSP** instruction.

Example

```
NLDAI  4,0          ;Get a constant 4.
WMSP   0            ;Increment the stack pointer by 4 doublewords.
```

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	Signed Multiply
WMULS	Wide Signed Multiply
NMUL	Narrow Multiply

Exceptions

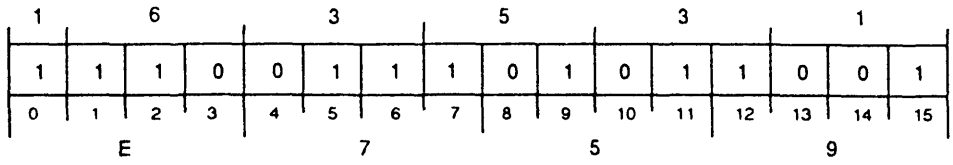
If result is outside the range, -2,147,483,648 to +2,147,483,647 inclusive, PSR(OVR) is set to 1, and *acd* contains least significant 32 bits of result.

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

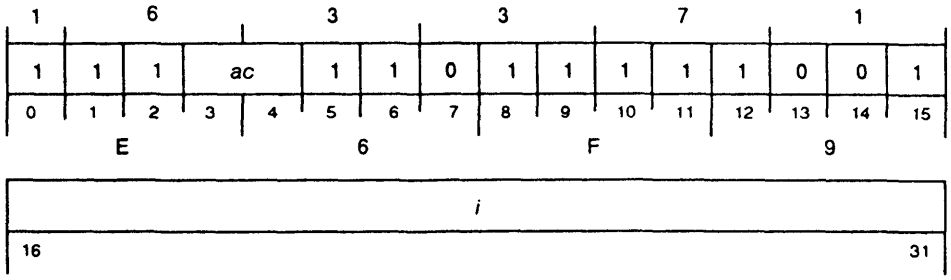
None

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 carry → 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 *ac*.

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

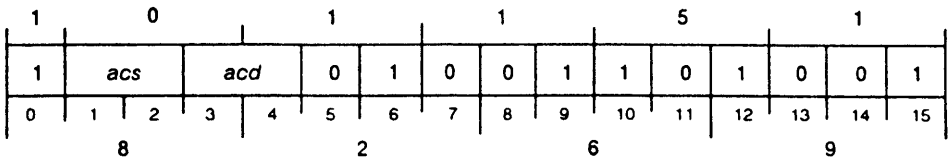
None

Example

```
    ;Convert lowercase input to uppercase.
    ;       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 lowercase.
        .DWORD      "A+40       ;Lower limit for compare.
        .DWORD      "Z+40       ;Upper limit for compare.
        WBR NOTLOW              ;Not lowercase.
        WNADI       -40,2       ;Yes, lowercase, convert to uppercase.
```

Wide Negate — WNEG

WNEG *acs,acd*



Function: $-acs \rightarrow acd$

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 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

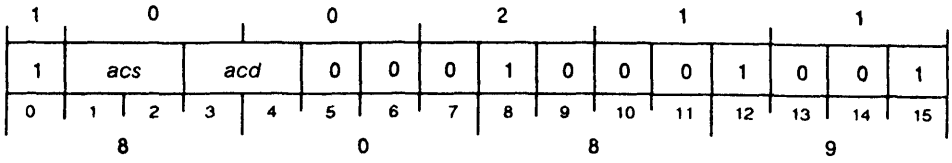
If the largest negative 32-bit integer (10000000000_8) is negated, PSR(OVR) is set to 1.

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) doublewords → wide stack
1st stack doubleword → *acs*
nth stack doubleword → *acd*
wsp-2*n → wsp

Parameters: None

NOTE: If *acs* is *acd*, 1 doubleword is popped.

WPOP pops doublewords off the wide stack and loads them into the specified accumulators. The number of doublewords 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 doubleword is popped and it is placed in the specified accumulator.

WPOP decrements the contents of the wide stack pointer by twice the number of doublewords popped and then checks for stack underflow.

Arguments

acs	Starting accumulator of set; receives first doubleword popped from stack.
acd	Ending accumulator of set; receives last doubleword 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

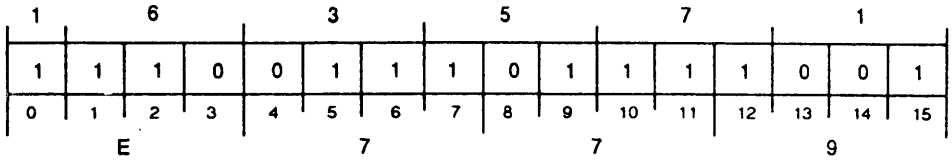
None

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 doublewords (wide return block)
wsp → wsp-(6th doubleword (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). The instruction pops six doublewords off the wide stack and places them in the following locations:

Doubleword Popped	Destination
1	Bit 0 to Carry; bits 1-31 to PC
2	AC3
3	AC2
4	AC1
5	AC0
6	Bits 0-15 to PSR, bit 16 is 0, bits 17-31 are dependent on what pushed the return block.

- 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 ring crossing to the outer ring 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 doubleword 6, bits 17-31). Execution continues with the location addressed by the program counter.

Arguments

None

Registers, Flags, and Stacks

AC0-AC3	After execution, contains doublewords 5 through 2 popped from stack.
Carry	After execution, contains bit 0 of first doubleword popped from stack.
Overflow	Unaffected
PC	After execution, contains bits 1-31 of first doubleword popped from stack.
PSR	After execution, contains bits 0-15 of sixth doubleword popped from stack.
Stack	Wide-stack pointer decremented by six doublewords.

Related Instructions

POPB Pop Block

Exceptions

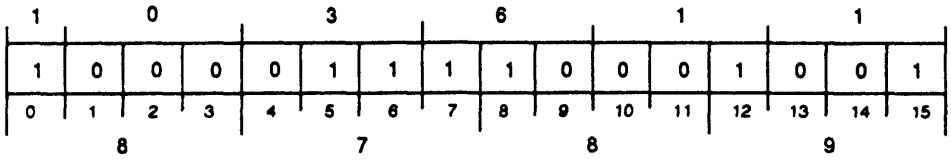
Two exceptions may be encountered: a protection fault, if a return involves an inward ring crossing; or a stack underflow fault, if the number of words popped exceeds the lower stack limit. When a fault occurs, the processor jumps to the appropriate fault handler. Note that the return block pushed as 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 six doublewords off the stack, loading
                ;AC0-AC3, CRY, the PSR, and the PC.
```

Wide Pop PC and Jump — WPOPJ

WPOPJ



Function: top stack doubleword → PC
wsp → wsp-2

Parameters: None

WPOPJ pops a doubleword 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 (the processor sets the ring bits to the current segment). 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 2; 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.

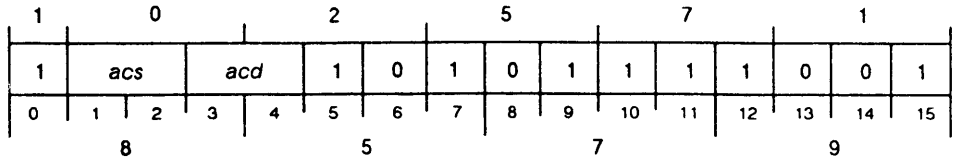
Example

```
;Subroutine to compare two strings
;
;Strings are assumed to be word aligned and 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* through *acd* → stack
+(1-4) doublewords → wide stack
wsp → wsp +2*(1-4)

Parameters: None

NOTE: If *acs* is *acd*, 1 ac is pushed.

WPSH pushes the 32-bit contents of the specified accumulators onto the wide stack. The number of doublewords 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.

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 doubleword pushed on stack.
acd	Ending accumulator of set; contains last doubleword pushed on 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

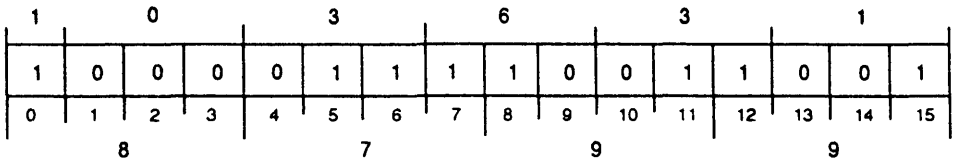
None

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 doublewords
1st through 5th doublewords (wide return block) → CRY, PC, AC3-AC0
6th doubleword → PSR,0
7th doubleword (1-15) → stack fault address
8th doubleword → wsb
9th doubleword → wsl
10th doubleword → wsp
11th doubleword → wfp

Parameters: None

WRSTR returns control from a base-level interrupt by popping 11 doublewords off the wide stack and placing them into the following locations:

Doubleword Popped	Destination
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 wide stack registers, stores the stack fault address in the wide 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 wide 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 doublewords 5 through 2 popped from stack.
Carry	After execution, contains bit 0 of first doubleword popped.

Overflow	Unaffected
PC	After execution, contains bits 1–31 of first doubleword popped.
PSR	After execution, contains bits 0–15 of sixth doubleword popped.
Stack	After execution, wide stack parameters modified as described above.

Related Instructions

POP, POPB, WPOP, WPOPB
Pop information from the stacks.

Exceptions

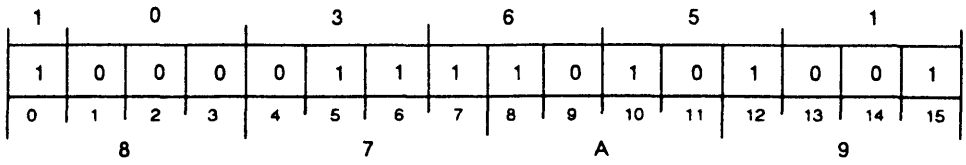
Two exceptions may be encountered: a protection fault, if a return involves an inward ring crossing; or a stack underflow fault, if the number of words popped exceeds the lower stack limit. When a fault occurs, the processor jumps to the appropriate fault handler routine. Note that the return block pushed 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 doublewords off the stack and
        ;restore processor state in returning from an interrupt.
```

Wide Return WRTN

WRTN



Function: stack → registers
stack → -6 doublewords (wide return block)
wsp-(6th doubleword (bits 17-31)*2 + 12) → wsp
AC3(popped) → wfp

Parameters: None

WRTN returns control from a subroutine (that at its entry point executed an instruction such as **WSAVS, WSAVR, WSSVS,** or **WSSVR**). The instruction does this by setting the WSP to equal the WFP and then popping six doublewords 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:

Doubleword Popped	Destination
1	Bit 0 to Carry; bits 1-31 to PC
2	AC3
3	AC2
4	AC1
5	AC0
6	Bits 0-15 to PSR, bit 16 is 0. bits 17-31 specify wide stack frame size.

- 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.

  Note that when **WRTN** returns control from a subroutine called by either the **LCALL** or **XCALL** instruction, **WRTN** also removes the number of arguments (specified by the **LCALL** or **XCALL** *argument_count*), from both the inner segment stack and the outer segment stack.

Arguments

None

Registers, Flags, and Stacks

AC0-AC3	After execution, contains doublewords 5 through 2 popped from stack.
Carry	After execution, contains bit 0 of first doubleword popped from stack.

Overflow	Unaffected
PC	After execution, contains bits 1–31 of first doubleword popped from stack.
PSR	After execution, contains bits 0–15 of sixth doubleword popped from stack.
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

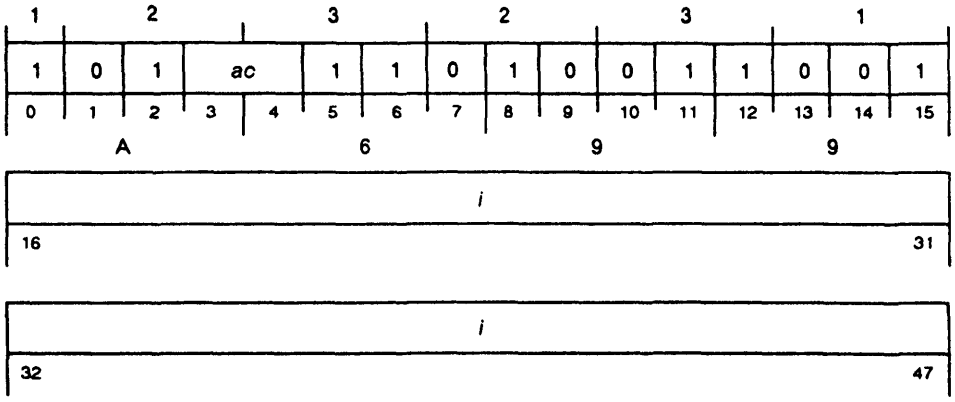
Two exceptions may be encountered: a protection fault, if a return involves an inward ring crossing; or a stack underflow fault, if the number of words popped exceeds the lower stack limit. When a fault occurs, the processor jumps to the appropriate fault handler. Note that the return block pushed as 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 removes an element from a linked list queue. It is
    ;the responsibility of the caller to set the transition bit, if
    ;necessary.
    ;
    ;Calling conventions:          XJSR PDEQ
    ;                    <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 performs a logical AND of the contents of the immediate field with the complement of the contents of *ac* 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

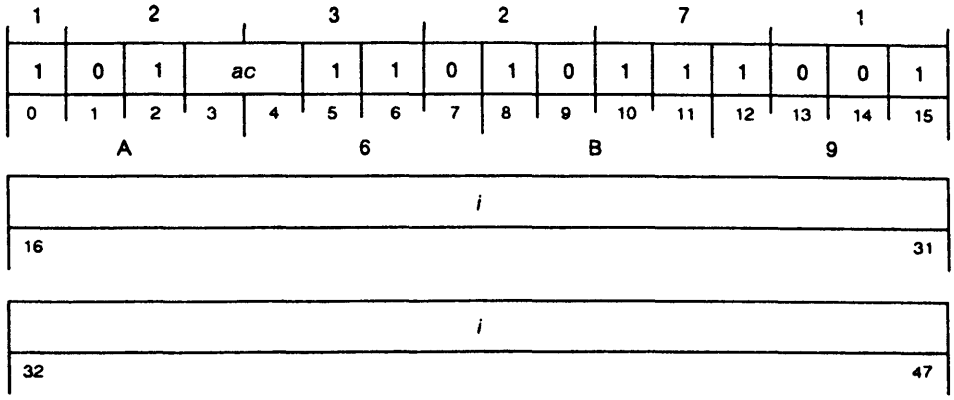
None

Example

```
    XWLDA  2,FLAGS      ;Get the flags doubleword.
    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 doubleword.
```

Wide Skip on All Bits Set in Doubleword Memory Location — WSALM

WSALM *i,ac*
(AND ≠0 return)
(AND = 0 return)



Function: If *i* AND ($\overline{ac}$) = 0 then skip
Parameters: None

WSALM performs a logical AND of the contents of the immediate field with the complement of the doubleword addressed by *ac* 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

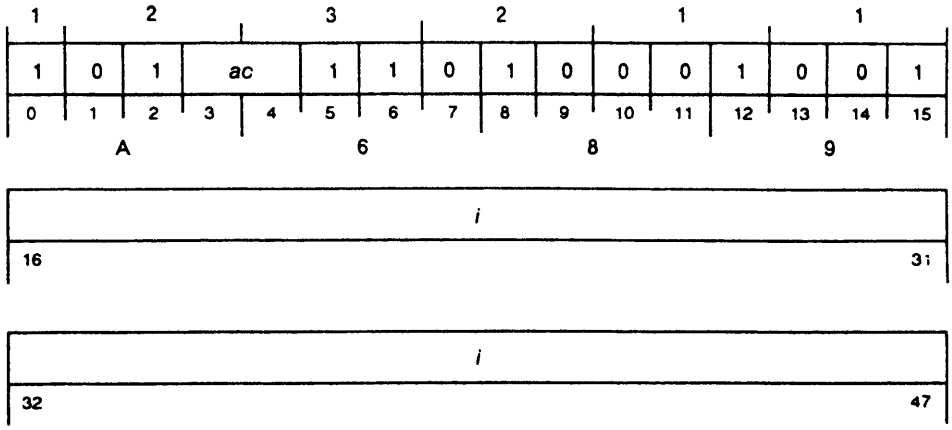
None

Example

```
    XLEF   2,FLAGS      ;Get address of the flags doubleword.
    WSALM  140002,2     ;Are bits 16, 17, and 30 all set?
    WBR    FAIL         ;No. One or more are 0.
    . . .               ;Yes. All bits are set.
FLAGS:    .DWORD 0      ;Flags doubleword.
```

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 performs a logical AND of the contents of the immediate field with the contents of *ac* 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

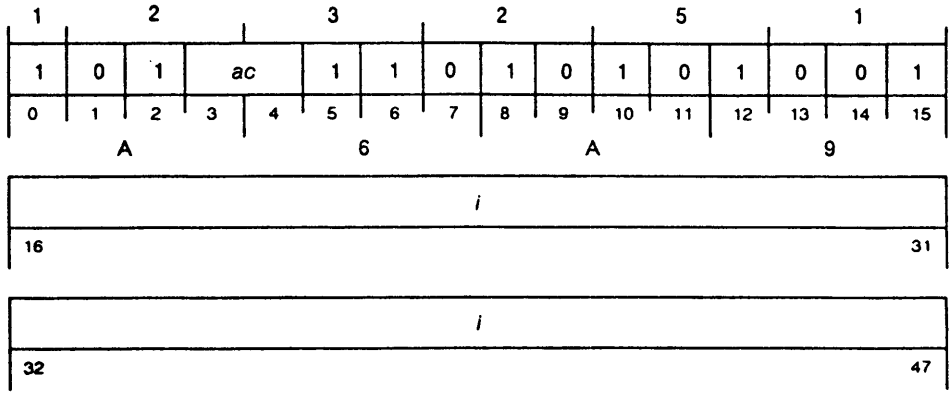
None

Example

```
    XWLDA 2,FLAGS      ;Get the flags doubleword.
    WSANA 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 doubleword.
```

Wide Skip on Any Bit Set in Doubleword Memory Location — WSANM

WSANM *i,ac*
(AND = 0 return)
(AND = non-0 return)



Function: If *i* AND (*ac*) ≠ 0 then skip
Parameters: None

WSANM performs a logical AND of the contents of the immediate field with the contents of the doubleword addressed by *ac* 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

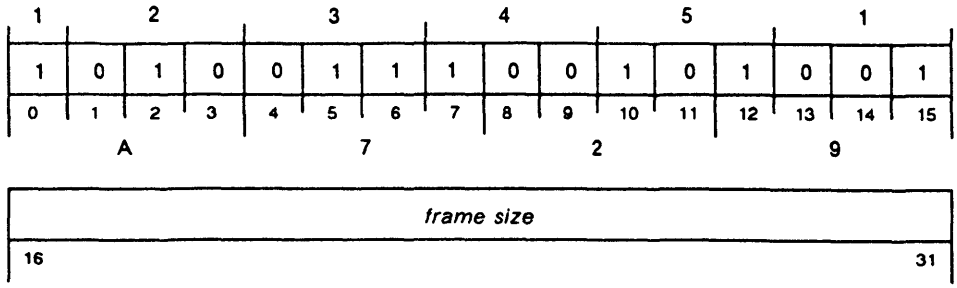
None

Example

```
      XLEF   2,FLAGS      ;Get the flags doubleword.
      WSANM  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 doubleword.
```

Wide Save/Reset Overflow Mask — WSAVR

WSAVR *frame size*



Function: 5 doublewords → wide stack (partial wide return block)
wsp(after push) → AC3
wsp(after push) → wfp
wsp + (*frame size**2) → wsp
0 → PSR(OVK)

Parameters: *frame size* = #(16-bit) → unchanged

NOTE: First doubleword should be pushed by the LCALL or XCALL instruction.

WSAVR pushes a return block of five doublewords onto the wide stack, resets the processor status register overflow mask (OVK) to 0, and increments the wide stack pointer by the *frame size*. The return block consists of the following:

Doubleword Pushed	Contents
1	AC0
2	AC1
3	AC2
4	Previous WFP
5	Carry (bit 0); AC3 bits 1-31, or return PC value for XCALL or LCALL (bits 1-31)

Note that the five values pushed may not make up all of the return block. An LCALL or XCALL instruction pushes the first doubleword of the return block, formatted as follows:

Bits 0-15 contain current PSR.
Bits 16-31 contain LCALL or XCALL *argument_count*.

After pushing the return block, the instruction places the new 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 PSR(OVK) to 0, disabling integer overflow.

Arguments

frame size — Unsigned 16-bit integer specifying size of frame area (in doublewords) for storing data on wide stack (beyond return block).

Registers, Flags, and Stacks

AC0 — Contains data for first doubleword pushed.

After execution, contents unchanged.

AC1	Contains data for second doubleword pushed. After execution, contents unchanged.
AC2	Contains data for third doubleword pushed. After execution, contents unchanged.
AC3	Contains data for bits 1–31 of fifth doubleword pushed. After execution, contains WSP (after push).
Carry	Unchanged
Overflow	Unaffected
PC	PC + 2
PSR	OVK bit set to 0.
Stack	Wide stack pointer incremented by five doublewords.

Related Instructions

SAVZ, WSAVS, WSSAVR, WSSAVS
Push a return block onto a stack.

WRTN Wide Return

Exceptions

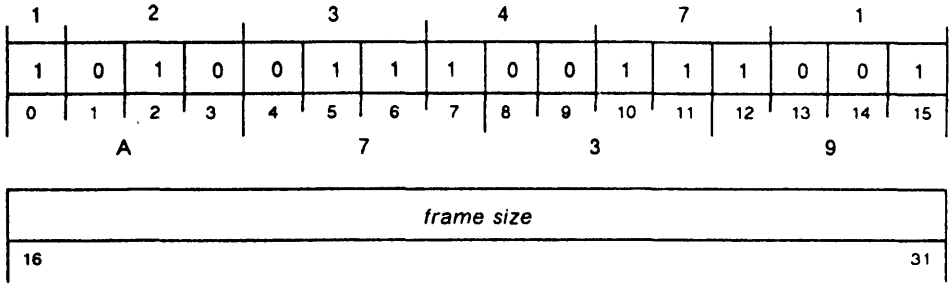
A check for stack overflow is made before the return block is pushed.

Example

```
        LCALL     SUBROUT,0,0   ;Subroutine call.
        ...
SUBROUT:
        WSAVR     2             ;Save ACs, update WSP, and allocate two
        . . .                   ;doublewords for local storage.
        WRTN                    ;Return from subroutine, restoring
                                ;ACs.
```

Wide Save/Set Overflow Mask — WSAVS

WSAVS *frame size*



Function: 5 doublewords → wide stack (partial wide return block)
wsp(after push) → AC3
wsp(after push) → wfp
wsp + (*frame size**2) → wsp
1 → PSR(OVK)

Parameters: *frame size* = #(16-bit) → unchanged

NOTE: First doubleword should be pushed by the LCALL or XCALL instruction.

WSAVS pushes a return block of five doublewords onto the wide stack, sets the processor status register overflow mask (OVK) to 1, and increments the wide stack pointer by the *frame size*. The return block consists of the following:

Doubleword Pushed	Content
1	AC0
2	AC1
3	AC2
4	Previous WFP
5	Carry (bit 0); AC3 bits 1-31, or return PC value for XCALL or LCALL (bits 1-31).

Note that the five values pushed may not make up all of the return block. An LCALL or **XCALL** instruction pushes the first doubleword of the return block, formatted as follows:

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

After pushing the return block, the instruction places the new 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 PSR(OVK) to 1, enabling integer overflow.

Arguments

frame size — Unsigned 16-bit integer specifying size of frame area (in doublewords) for storing data on wide stack (beyond return block).

Registers, Flags, and Stacks

AC0 — Contains data for first doubleword pushed.

After execution, contents unchanged.

AC1	Contains data for second doubleword pushed.
	After execution, contents unchanged.
AC2	Contains data for third doubleword pushed.
	After execution, contents unchanged.
AC3	Contains data for bits 1–31 of fifth doubleword pushed.
	After execution, contains WSP (after push).
Carry	Unchanged
Overflow	Unaffected
PC	PC + 2
PSR	OVK bit set to 1.
Stack	Wide stack pointer incremented by five doublewords.

Related Instructions

SAVZ, WSAVS, WSSAVR, WSSAVS
Push a return block onto a stack.

WRTN Wide Return

Exceptions

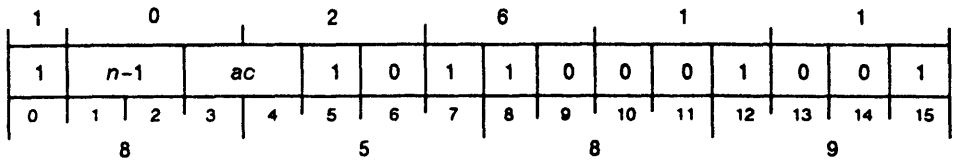
A check for stack overflow is made before the return block is pushed.

Example

```
        LCALL   SUBROUT,0,0   ;Subroutine call.
. . .
SUBROUT:
        WSAVS   2             ;Save ACs, update WSP, and allocate two
        . . .                 ;doublewords for local storage.
        WRTN                  ;Return from the subroutine, restoring
                              ;ACs.
```

Wide Subtract Immediate — WSBI

WSBI *n,ac*



Function: *ac* − *n* → *ac*
ALU carry → CRY

Parameters: None

WSBI subtracts an integer in the range of 1 to 4 from a signed 32-bit integer contained in *ac*, storing the result in *ac*.

Arguments

n	Integer in range 1–4. Since Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field, you 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 ALU overflows.
PC	PC + 1
PSR	OVR set to 1 if overflow occurs.
Stack	Unchanged

Related Instructions

SBI	Subtract Immediate
NSBI	Narrow Subtract Immediate

Exceptions

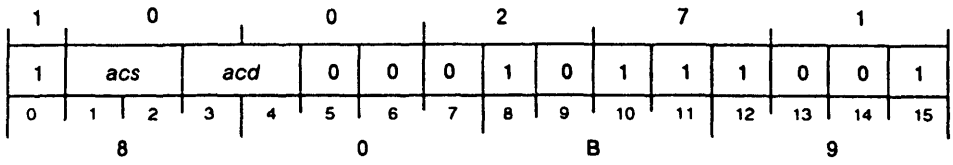
None

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 the next sequential word 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

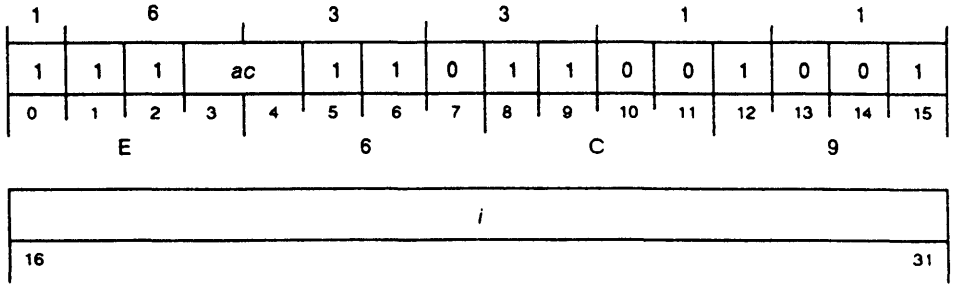
None

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 the next sequential word if they are equal.

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

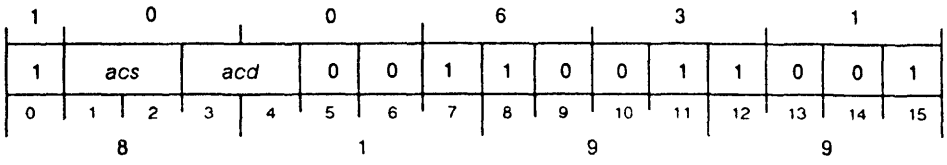
None

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 integers contained in *acs* and *acd*, and skips the next sequential word if *acs* is greater than or equal to *acd*.

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

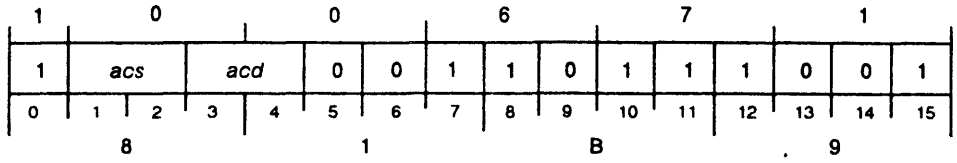
None

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 integers in *acs* and *acd*, and skips the next sequential word if *acs* is greater than *acd*.

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

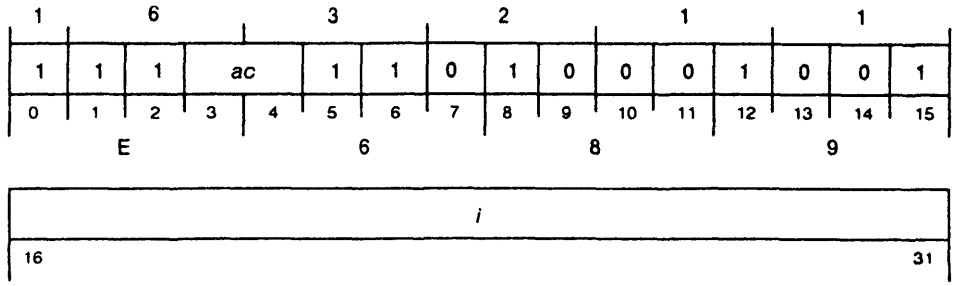
None

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 the next sequential word 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

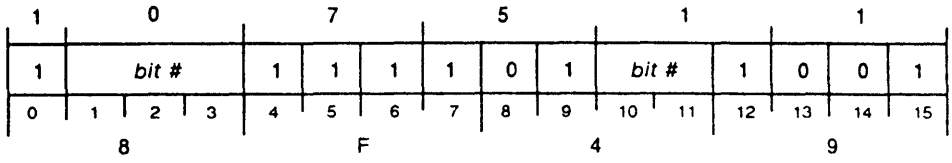
None

Example

```
WSGTI  400,2        ;Is contents of AC2 > constant 400₈?
WBR    AC2_LE       ;AC2 is less than or equal to 400₈.
. . .               ;AC2 is greater than 400₈.
```

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 the next sequential word if the bit is 1.

Arguments

bit number — Bits 1–3 and 10–11 specify bit position in AC0 (in range 0–31). Value 0_8 specifies highest-order bit and value 31_8 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 (bit = 0) PC + 2 (bit = 1)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WSKBZ — Wide Skip on Bit Set to Zero

Exceptions

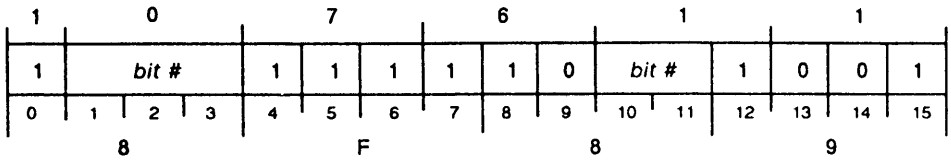
None

Example

```
XWLDA  0,FLAGS      ;Get the flags doubleword.
WSKBO  18.          ;Skip if bit 18 is 1.
WBR    NOT_ONE      ;Bit 18 is 0.
. . .               ;Bit 18 is 1.
```

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 the next sequential word if the bit is 0.

Arguments

bit number	Bits 1–3 and 10–11 specify bit position in AC0 (in range 0–31). Value 0_8 specifies highest-order bit and value 31_8 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 (bit = 1) PC + 2 (bit = 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

WSKBO	Wide Skip on Bit Set to One

Exceptions

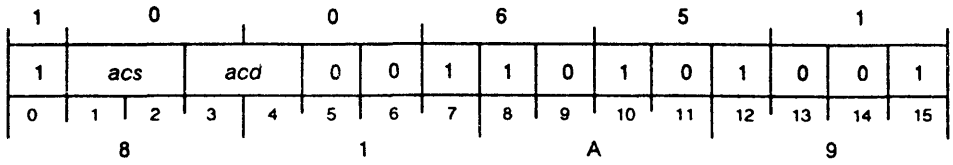
None

Example

```
XWLDA 0,FLAGS      ;Get the flags doubleword.
WSKBZ 18.          ;Skip if bit 18 is 0.
WBR   NOT_ZERO     ;Bit 18 is 1.
. . .              ;Bit 18 is 0.
```

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 integers in *acs* and *acd* and skips the next sequential word if *acs* is less than or equal to *acd*.

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

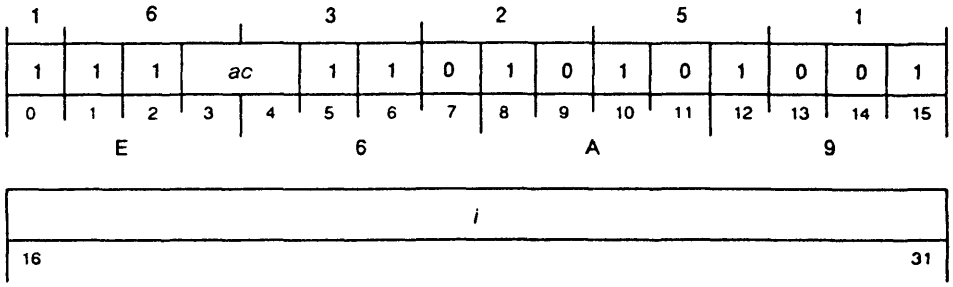
None

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 the next sequential word if *ac* is less than or equal to *i*.

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

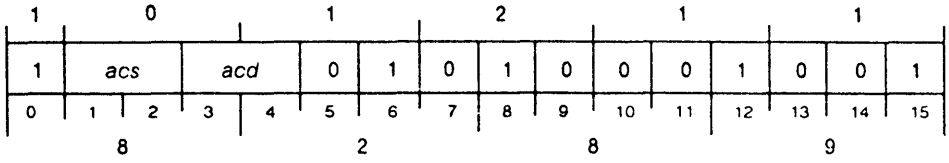
None

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 integers in *acs* and *acd* and skips the next sequential word if *acs* is less than *acd*.

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

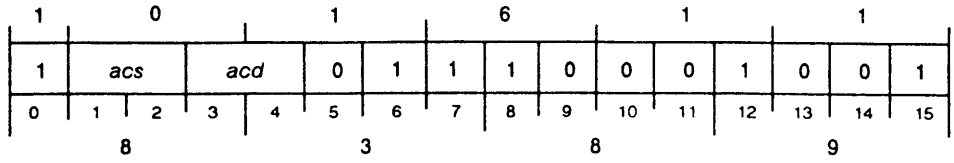
None

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 the next sequential word if the bit referred to is set to 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 0 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

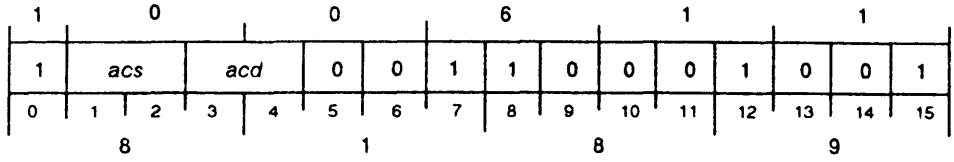
None

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 the next sequential word 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

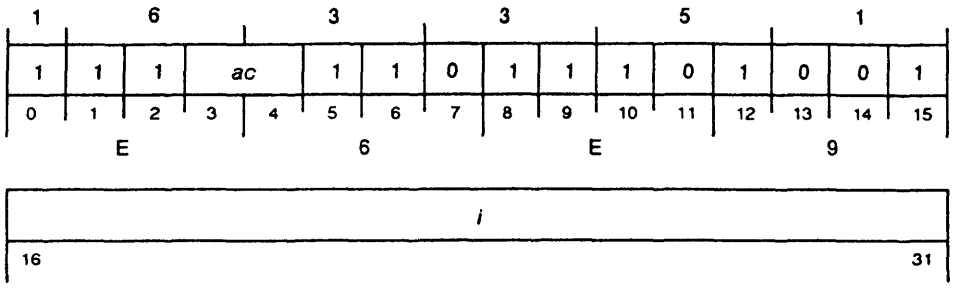
None

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 the next sequential word 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

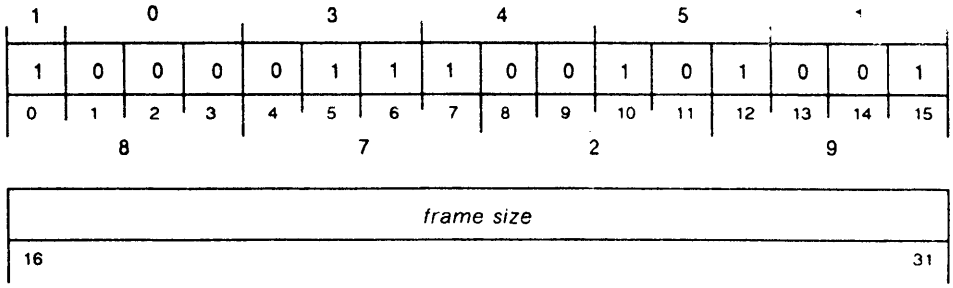
None

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 doublewords → wide stack
PSR + 0s → 1st doubleword
AC0 → 2nd doubleword
AC1 → 3rd doubleword
AC2 → 4th doubleword
wfp(previous) → 5th doubleword
CRY → 6th doubleword(bit 0)
AC3 → 6th doubleword(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 six doublewords (representing the current operating environment) onto the wide stack, resets the processor status register flags OVK and OVR to 0, and increments the wide stack pointer by the *frame size*. The return block consists of the following:

Doubleword Pushed	Contents
1	PSR (bits 0-15), zeros (bits 16-31)
2	AC0
3	AC1
4	AC2
5	Previous WFP
6	CRY (bit 0); AC3, or return PC value (bts 1-31).

After pushing the return block, the instruction places the new 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 PSR(OVK) and PSR(OVR) to 0, disabling integer overflow.

WSSVR is typically used after an **XJSR** or **LJSR** instruction, which perform an intra-ring transfer to a subroutine that requires no parameters and that uses a **WRTN** instruction to return control back to the calling sequence.

Arguments

frame size — Unsigned 16-bit integer specifying size of frame area (in doublewords) for storing data on wide stack (beyond return block).

Registers, Flags, and Stacks

AC0	Contains data for second doubleword pushed. After execution, contents unchanged.
AC1	Contains data for third doubleword pushed. After execution, contents unchanged.
AC2	Contains data for fourth doubleword pushed. After execution, contents unchanged.
AC3	Contains data for bits 1-31 of fifth doubleword pushed. After execution, contains new WFP (after push).
Carry	Unchanged
Overflow	Unaffected
PC	PC + 2
PSR	OVK and OVR bits set to 0.
Stack	Wide stack pointer incremented by six doublewords.

Related Instructions

SAVZ, WSAVR, WSAVS,WSSAVS
Push a return block on a stack.

WRTN Wide Return

Exceptions

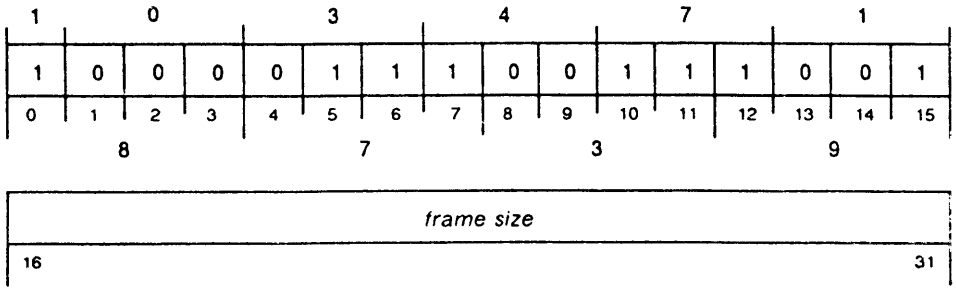
A check for stack overflow is made before the return block is pushed.

Example

```
        LJSR SUBROUT ;Subroutine call.
        . . .
SUBROUT:
        WSSVR 2      ;Save ACs, update WSP, and allocate two
        . . .        ;doublewords for local storage.
        WRTN         ;Return from the subroutine, restoring ACs.
```

Wide Special Save/Set Overflow Mask — WSSVS

WSSVS *frame size*



Function:
6 doublewords → wide stack
PSR + 0s → 1st doubleword
AC0 → 2nd doubleword
AC1 → 3rd doubleword
AC2 → 4th doubleword
wfp(previous) → 5th doubleword
CRY → 6th doubleword(bit 0)
AC3 → 6th doubleword(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 six doublewords (representing the current operating environment) onto the wide stack, sets the processor status register overflow mask (OVK) to 1 and the overflow flag (OVR) to 0, and increments the wide stack pointer by the *frame size*. The return block consists of the following:

Doubleword Pushed	Contents
1	PSR (bits 0-15), zeros (bits 16-31)
2	AC0
3	AC1
4	AC2
5	Previous WFP
6	Carry (bit 0); AC3, or return PC value (bits 1-31).

After pushing the return block, the instruction places the value of the new 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 PSR(OVK) to 1 and PSR(OVR) 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** instruction to return control back to the calling sequence.

Arguments

frame size — Unsigned 16-bit integer specifying size of frame area (in doublewords) for storing data on wide stack (beyond return block).

Registers, Flags, and Stacks

AC0	Contains data for second doubleword pushed.
	After execution, contents unchanged.
AC1	Contains data for third doubleword pushed.
	After execution, contents unchanged.
AC2	Contains data for fourth doubleword pushed.
	After execution, contents unchanged.
AC3	Contains data for bits 1–31 of fifth doubleword pushed.
	After execution, contains new WFP (after push).
Carry	Unchanged
Overflow	Unaffected
PC	PC + 2
PSR	OVK set to 1, OVR set to 0.
Stack	Wide stack pointer incremented by six doublewords.

Related Instructions

SAVZ, WSAVS,WSSAVS
Push a return block onto a stack.

WRTN Wide Return

Exceptions

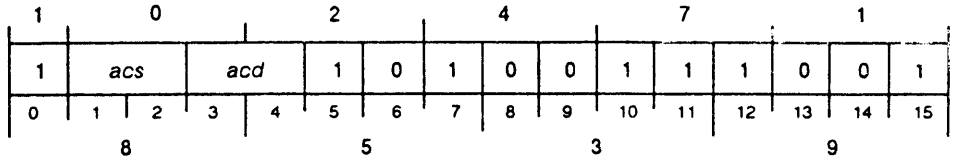
A check for stack overflow is made before the return block is pushed.

Example

```
        LJSR SUBROUT ;Subroutine call.
        . . .
SUBROUT:
        WSSVS 2      ;Save ACs, update WSP, and allocate two
        . . .        ;doublewords 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

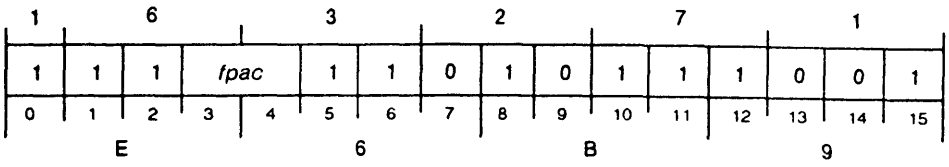
None

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 *fpac* 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 *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 decimal/ASCII fault occurs.

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.

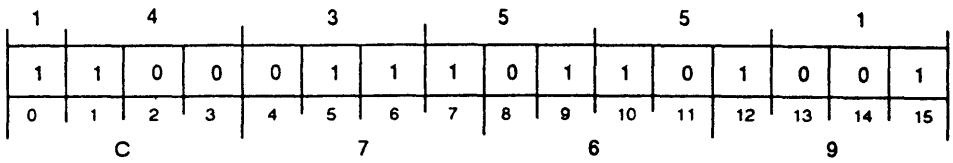
For data type 7, the low-order bytes are set to 0.

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 the type
                   ;specified by AC1, at the location
                   ;specified by AC3.
```

Wide Store Integer Extended — WSTIX

WSTIX



Function: fpac(0-3)[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	Set to 1 if integer too large to fit in destination field; otherwise set to 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	After execution, contents 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 decimal/ASCII fault occurs

If any of the floating-point accumulators contains a value greater than 10^{18} - 1, a decimal/ASCII fault occurs.

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.

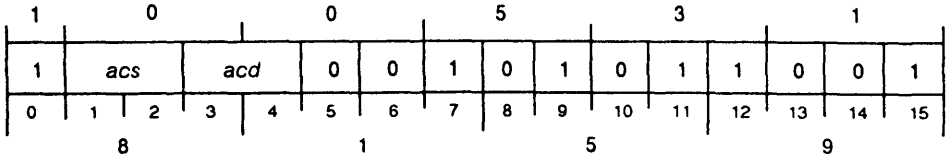
If the integer does not completely fill the destination field, the high-order bits are set to 0.

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 an 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

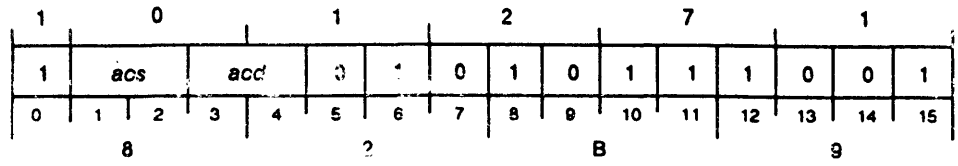
If an overflow occurs, PSR(OVR) is set to 1.

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 the next sequential word 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

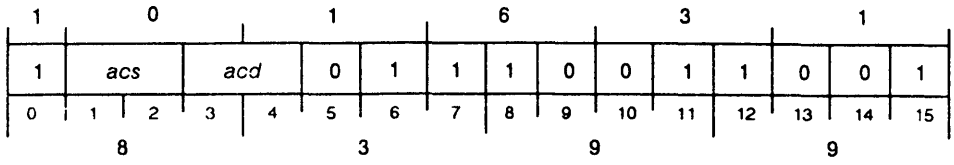
None

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

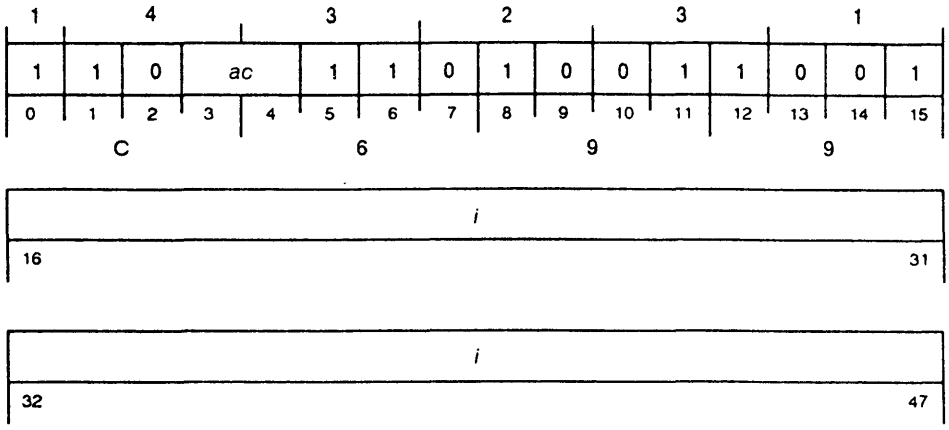
None

Example

```
;This subroutine removes an element from a linked list queue. It is
;the responsibility of the caller to set the transition bit, if
;necessary.
;
;Calling conventions:          XJSR PDEQ
;                              <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 the 32-bit immediate field and skips the next sequential word if *ac* is greater than *i*.

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

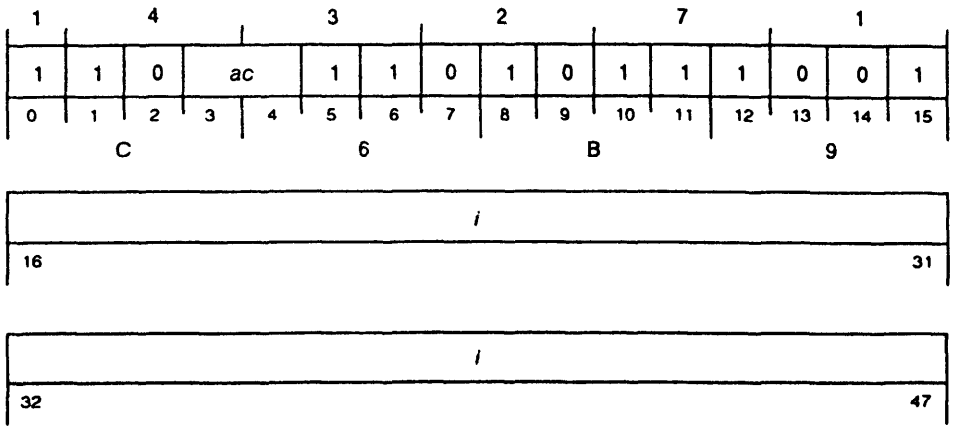
None

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 the next sequential word if *ac* is less than or equal to *i*.

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

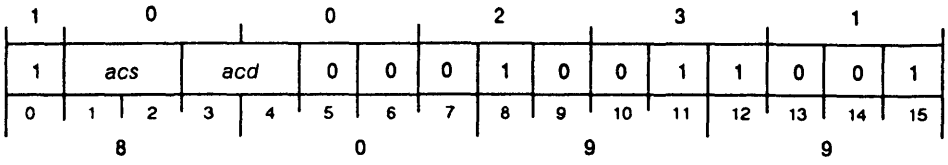
None

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. If *acd* is *acs*, *acs* is compared with 0.

WUSGE performs an unsigned comparison of the integers in *acs* and *acd*, and skips the next sequential word if the integer in *acs* is greater than or equal to the integer in *acd*.

If *acs* and *acd* are the same accumulator, then the instruction compares the integer contained in the accumulator to 0 (this will always cause a skip of the next word).

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

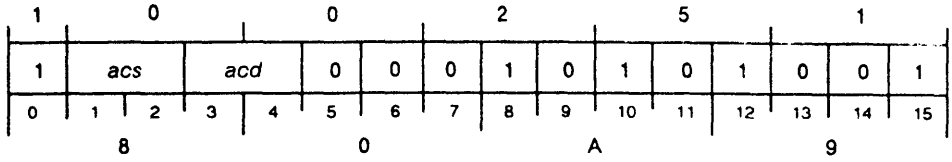
If *acs* and *acd* are the same accumulator, **WUSGE** always skips the next sequential word.

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 performs an unsigned comparison of the integers in *acs* and *acd* and skips the next sequential word if the integer in *acs* is greater than the integer in *acd*.

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

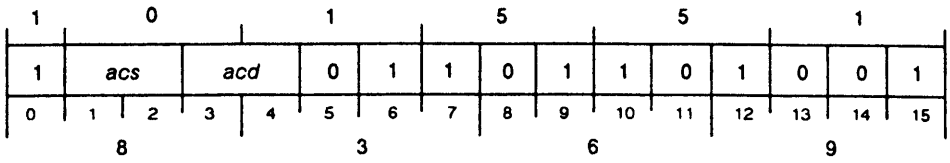
None

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

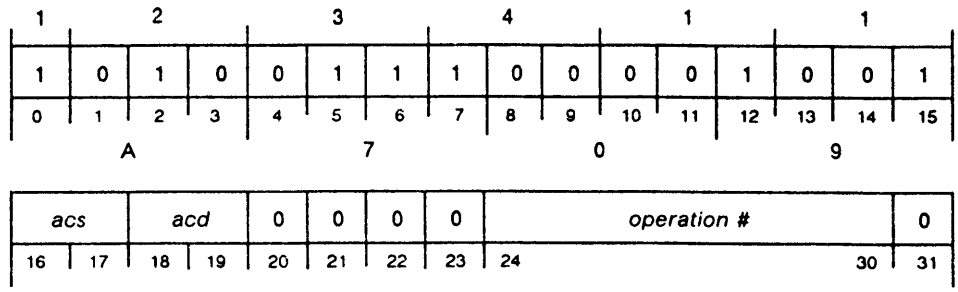
None

Example

```
WXCH  0,2          ;Exchange the contents of AC0 and AC2.
```

Wide Extended Operation — WXOP

WXOP *acs,acd,operation #*



Function: 6 doublewords → 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 of six doublewords onto the wide stack and transfers control to a procedure pointed to by the selected address in an extended operations table (**WXOP** table). This is an efficient way to transfer control from one procedure to another. The return block consists of the following:

Doubleword Pushed	Contents
1	PSR (bits 0-15), zeros (bits 16-31)
2	AC0
3	AC1
4	AC2
5	AC3
6	Carry (bit 0), WXOP address + 2 (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 (*operation #*) as an offset from the starting address of the **WXOP** table. The **WXOP** table is an array of doublewords, each of which contains an address; the instruction treats these addresses as intermediate addresses. 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). The table's starting address is stored in page zero locations 12_8 and 13_8 of reserved memory for the current segment. All addresses must refer to locations 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	Contains data for second doubleword pushed.
	After execution, contents unchanged.
AC1	Contains data for third doubleword pushed.
	After execution, contents unchanged.
AC2	Contains data for fourth doubleword pushed.
	After execution, contains stack address of pushed contents of *acs*.
AC3	Contains data for fifth doubleword 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 doublewords.

Related Instructions

WPOPB	Use the Wide Pop Block instruction to restore pushed values and return.
XOP0	Extended Operation

Exceptions

If a stack overflow occurs, a wide stack fault occurs.

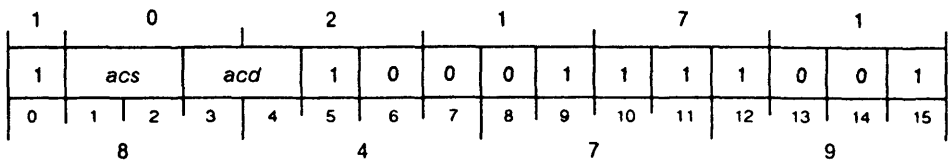
Example

```
    .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

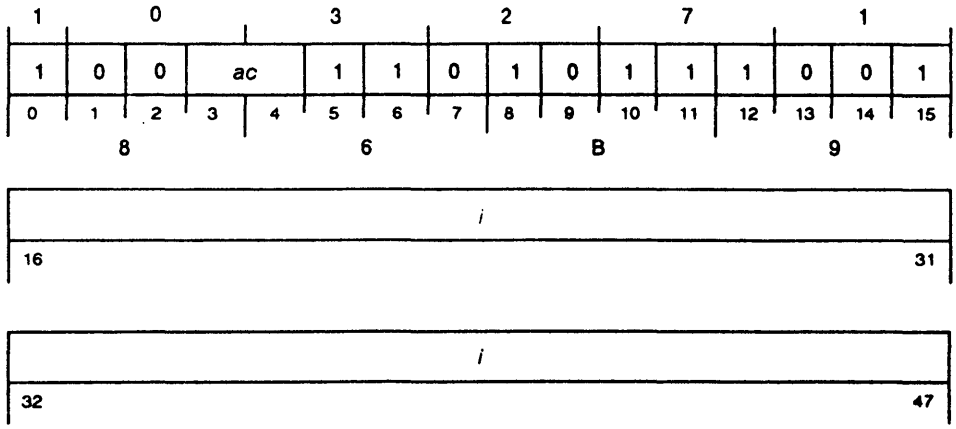
None

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 between corresponding bits of *ac* and the value contained in the immediate field, placing the result in *ac*.

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

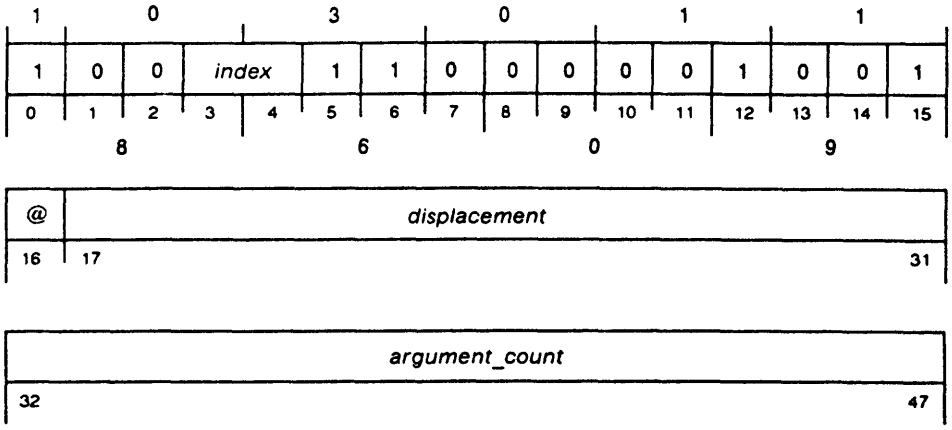
None

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 0 and loads the program counter with the target address.

For additional information on **XCALL**, refer to the section, "Transferring Program Control to Another Segment," in the chapter, "Program Flow Management."

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 AC1 contains error code 7. 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, AC1 contains error code 6, and no subroutine call made. PC in return block undefined.

If gate is 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* doubleword depending on value of high bit (bit 48) of *argument_count*.

If high bit is 0, **XCALL** pushes onto the wide stack a doubleword 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 doubleword 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 doubleword and ignores *argument_count* coded with **XCALL** instruction. The instruction then places current PSR into bits 0-15 of stack doubleword.

(If target address is in inner segment, **XCALL** copies the number of doublewords specified in *argument_count* from the outer segment stack to the inner segment stack, and then pushes the PSR/*argument_count* doubleword onto inner stack.) Note that, in this case, the instruction does not push the PSR/*argument_count* onto the outer segment's stack — if the *argument_count* is already on the outer segment stack, the instruction pops this value before crossing to the lower segment.)

Registers, Flags, and Stacks

AC0-AC2	Unused
AC3	After execution, contains PC + 3 (always refers to 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* doubleword.

If target address is inner segment, inner segment wide stack contains PSR/*argument_count* doubleword and copy of arguments.

Related Instructions

WSAVR, WSAVS

Push five doublewords onto the wide stack. Generally, should be the first instruction in the subroutine.

WRTN

Pops six doublewords from wide stack. Generally, should be the last instruction in the subroutine.

When returning to an outer segment, the Wide Return instruction pops the return block, loads the PSR, and removes the number of arguments, specified by the **XCALL** *argument_count*, from both the inner segment stack and the outer segment stack.

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* doubleword, 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	Program Counter Contents
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

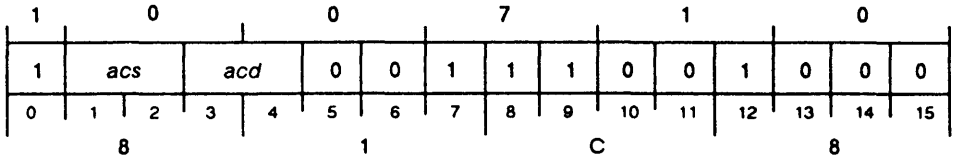
Example

```
    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 six 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

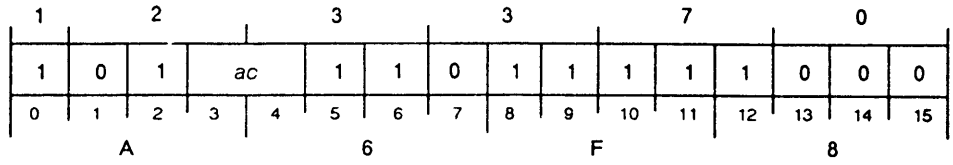
Example

```
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 contents of *ac* 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.

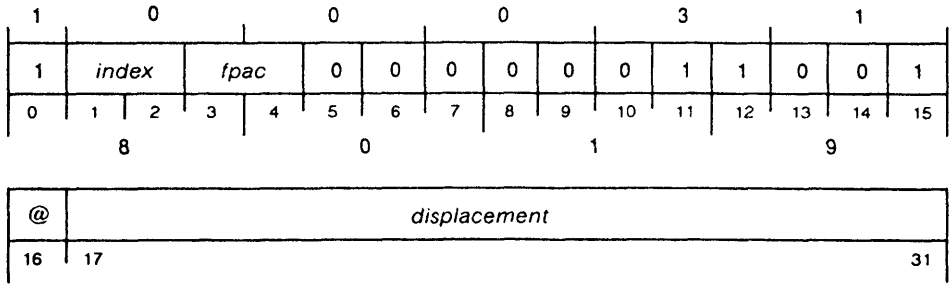
If the instruction in *ac* is an Execute instruction that specifies the same *ac*, the processor is placed in a one-instruction loop. Because of this possiblility, this instruction is interruptible. An I/O interrupt can occur immediately prior to each time the instruction in *ac* is executed. If an I/O 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 creates 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 double-precision floating-point number in memory to the double-precision floating-point number in *fpac* and places the normalized result into *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

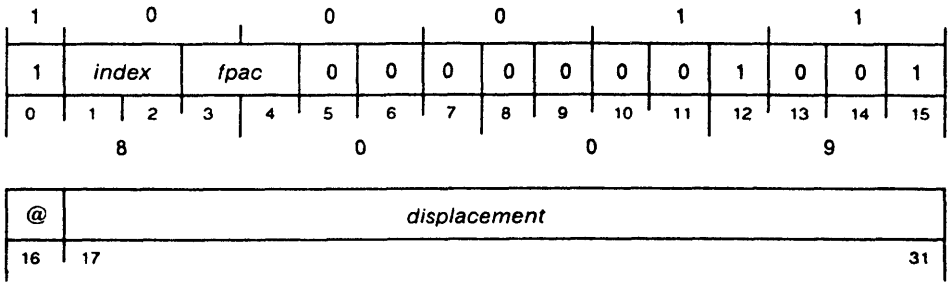
If addition produces an exponent overflow, the processor sets FPSR(OVF) to 1 and terminates the instruction.

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 single-precision floating-point number in memory to the single-precision floating-point number in *fpac* and places the normalized result into *fpac*.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contains normalized 32-bit result (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

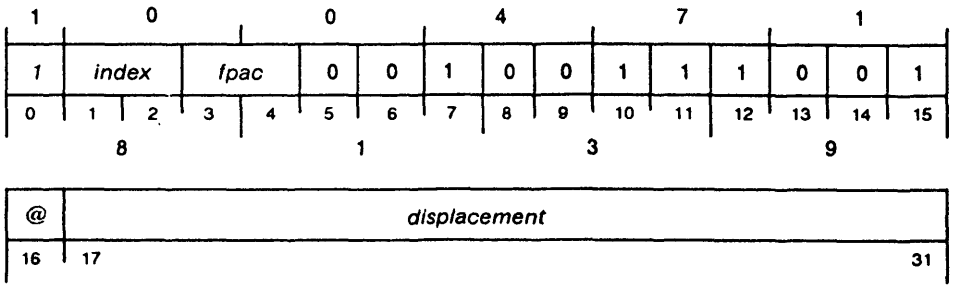
If the addition produces an exponent overflow, the processor sets FPSR(OVF) to 1 and terminates the instruction.

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 double-precision floating-point number in *fpac* by a double-precision floating-point number in memory and places the normalized result into *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

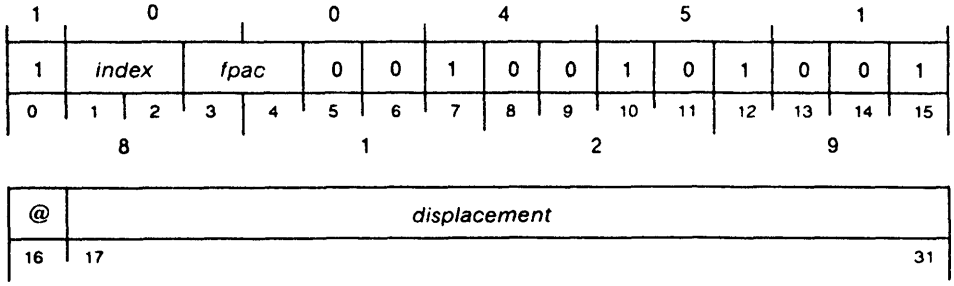
If the divisor (in memory) is 0, the processor sets FPSR(INV) to 1, places error code 0 in FPSR(INP) and the address of the instruction in FPSR(FPPC), and terminates the instruction.

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 single-precision floating-point number in *fpac* by a single-precision floating-point number in memory and places the normalized result into *fpac*.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contains normalized 32-bit result (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

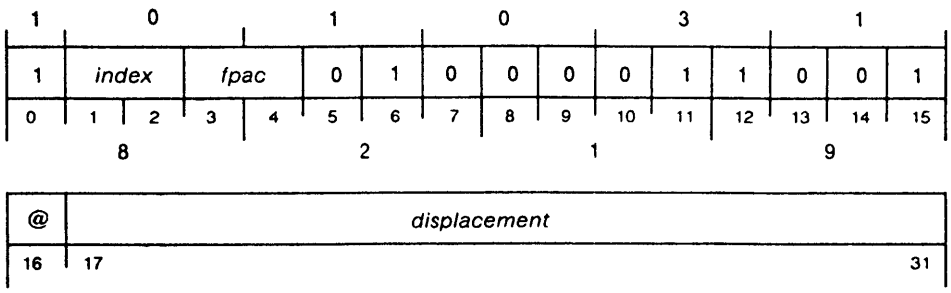
If the divisor (in memory) is 0, the processor sets FPSR(INV) to 1, places error code 0 in FPSR(INP) and the address of the instruction in FPSR(FPPC), and terminates the instruction.

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.
```

Load Floating-Point Double (Extended Displacement) XFLDD

XFLDD *fpac*,[@]*displacement*[,*index*]



Function: (E) → *fpac*

Parameters: None

XFLDD loads a 64-bit floating-point value from memory into the *fpac*. Unnormalized data is moved without change.

Arguments

fpac After execution, contains 64-bit floating-point value.

[@]*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

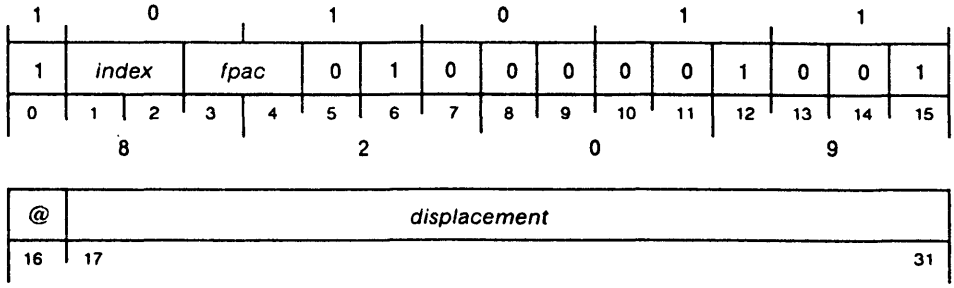
None

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 value from memory into the *fpac*. Unnormalized data is moved without change.

Arguments

fpac(0-31) After execution, contains 32-bit floating-point value (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

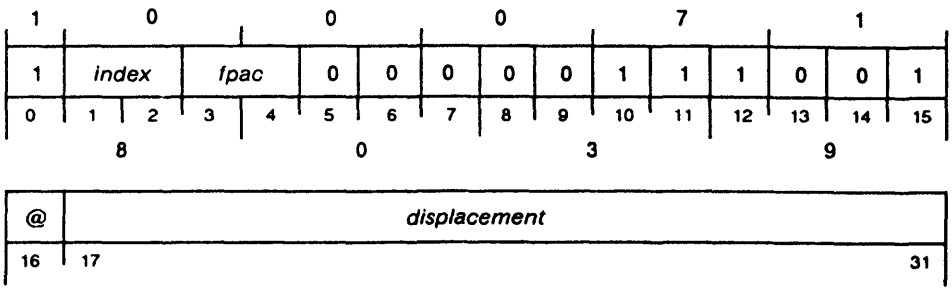
None

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 double-precision floating-point number contained in memory by the double-precision floating-point number in *fpac* and places the normalized result into *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

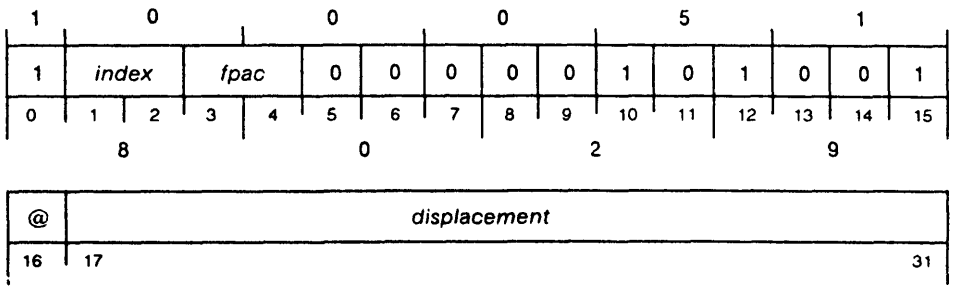
If multiplication produces an exponent overflow, the processor sets FPSR(OVF) to 1, and terminates the instruction.

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 single-precision floating-point number in memory by the single-precision floating-point number in *fpac* and places the normalized result into *fpac*.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contains normalized 32-bit result (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

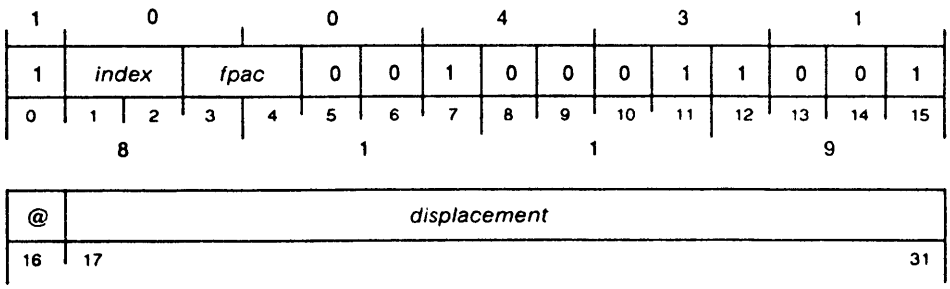
If multiplication produces an exponent overflow, the processor sets FPSR(OVF) to 1, and terminates the instruction.

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 double-precision floating-point number in memory from the double-precision floating-point number in *fpac*, and places the normalized result into *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

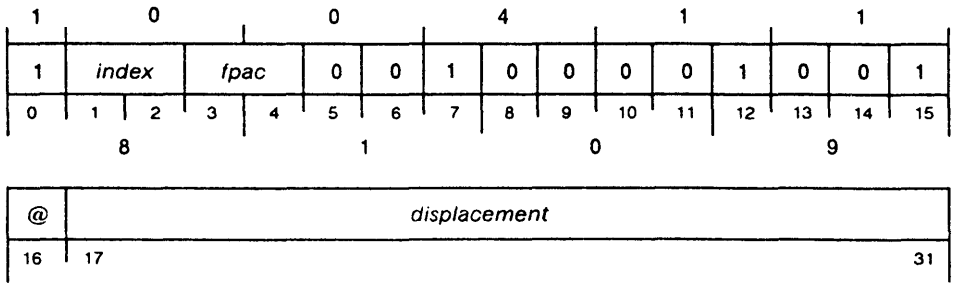
If subtaction produces an exponent overflow or underflow, the processor sets the appropriate FPSR flag to 1, and terminates the instruction.

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 single-precision floating-point number in memory from the single-precision floating-point number in *fpac*, and places the normalized result into *fpac*.

Arguments

fpac(0-31) Before execution, contains 32-bit floating-point number.

After execution, contains normalized 32-bit result (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

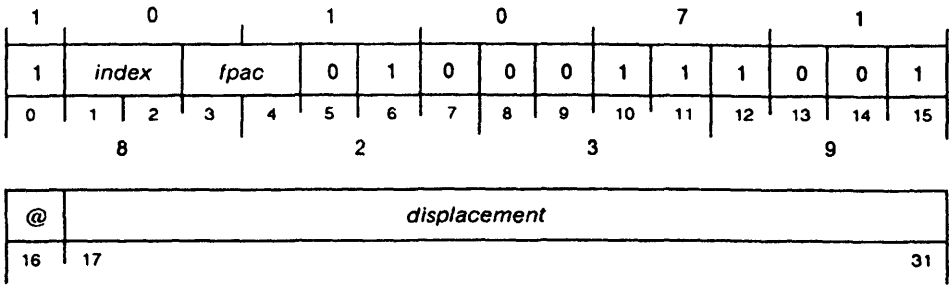
If subtraction produces an exponent overflow or underflow, the processor sets the appropriate FPSR flag to 1, and terminates the instruction.

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 *fpac* 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

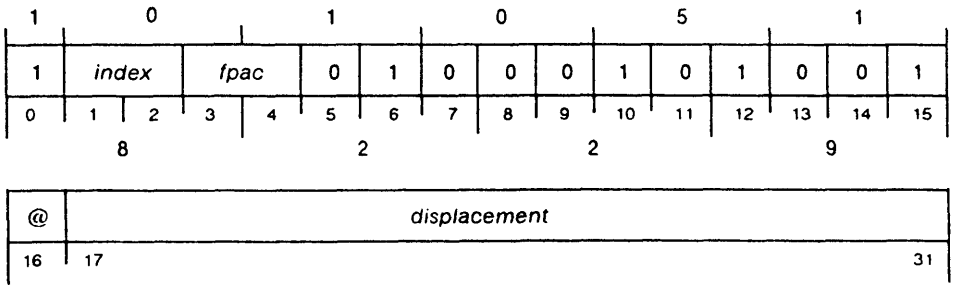
None

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 *fpac* 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

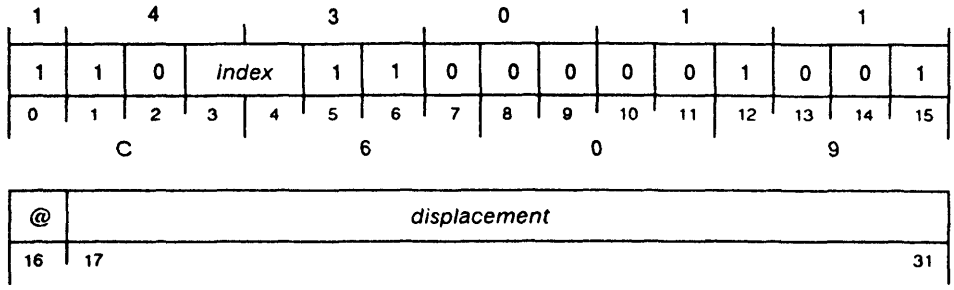
None

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

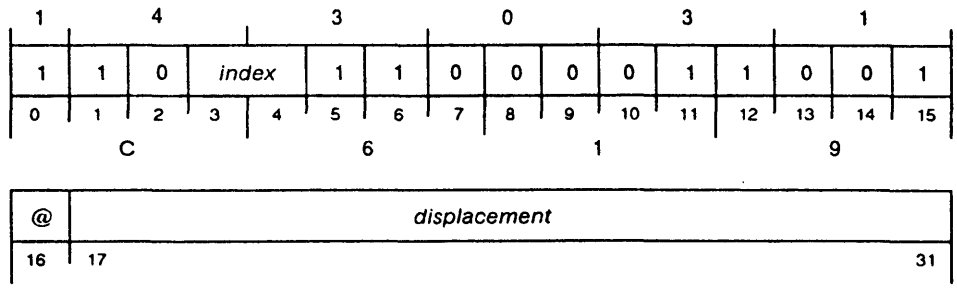
None

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 places the effective address into the program counter (transferring 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

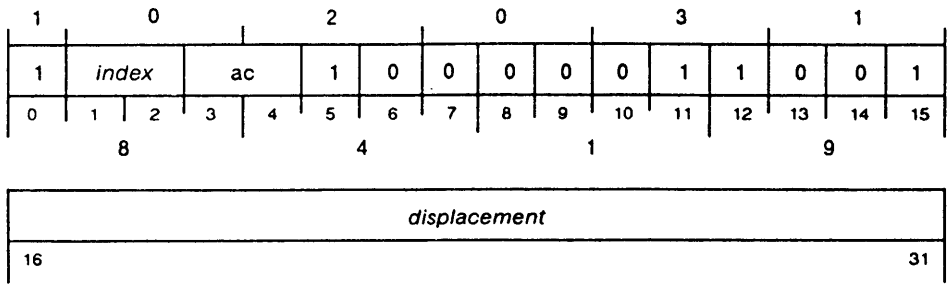
None

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 refer to a byte in memory. The instruction then loads the addressed byte into *ac* and zero-extends the value to 32 bits.

Arguments

ac(24-31) After execution, contains byte from memory (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 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

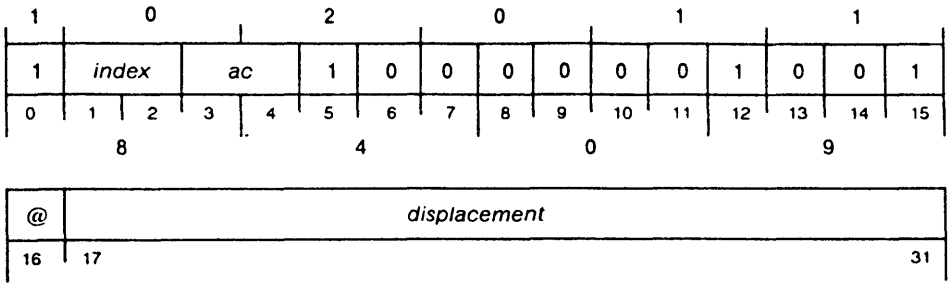
None

Example

```
    XLDB   2,(BYTE_PAIR*2)+1        ;Load AC2 with the low-order byte
    .                               ;from the word, BYTE_PAIR.
    .
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 *ac*. Bit 0 of the result is guaranteed to be 0.

Arguments

ac After execution, contains result (bit 0 is 0).

[@]*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

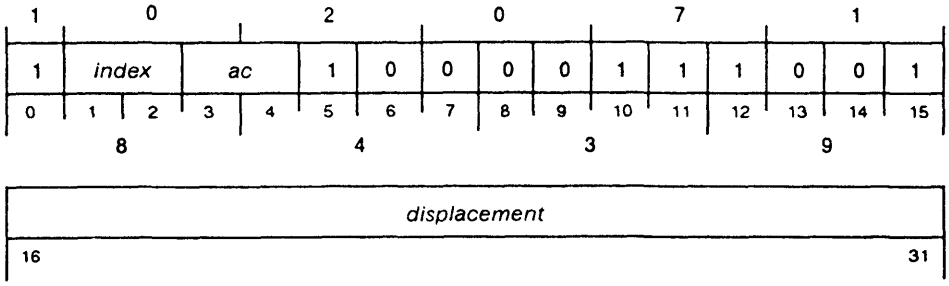
None

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 *ac*.

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

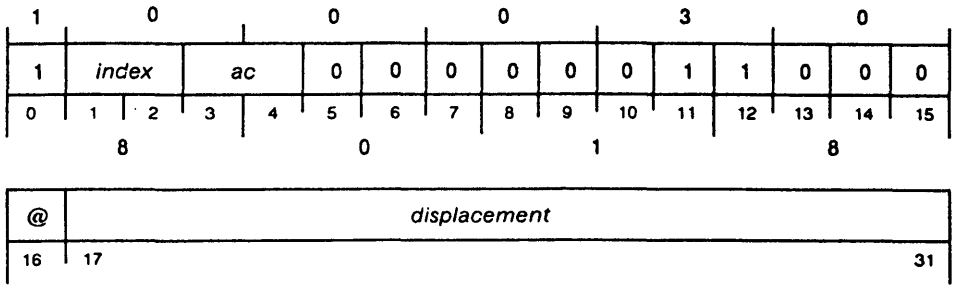
Example

```
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 carry → CRY

Parameters: None

XNADD adds the signed 16-bit integer in memory to the signed 16-bit integer in *ac*. The instruction then sign-extends the 16-bit result to 32 bits and loads it into *ac*.

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

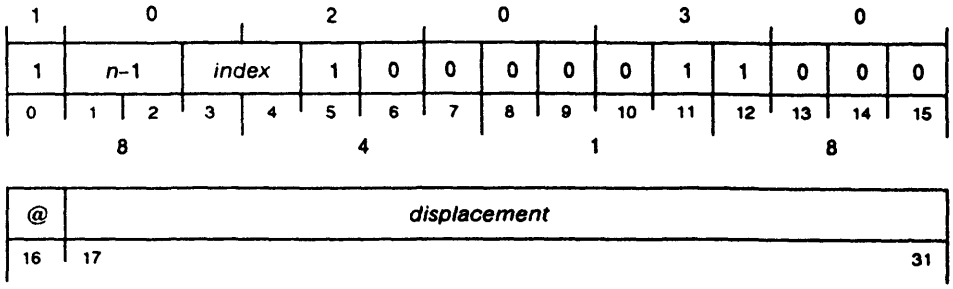
If the result of the add produces a value greater than 32,767, PSR(OVR) is set to 1.

Example

```
XNLDA 0,FIRST      ;Get one value (only 16 bits).
XNADD 0,SECOND     ;Add the second value (16-bit arithmetic).
XWSTA 0,RESULT     ;Store the doubleword result.
```

Narrow Add Immediate (Extended Displacement) XNADI

XNADI *n*,[@]*displacement*[,*index*]



Function: *n* + (E) → (E)
ALU carry → 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.

Since Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field, you 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

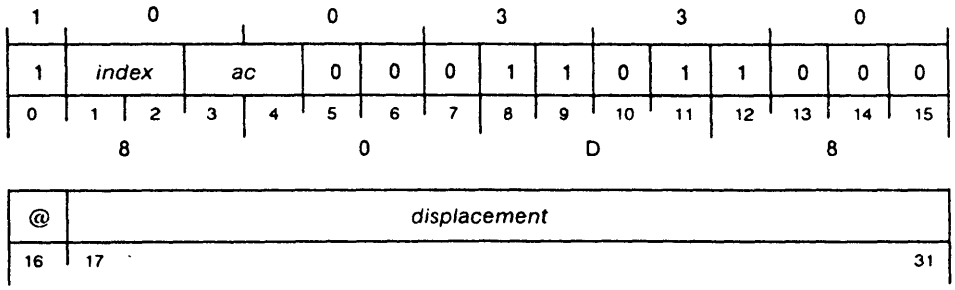
If the add produces a result greater than 32,767, PSR(OVR) is set to 1.

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; PSR(OVR) = 1.

XNDIV sign-extends the signed 16-bit integer in *ac* to 32 bits and divides it by the signed 16-bit integer in memory. It then sign-extends the result to 32 bits and loads it into *ac*.

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 0; 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

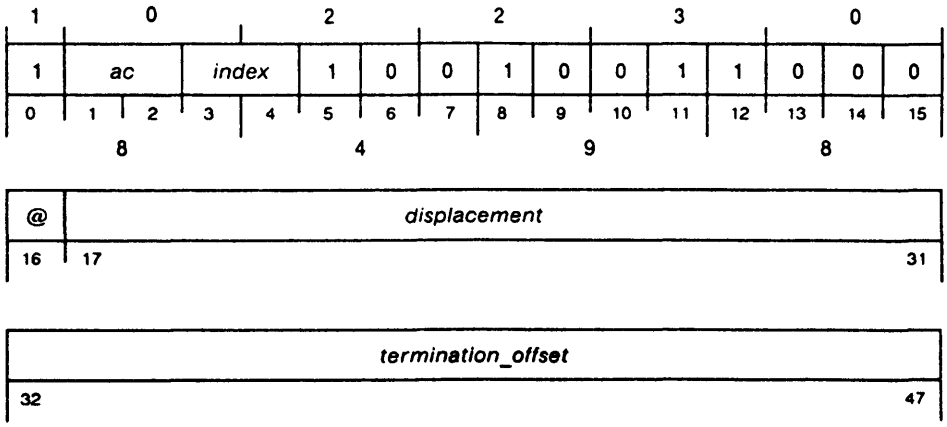
If the quotient is outside the range, -32,768 to +32,767 inclusive, or if the memory location contains 0, an overflow occurs, and PSR(OVR) is set to 1.

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 carry → 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 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.
- 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 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 **XNDO**.

termination_offset
: Specifies signed PC-relative address for normal return. Argument ranges from 0 to 64 Kwords. (This value is sign-extended to 32 bits for the addition. The final value contains the current segment of execution in bits 1-3.)

[@]*displacement*[,*index*]
: Specifies effective address of a memory word 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 + 3 (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 a value one greater than actual loop count.

WBR
: Use the Wide Banch instruction to end the DO-loop (to loop back to the **XNDO** instruction).

Exceptions

In any return block, the contents of the specified memory location and the program counter value are both undefined.

If a fixed-point overflow fault occurs while incrementing the DO-loop variable, the contents of the memory location and the PC value in the return block are undefined. (AC0 will contain the address of the DO-loop instruction.)

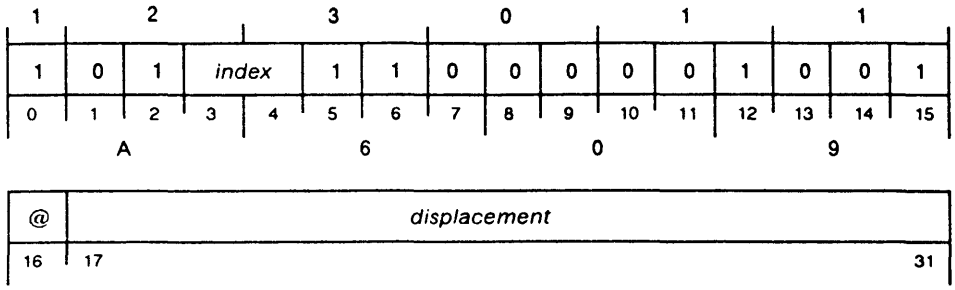
Example

```
    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 memory 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

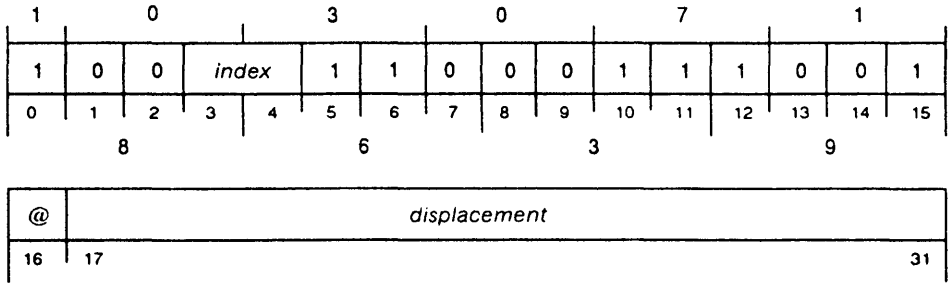
None

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 0.
    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 memory 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 = 0)
PSR	Unchanged
Stack	Unchanged

Related Instructions

ISZ, EISZ, XWISZ, LNISZ, LWISZ
Increment contents of memory and skip if result is zero.

Exceptions

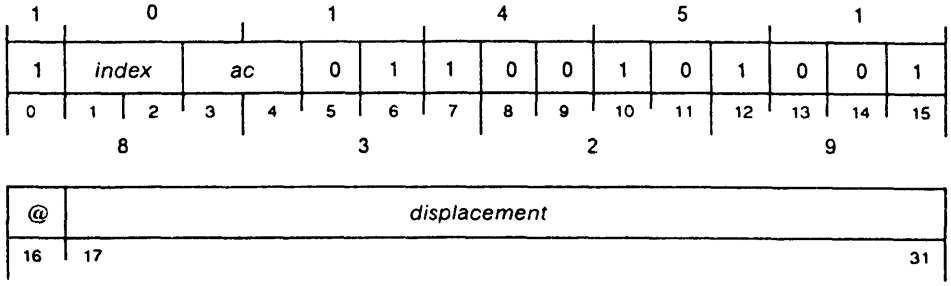
None

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 0.
          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 copies the signed 16-bit integer in memory, sign-extends this integer to 32 bits, and loads it into *ac*.

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 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

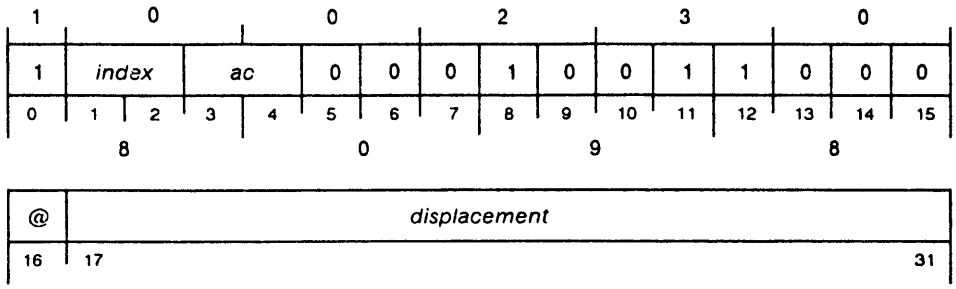
Example

```
XNLDA 0,SINGLE_WORD          ;Get 16 bit value and sign-extend.
XWSTA 0,DOUBLE_WORD          ;Store the value as a doubleword.
```

Narrow Multiply Memory Word — XNMUL

(Extended Displacement)

XNMUL *ac*,[@]*displacement*[,*index*]



Function: (E) * *ac* → *ac*

Parameters: None

XNMUL multiplies the signed 16-bit integer in memory by the signed 16-bit integer in *ac*. It then sign-extends the result to 32 bits and places the result in *ac*.

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

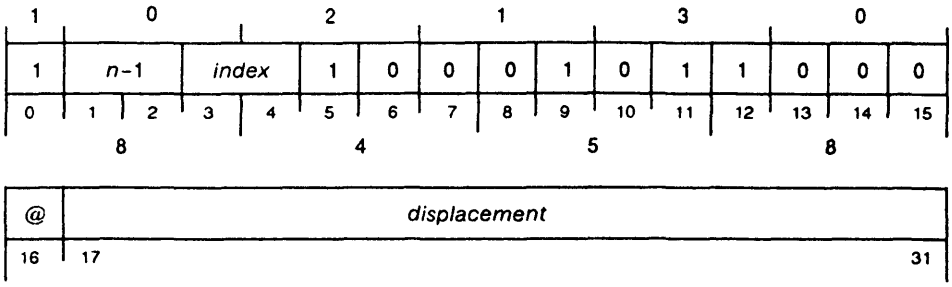
If the result is outside the range, -32,768 to +32,767 inclusive, an overflow occurs, and PSR(OVR) is set to 1.

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 carry → CRY

Parameters: None

XNSBI subtracts an integer in the range of 1 to 4 from the signed 16-bit integer in the specified memory location.

Arguments

n — Integer in range 1 to 4.

Since Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field, you 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

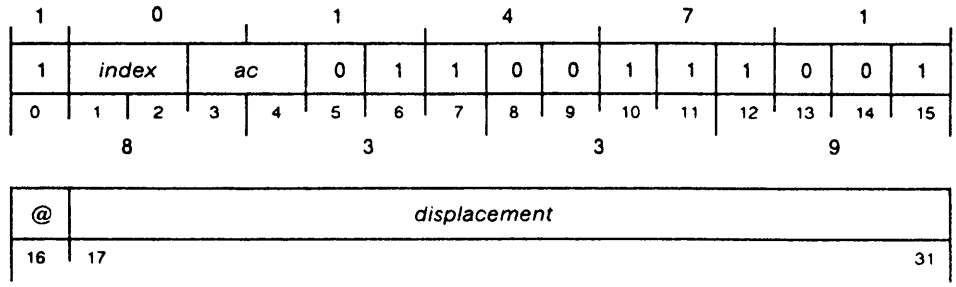
None

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 *ac* 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

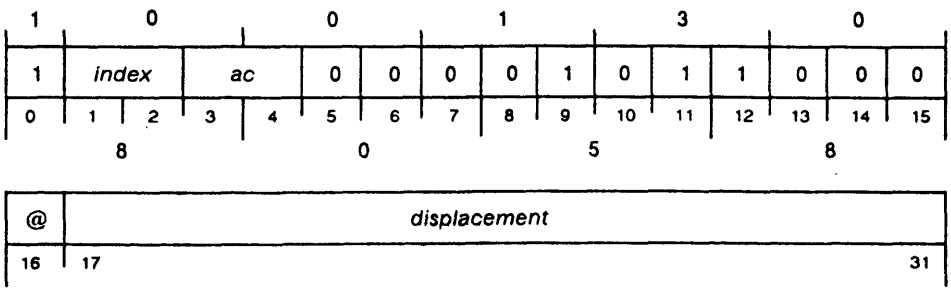
None

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 carry → CRY

Parameters: None

XNSUB subtracts a signed 16-bit integer in memory from the signed 16-bit integer in *ac*. Then it sign-extends the result to 32 bits and stores it in *ac*.

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

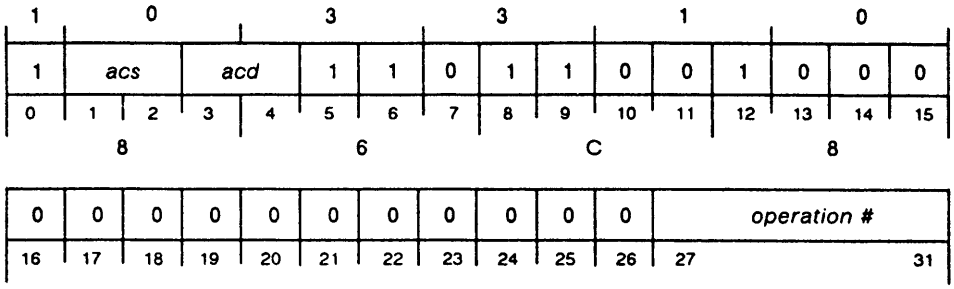
None

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 of five words onto the narrow stack and transfers control to a procedure pointed to by the selected address in an extended operations table (XOP table). This is an efficient way to transfer control from one procedure to another. The return block consists of the following:

Word Pushed	Contents
1	AC0
2	AC1
3	AC2
4	AC3
5	Carry (bit 0), **XOP0** address + 2 (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 (*operation #*) as an offset from the starting address of the XOP table. The XOP table is an array of words, each of which contains an address; the instruction treats these addresses as intermediate addresses. 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). The table's starting address is stored in page zero location 44_8 of reserved memory for the current segment. All addresses must refer to locations in the current segment.

Arguments

acs	Specifies accumulator whose address on narrow stack is stored in AC2.
acd	Specifies accumulator whose address on narrow stack is stored in AC3.
operation #	Unsigned 5-bit integer specifying offset from XOP table starting address.

Registers, Flags, and Stacks

AC0	Contains data for first word pushed.
	After execution, contents unchanged.
AC1	Contains data for second word pushed.
	After execution, contents unchanged.
AC2	Contains data for third word pushed.
	After execution, contains stack address of pushed contents of *acs*.
AC3	Contains data for 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

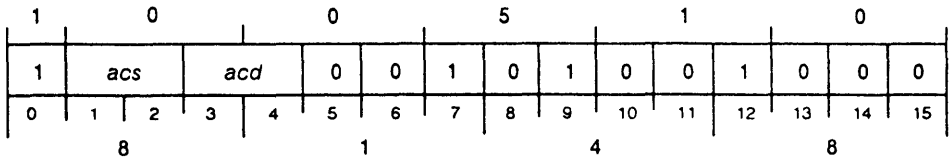
Example

```
XOP0
```

Exclusive OR — XOR

ECLIPSE Instruction

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*. The instruction 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

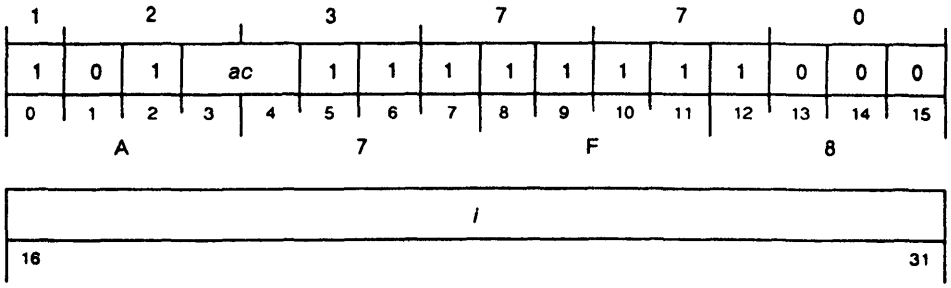
Example

```
XOR   0,1          ;Exclusively OR AC0[16-31] and AC1[16-31].
```

Exclusive OR Immediate XORI

ECLIPSE Instruction

XORI *i,ac*



Function: *i* XOR *ac* → *ac*

Parameters: None

XORI forms the logical exclusive OR of the contents of the 16-bit immediate field and the 16-bit value in *ac*, placing the result in *ac*.

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

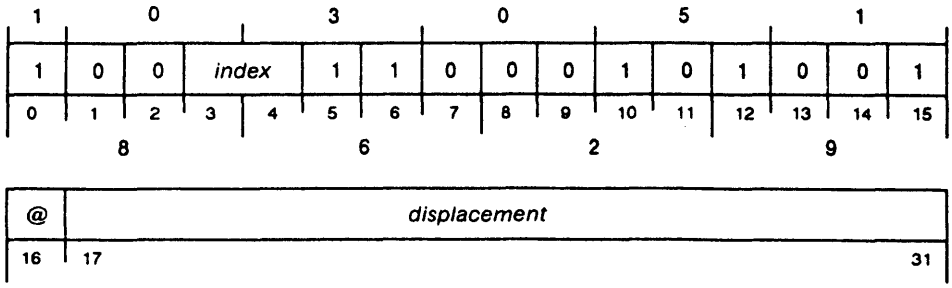
None

Example

```
XORI   377,0        ;Form the one'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 word address and pushes it onto the wide stack. The instruction then 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 doubleword of wide stack contains effective word address with bit 0 set to 0.

Related Instructions

XPEFB Push Byte Address (Extended Displacement)

Exceptions

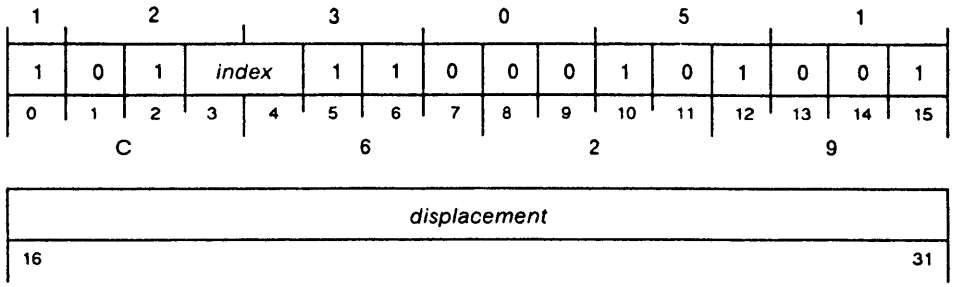
None

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 three 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. The instruction then 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 doubleword of wide stack contains 32-bit byte address.

Related Instructions

XPEF	Push Address (Extended Displacement)

Exceptions

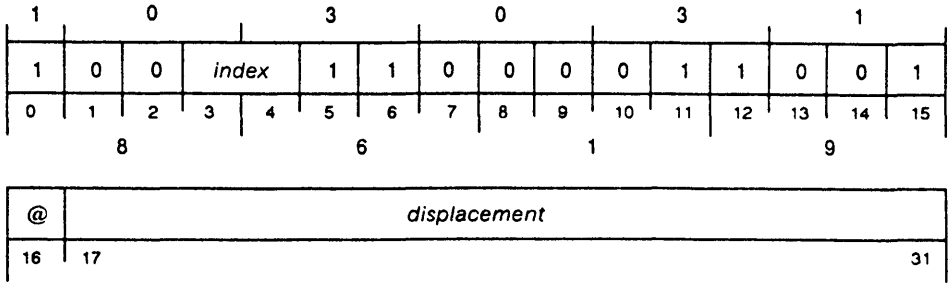
None

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 two 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 program counter plus 2 onto the wide 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. The pushed address always refers to 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

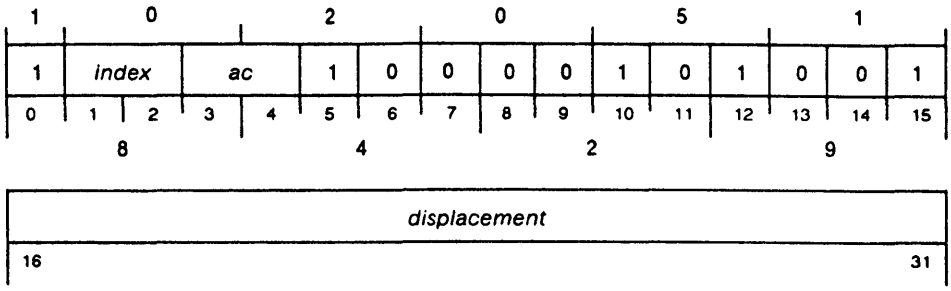
None

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 moves a copy of the contents of bits 24–31 of *ac* 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

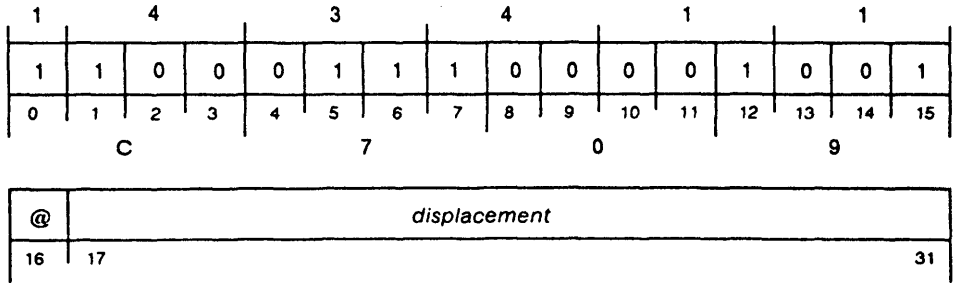
Example

```
    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 # = doubleword offset to vector table entry
Vector table entry(bits 1-31) = E(DCT entry 0)
Wide stack registers = ? → Vector stack
Wide stack fault handler address = ? → Vector stack handler address
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 doubleword 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 program counter.

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

Arguments

[@]*displacement* Effective address (E) refers to entry 0 of vector table in segment 0. Indirection chain, if any, is narrow.

Interrupting device number becomes doubleword offset pointing to appropriate entry in vector table.

Vector table entry bits 1-31 contains address of entry 0 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 0, address of DCT.
AC3	Unused
PSR	Initialized by processor to contain word 4 of DCT.
Overflow	Unaffected
PC	Initialized by processor to contain address of device interrupt routine from bits 4-31 of first doubleword in 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

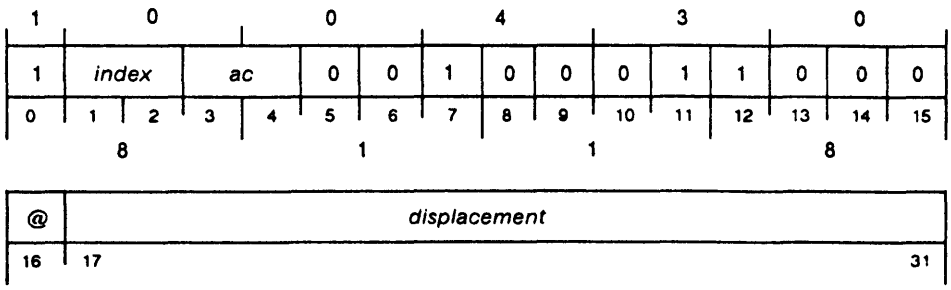
Example

```
XVCT
```

Wide Add Memory Word to Accumulator **XWADD**

(Extended Displacement)

XWADD *ac*,[@]*displacement*[,*index*]



Function: (E) + *ac* → *ac*
ALU carry → CRY

Parameters: None

XWADD adds a signed 32-bit integer in memory to the signed 32-bit integer in *ac*, placing the result in *ac*.

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

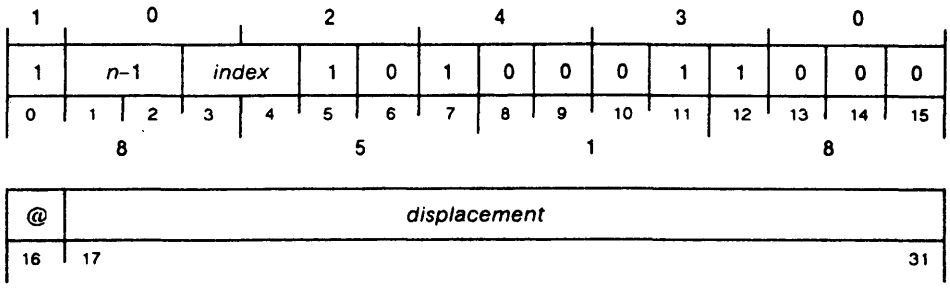
If the result of the add produces a result outside the range, -2,147,483,648 to +2,147,483,647, PSR(OVR) is set to 1.

Example

```
XWLDA 0,FIRST      ;Get one value (32 bits).
XWADD 0,SECOND     ;Add the second value (32-bit arithmetic).
XWSTA 0,RESULT     ;Store the doubleword result.
```

Wide Add Immediate (Extended Displacement) XWADI

XWADI *n*,[@]*displacement*[,*index*]



Function: *n* + (E) → (E)
ALU carry → CRY

Parameters: None

XWADI adds an integer in the range of 1 to 4 to the signed 32-bit integer in memory.

Arguments

n Integer in range 1 to 4.

Since Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field, you 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

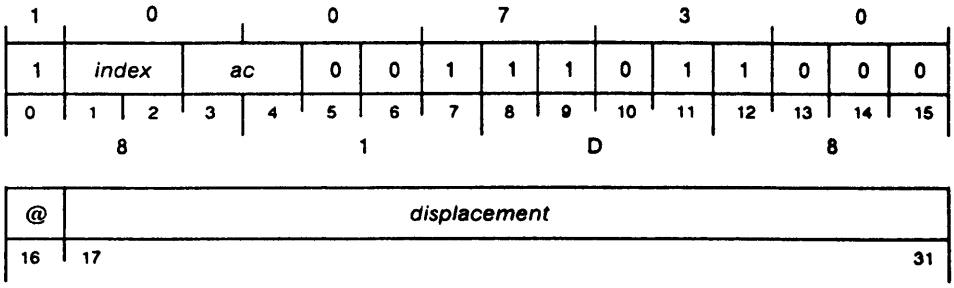
None

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; PSR(OVR) = 1, *ac* = unchanged.

XWDIV sign-extends the signed 32-bit integer in *ac* to 64 bits and divides it by the signed 32-bit integer in memory, placing the quotient into *ac*.

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 memory contains 0, an overflow occurs, PSR(OVR) is set to 1, and *ac* is unchanged.

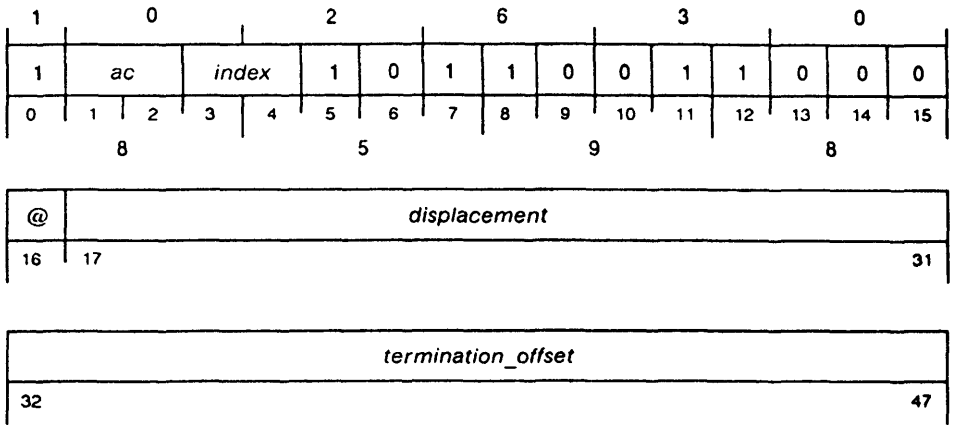
Example

```
XWLDA 0,DIVIDEND   ;Get the dividend (32 bits wide).
XWDIV 0,DIVISOR    ;Divide by the divisor.
XWSTA 0,RESULT     ;Store the doubleword 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 carry → 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 contents of memory 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.
- 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 (i.e., between **XWDO** 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 XWDO.

termination_offset
: Specifies signed PC-relative address for normal return. Argument ranges from 0 to 64 Kwords. (This value is sign-extended to 32 bits for the addition. The final value contains the current segment of execution in bits 1-3.)

[@]*displacement*[,*index*]
: Specifies effective address of a doubleword in memory to be incremented during each pass of DO-loop. Memory doubleword 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 + 3 (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 a value one greater than actual loop count.

WBR
: Use the Wide Branch instruction to end the DO-loop (loop back to the **XWDO** instruction).

Exceptions

In any return block, the contents of the specified memory location and the program counter value are both undefined.

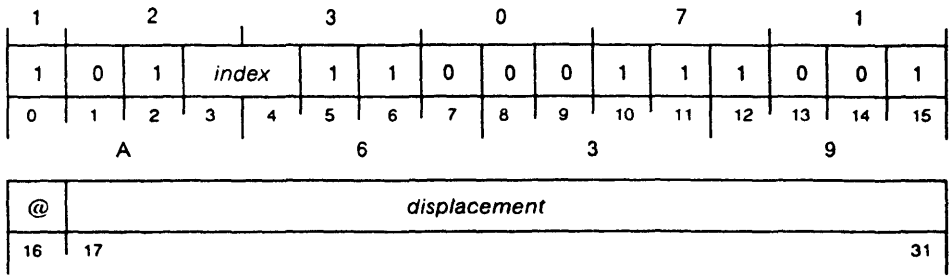
If a fixed-point overflow fault occurs while incrementing the DO-loop variable, the contents of memory and the PC value in the return block are undefined. (AC0 will contain the address of the DO-loop instruction.)

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 five 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 decrements by 1 the unsigned 32-bit integer in memory and skips the next sequential word if the result is 0. **XWDSZ** executes in one indivisible memory cycle if the value to be decremented is located on a doubleword 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.

XWSBI
Use the Wide Subtract Immediate instruction to decrement a pointer in memory in multiple-processor configurations (possible performance improvement). **XWSBI** is a non-atomic instruction.

Exceptions

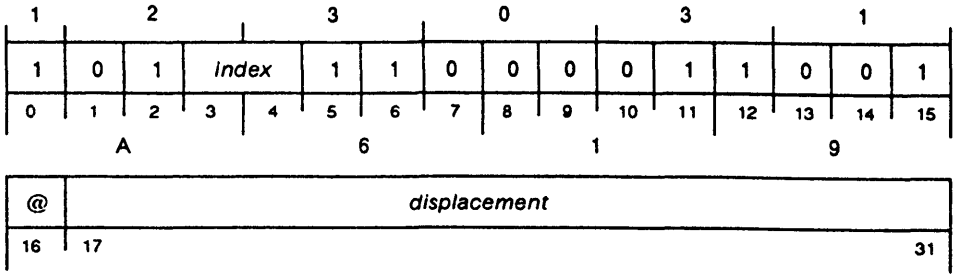
None

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 0.
           WBR LOOP     ;We're not done yet.
           . . .        ;We did the loop five 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 increments by 1 the unsigned 32-bit integer in memory and skips the next sequential word if the result is 0. **XWISZ** executes in one indivisible memory cycle if the value to be incremented is located on a doubleword 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.

XWADI
Use the Wide Add Immediate instruction to increment a pointer in memory in multiple-processor configurations (possible performance improvement). **XWADI** is a non-atomic instruction.

Exceptions

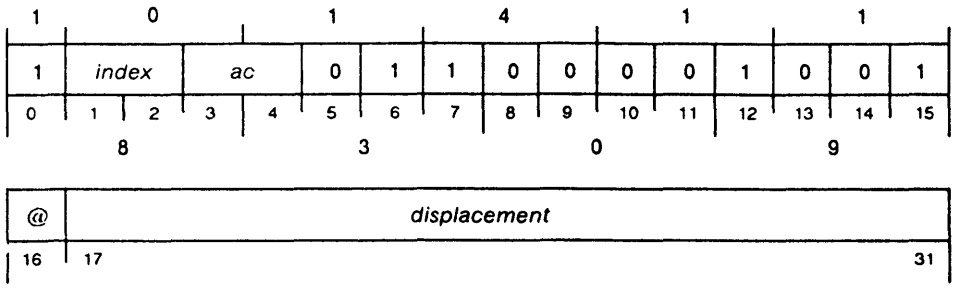
None

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 five times.
COUNTER:  .DWORD 0       ;Counter variable.
```

Wide Load Accumulator (Extended Displacement) XWLDA

XWLDA *ac*,[@]*displacement*[,*index*]



Function: (E) → *ac*

Parameters: None

XWLDA loads a copy of the 32-bit value in memory into *ac*.

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

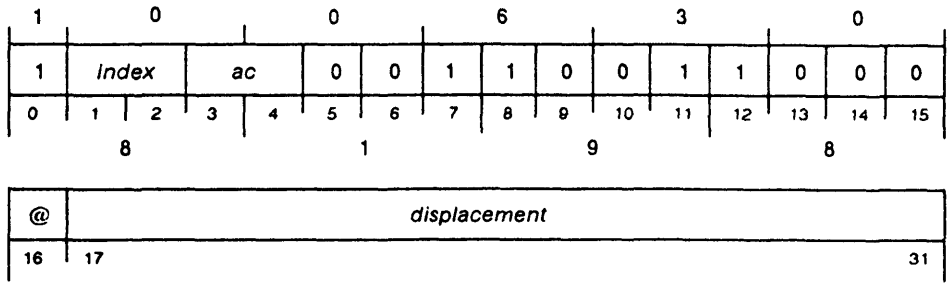
None

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 in memory by the signed 32-bit integer in *ac*. The instruction then loads the least significant 32 bits of the result into *ac*.

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

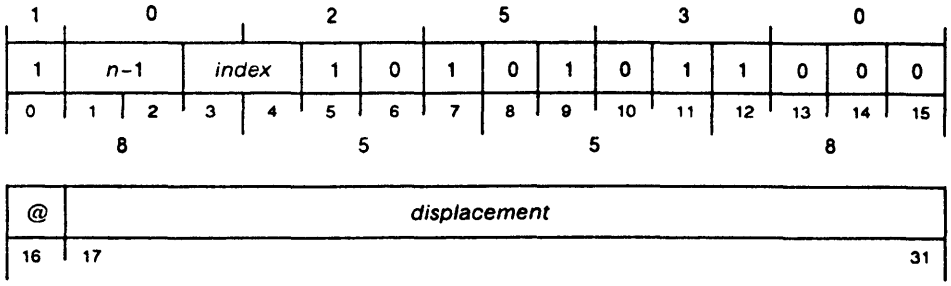
If the result is outside the range, -2,147,483,648 to +2,147,483,647 inclusive, an overflow occurs, and PSR(OVK) is set to 1.

Example

```
XWLDA 0,FIRST      ;Get one value (32 bits).
XWMUL 0,SECOND     ;Multiply by the second value (32-bit arithmetic).
XWSTA 0,RESULT     ;Store the doubleword result.
```

Wide Subtract Immediate (Extended Displacement) XWSBI

XWSBI *n*,[@]*displacement*[,*index*]



Function: (E) - *n* → (E)
ALU carry → CRY

Parameters: None

XWSBI subtracts an integer in the range of 1 to 4 from the signed 32-bit integer in memory.

Arguments

n — Integer in range 1 to 4.

Since Assembler takes coded value of *n* and subtracts 1 from it before placing it in immediate field, you 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

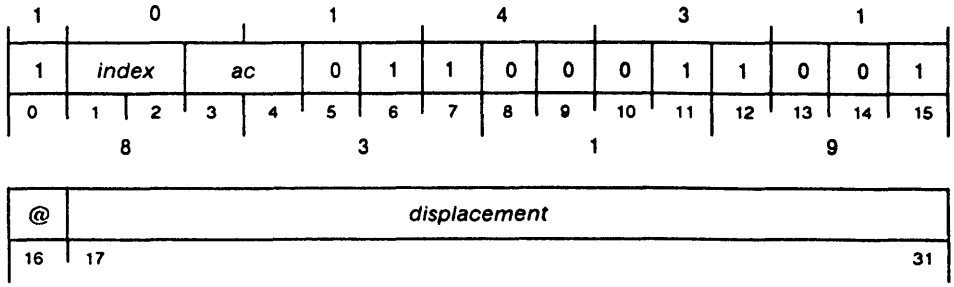
None

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 stores a copy of the 32-bit contents of *ac* into memory.

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

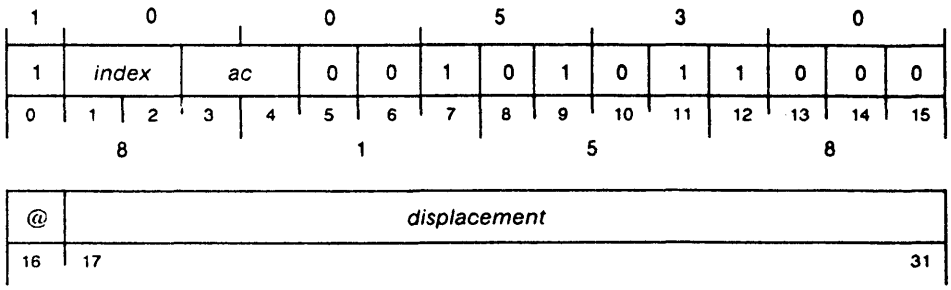
None

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 carry → CRY

Parameters: None

XWSUB subtracts the signed 32-bit integer in memory from the signed 32-bit integer in *ac*. The instruction then loads the result into *ac*.

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

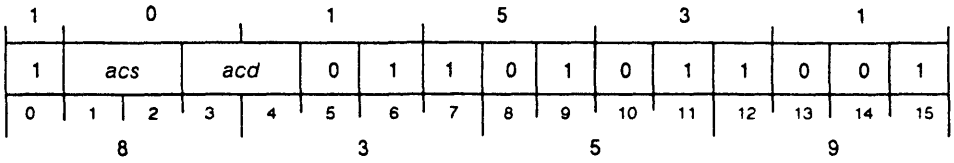
None

Example

```
XWLDA 0,FIRST     ;Get one value (32 bits).
XWSUB 0,SECOND    ;Subtract the second value (32-bit arithmetic).
XWSTA 0,RESULT    ;Store the doubleword 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 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.
```

Index

Within the index, the page number refers to the first page for an entry (even if the subject spans multiple pages). Instruction mnemonics are printed in boldface type (such as **LEF**); instruction names are printed with initial capital letters (such as Load Effective Address).

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

TIPS ORDERING PROCEDURES

TO ORDER

1. An order can be placed with the TIPS group in two ways:
   a) MAIL ORDER - Use the order form on the opposite page and fill in all requested information. Be sure to include shipping charges and local sales tax. If applicable, write in your tax exempt number in the space provided on the order form.

      Send your order form with payment to:
      Data General Corporation
      ATTN: Educational Services/TIPS G155
      4400 Computer Drive
      Westboro, MA 01581-9973

   b) TELEPHONE - Call TIPS at (508) 870-1600 for all orders that will be charged by credit card or paid for by purchase orders over $50.00. Operators are available from 8:30 AM to 5:00 PM EST.

METHOD OF PAYMENT

2. As a customer, you have several payment options:
   a) Purchase Order - Minimum of $50. If ordering by mail, a hard copy of the purchase order must accompany order.
   b) Check or Money Order - Make payable to Data General Corporation.
   c) Credit Card - A minimum order of $20 is required for Mastercard or Visa orders.

SHIPPING

3. To determine the charge for UPS shipping and handling, check the total quantity of units in your order and refer to the following chart:

Total Quantity	Shipping & Handling Charge
1-4 Units	$5.00
5-10 Units	$8.00
11-40 Units	$10.00
41-200 Units	$30.00
Over 200 Units	$100.00

   If overnight or second day shipment is desired, this information should be indicated on the order form. A separate charge will be determined at time of shipment and added to your bill.

VOLUME DISCOUNTS

4. The TIPS discount schedule is based upon the total value of the order.

Order Amount	Discount
$1-$149.99	0%
$150-$499.99	10%
Over $500	20%

TERMS AND CONDITIONS

5. Read the TIPS terms and conditions on the reverse side of the order form carefully. These must be adhered to at all times.

DELIVERY

6. Allow at least two weeks for delivery.

RETURNS

7. Items ordered through the TIPS catalog may not be returned for credit.
8. Order discrepancies must be reported within 15 days of shipment date. Contact your TIPS Administrator at (508) 870-1600 to notify the TIPS department of any problems.

INTERNATIONAL ORDERS

9. Customers outside of the United States must obtain documentation from their local Data General Subsidiary or Representative. Any TIPS orders received by Data General U.S. Headquarters will be forwarded to the appropriate DG Subsidiary or Representative for processing.

TIPS ORDER FORM

Mail To: Data General Corporation
Attn: Educational Services/TIPS G155
4400 Computer Drive
Westboro, MA 01581 - 9973

BILL TO:	SHIP TO: (No P.O. Boxes - Complete Only If Different Address)
COMPANY NAME	COMPANY NAME
ATTN:	ATTN:
ADDRESS	ADDRESS (NO PO BOXES)
CITY	CITY
STATE ZIP	STATE ZIP

Priority Code _____ (See label on back of catalog)

Authorized Signature of Buyer (Agrees to terms & conditions on reverse side) | Title | Date | Phone (Area Code) | Ext.

ORDER #	QTY	DESCRIPTION	UNIT PRICE	TOTAL PRICE

A SHIPPING & HANDLING

☐ UPS	ADD
1-4 Items	$ 5.00
5-10 Items	$ 8.00
11-40 Items	$ 10.00
41-200 Items	$ 30.00
200+ Items	$100.00

Check for faster delivery

Additional charge to be determined at time of shipment and added to your bill.

☐ UPS Blue Label (2 day shipping)
☐ Red Label (overnight shipping)

B VOLUME DISCOUNTS

Order Amount	Save
$0 - $149.99	0%
$150 - $499.99	10%
Over $500.00	20%

Tax Exempt # or Sales Tax (if applicable) _____

ORDER TOTAL	
Less Discount See B	-
SUB TOTAL	
Your local* sales tax	+
Shipping and handling - See A	+
TOTAL - See C	

C PAYMENT METHOD

☐ Purchase Order Attached ($50 minimum)
P.O. number is _____. (Include hardcopy P.O.)
☐ Check or Money Order Enclosed
☐ Visa ☐ MasterCard ($20 minimum on credit cards)

Account Number | Expiration Date

Authorized Signature
(Credit card orders without signature and expiration date cannot be processed.)

THANK YOU FOR YOUR ORDER

PRICES SUBJECT TO CHANGE WITHOUT PRIOR NOTICE.
PLEASE ALLOW 2 WEEKS FOR DELIVERY.
NO REFUNDS NO RETURNS.

* Data General is required by law to collect applicable sales or use tax on all purchases shipped to states where DG maintains a place of business, which covers all 50 states. Please include your local taxes when determining the total value of your order. If you are uncertain about the correct tax amount, please call 508-870-1600.

Form 702
Rev. 8/87

DATA GENERAL CORPORATION
TECHNICAL INFORMATION AND PUBLICATIONS SERVICE
TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form. These terms and conditions apply to all orders, telephone, telex, or mail. By accepting these products the Customer accepts and agrees to be bound by these terms and conditions.

1. CUSTOMER CERTIFICATION
Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

2. TAXES
Customer shall be responsible for all taxes, including taxes paid or payable by DGC for products or services supplied under this Agreement, exclusive of taxes based on DGC's net income, unless Customer provides written proof of exemption.

3. DATA AND PROPRIETARY RIGHTS
Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

4. LIMITED MEDIA WARRANTY
DGC warrants the CLI Macros media, provided by DGC to the Customer under this Agreement, against physical defects for a period of ninety (90) days from the date of shipment by DGC. DGC will replace defective media at no charge to you, provided it is returned postage prepaid to DGC within the ninety (90) day warranty period. This shall be your exclusive remedy and DGC's sole obligation and liability for defective media. This limited media warranty does not apply if the media has been damaged by accident, abuse or misuse.

5. DISCLAIMER OF WARRANTY
EXCEPT FOR THE LIMITED MEDIA WARRANTY NOTED ABOVE, DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS, CLI MACROS OR MATERIALS SUPPLIED HEREUNDER.

6. LIMITATION OF LIABILITY
A. CUSTOMER AGREES THAT DGC'S LIABILITY, IF ANY, FOR DAMAGES, INCLUDING BUT NOT LIMITED TO LIABILITY ARISING OUT OF CONTRACT, NEGLIGENCE, STRICT LIABILITY IN TORT OR WARRANTY SHALL NOT EXCEED THE CHARGES PAID BY CUSTOMER FOR THE PARTICULAR PUBLICATION OR CLI MACRO INVOLVED. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO CLAIMS FOR PERSONAL INJURY CAUSED SOLELY BY DGC'S NEGLIGENCE. OTHER THAN THE CHARGES REFERENCED HEREIN, IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOST PROFITS AND DAMAGES RESULTING FROM LOSS OF USE, OR LOST DATA, OR DELIVERY DELAYS, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY THEREOF; OR FOR ANY CLAIM BY ANY THIRD PARTY.

B. ANY ACTION AGAINST DGC MUST BE COMMENCED WITHIN ONE (1) YEAR AFTER THE CAUSE OF ACTION ACCRUES.

7. GENERAL
A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts, excluding its conflict of law rules. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer. DGC hereby rejects all such different, conflicting, or additional terms.

8. IMPORTANT NOTICE REGARDING AOS/VS INTERNALS SERIES (ORDER #1865 & #1875)
Customer understands that information and material presented in the AOS/VS Internals Series documents may be specific to a particular revision of the product. Consequently user programs or systems based on this information and material may be revision-locked and may not function properly with prior or future revisions of the product. Therefore, Data General makes no representations as to the utility of this information and material beyond the current revision level which is the subject of the manual. Any use thereof by you or your company is at your own risk. Data General disclaims any liability arising from any such use and I and my company (Customer) hold Data General completely harmless therefrom.

ECLIPSE®
MV/Family
(32-Bit)
Systems
Instruction
Dictionary
014-001372-01

Cut here and insert in binder spine pocket

DataGeneral
Data General Corporation, Westboro, Massachusetts 01580

014-001372-01