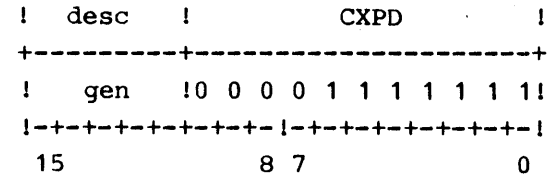
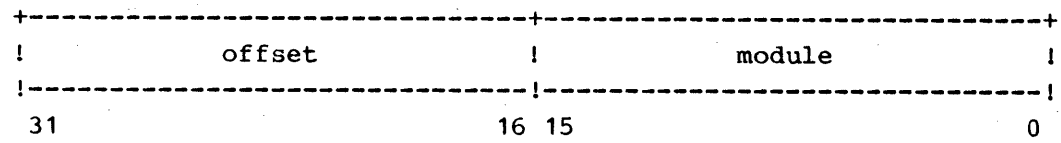
Call External Procedure with Descriptor

Syntax: **CXPD** **desc**
gen
addr



The CXPD instruction calls the external procedure specified by the desc (descriptor) operand. The descriptor is a 32-bit value in the following format:



The instruction does the following:

1. Decrements the current Stack Pointer by 2, then pushes the contents of the MOD register (16 bits) onto the current stack. The stack pointer is modified by a total of four in this step. The extra two bytes placed on the stack are reserved for future use.

2. Saves the address of the next sequential instruction (32 bits) onto the currently-selected stack. This double-word is the return address.

3. Copies the low-order word of the descriptor to the MOD register. The low-order word is the address of the new Module Table entry.

4. Copies the double-word at address MOD+0 to the SB register. This double-word is the Static Base pointer for the new module.

5. Copies to the PC register the sum of the high-order word of the descriptor (interpreted as an unsigned value) and the double-word at address MOD+8. This sum is the address of the external procedure entry point in the new module.

Program execution continues at the address placed in the PC register. The procedure has been invoked, and is running in its own module environment.

Flags Affected: None.

Traps: None.

Example:

```
CXPD  8(SB)                          7F D0 08
```

The above example calls an external procedure whose descriptor is contained at memory address 8(SB).

The instruction is illustrated below:

Operands	Operand Values: Hex Before	After
00009088 * (descriptor)	00100020 (module 0020) (offset 0010)	00100020
MOD	0010	0020
PC	00009005	0000F010
SB	00009080	0000F100
SP	0000FFF8	0000FFF0
Stack:		
0000FFF0	xxxxxxxx	00009007
0000FFF4	xxxxxxxx	xxxx0010
0000FFF8	AAAAAAAA	AAAAAAAA

Module Table:

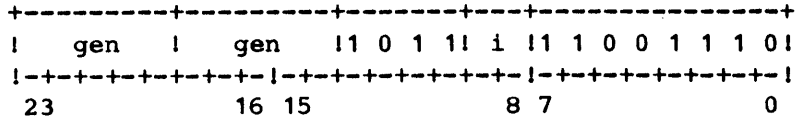
00000010	00009080	(SB)
14	000090A0	(LB)
18	00009000	(PB)
1C	xxxxxxxx	
00000020	0000F100	(SB)
24	0000F110	(LB)
28	0000F000	(PB)
2C	xxxxxxxx	

* 9088 (Hex) is the descriptor's effective address, as specified by 8(SB).

** The 16-bit field shown as "xxxx" is reserved for future use, and should be treated as don't-care bits.

Divide Extended Integer

Syntax: **DEIi** **src,** **dest** DEIB
gen gen DEIW
read.i rmw.2i DEID



The DEIi instruction divides the entire dest operand by the src operand and places the quotient and the remainder in the dest operand location.

The instruction places the quotient in the high-order half of dest and the remainder in the low-order half. The dest operand may be specified as an even-odd general purpose register pair. In such cases, the instruction places the remainder in the even register and the quotient in the next consecutive (odd) register. The register pair must be specified in assembly language by the name of the even register of the pair.

The src and dest operands are interpreted as unsigned integers.

Flags Affected: None.

Traps: DVZ (Divide by Zero) activated if src equals zero.

Example:

```
DEIW  R2, R0                    CE 2D 10
```

The above example divides the double-word value contained in the low-order words of R0 and R1 by the low-order word of register R2. The result is a double-word containing a quotient and a remainder. The remainder is written to the low-order word of register R0; the quotient is written to the low-order word of register R1. The high-order words of registers R0 and R1 are not used or affected.

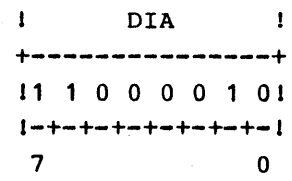
The instruction is illustrated below:

Operands	Operand Values: Hex (Dec) Before	After
R2	AAAA0001 (+1)	AAAA0001 (+1)
R0	BBBBFFFF	BBBB0000
R1	CCCC0000 (+65535)	CCCCFFFF (+65535, rem. 0)

The above case divides 65535 by 1 (H'0000FFFF by H'0001). The quotient is 65535 (in R1), and the remainder is 0 (in R0).

DIA
Diagnose

Syntax: **DIA**



The DIA instruction is intended to support breakpointing circuitry, and is not intended for use in a program. It is a one-byte instruction which performs a branch to itself, establishing an "infinite loop" which is interruptible. When the loop thus established is interrupted, the return address pushed onto the Interrupt Stack is the address of the DIA instruction itself.

Flag Affected: None.

Traps: None.

Example:

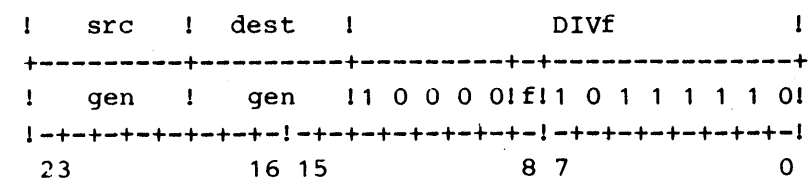
```
DIA                                C2
```

Divide Floating

Syntax: DIVf src, dest

DIVF
DIVL

	src	dest
	gen	gen
	read.f	rmw.f



The DIVf instruction divides the dest operand by the src operand and places the result in the dest operand location.

Results for normalized and zero operands are given in the table below. The symbols "+n" and "-n" represent non-zero normalized numbers, positive and negative, respectively. The symbols "+z" and "-z" represent positive and negative zero, respectively.

dest: / src	+n	-n	+z	-z
+n	*	*	+z	-z
-n	*	*	-z	+z

* The result in these cases is the quotient of the two operands.

Flags Affected: No PSR flags. Two FSR flags may be affected:
UF is set if an underflow occurs; unaffected otherwise.
IF is set on an inexact result; unaffected otherwise.
See Section 3.3 for floating-point exception definitions.

Traps: Undefined Instruction Trap (UND) is activated if the F bit in the CFG register is clear.

Floating-Point Trap (FPU) is activated if a floating-point exception is detected. See Section 3.3. Particularly relevant to floating-point division are the Divide by Zero exception, caused by attempting to divide a non-zero number by zero, and the Invalid Operation exception, caused by attempting to divide zero by zero.

Examples:

```
1.  DIVF   F0, F7                  BE E1 01
2.  DIVL   -8(FP), 16(SB)          BE A0 C6 78 10
```

Example 1 divides the single-precision number in register F7 by the number in register F0 and places the result in F7.

Example 2 divides the double-precision number at address 16(SB) by the number at address -8(FP) and places the result at address 16(SB).

The instructions are illustrated below:

	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	F0	42250000 (+41.25)	42250000 (+41.25)
	F7	434E4000 (+206.25)	40A00000 (+5.0)
Ex. 2:	-8(FP)	409F440000000000 (+2001.0)	409F440000000000 (+2001.0)
	16(SB)	41A2B128DDC00000 (+156800110.875)	40F3218E00000000 (78360.875)

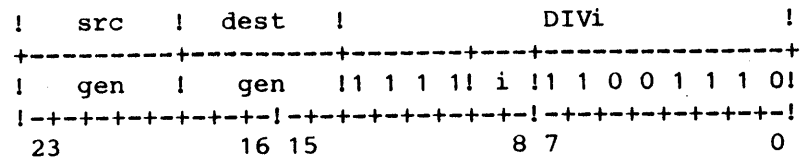
Divide

Syntax: **DIVi** **src,** **dest**

gen gen

read.i rmw.i

DIVB
DIVW
DIVD



The DIVi instruction divides the dest operand by the src operand, rounds the quotient to the next lower (or more negative) integer, and places the result in the dest operand location. The src and dest operands are interpreted as signed integers.

Flags Affected: None.

Traps: DVZ (Divide by Zero) activated if src equals zero.

Examples:

1.	DIVW 10(SP), 4(SP)	CE 7D CE 0A 04
2.	DIVD -6(FP), 12(SB)	CE BF C6 7A 0C

Example 1 divides the word at the address specified by 4(SP) by the word at the address specified by 10(SP). The instruction rounds the quotient and places the result at 4(SP).

Example 2 divides the double-word at the address specified by 12(SB) by the double-word at the address specified by -6(FP). The instruction rounds the quotient and places the result in the double-word at address 12(SB).

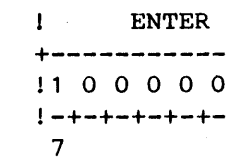
These instructions are illustrated below:

	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	10(SP)	000A (+10)	000A (+10)
	4(SP)	006F (+111)	000B (+11)
Ex. 2:	-6(FP)	00000014 (+20)	00000014 (+20)
	12(SB)	FFFFFF9F (-97)	FFFFFFFB (-5)

In example 1, 111 divided by 10 is 11.1. The next lower integer is 11.
In example 2, -97 divided by 20 is -4.85. The next lower integer is -5.

ENTER
Enter New Procedure Context

Syntax: **ENTER reglist, constant**
imm disp



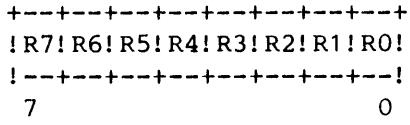
The ENTER instruction creates a "Frame" on the current stack for use by a procedure. A Frame is a block of memory on the stack that provides local storage for the current procedure. The constant operand specifies the number of bytes to be reserved on the stack for local data storage. The Frame Pointer (FP) register is saved and then set up as a pointer from which frame information can be located.

The instruction does the following:

1. Pushes the contents of the FP register (32 bits) onto the stack.
2. Copies the contents of the current stack pointer to the FP register.
3. Subtracts the constant operand from the value of the current stack pointer, lengthening the stack by that number of bytes.
4. Pushes the general-purpose registers specified by reglist onto the stack.

The reglist operand is specified in assembly language by a list of zero or more general-purpose register names, enclosed in brackets "[]". The instruction pushes the contents of each register in the list as a double-word onto the currently-selected stack. Register names may appear in any order within reglist but must be separated by commas. Brackets are required even if no register names are given.

In the machine instruction, the reglist operand is encoded in an eight-bit field as shown below. Each bit in the field corresponds to one general-purpose register. When the instruction is executed, the instruction reads the bits in the field from right to left beginning with bit 0. If a bit is "0", the instruction ignores the corresponding register. If a bit is "1", it saves the corresponding register.



Flags Affected: None.

Traps: None.

Enter New Procedure Context (continued)

Example:

```
ENTER  [R0, R2, R7], 16          82 85 10
```

This instruction creates a frame on the stack consisting of 16 bytes for local data storage and the contents of register R0, R2, and R7.

In the following illustration, the PSR S flag is assumed to be 1, selecting SP1 as the current stack pointer.

Operands	Operand Values: Hex Before	After
R0	00000010	00000010
R2	FFFFFFEF	FFFFFFEF
R7	FFFFF9AB	FFFFF9AB
16 (disp)	10 (+16)	--
FP	00001000	000010EC
SP1	000010F0	000010D0
Stack:		
000010D0	xxxxxxxx	FFFFF9AB
000010D4	xxxxxxxx	FFFFFFEF
000010D8	xxxxxxxx	00000010
000010DC	xxxxxxxx	xxxxxxxx *
000010E0	xxxxxxxx	xxxxxxxx
000010E4	xxxxxxxx	xxxxxxxx
000010E8	xxxxxxxx	xxxxxxxx
000010EC	xxxxxxxx	00001000
000010F0	AAAAAAAA	AAAAAAAA

* 16 bytes of uninitialized local data storage.

EXIT
Exit Procedure Context

Syntax: **EXIT** **reglist**
imm

```
!      EXIT       !
+-----------------+
!1 0 0 1 0 0 1 0!
!-+-+-+-+-+-+-+-+
 7               0
```

The EXIT instruction removes the frame of the current procedure from the stack, restores the old contents of the specified general-purpose registers (i.e., their contents prior to entering the current procedure), and restores the frame of the previous procedure as the current procedure context.

The instruction does the following:

1. Restores the general-purpose registers specified by reglist by popping them from the current stack.

2. Copies the contents of the FP register to the current stack pointer.

3. Pops the old frame address (32 bits) from the stack to the FP register.

In assembly language, the reglist operand is specified as a list of zero or more general-purpose register names, enclosed in brackets "[]". The instruction copies to each register in the list a double-word popped from the stack. Register names may appear in any order within reglist but must be separated by commas. Brackets are required even if no register names are given.

In the machine instruction, the reglist operand is encoded in an eight-bit field as shown below. Each bit in the field corresponds to one general-purpose register. When the instruction is executed, the instruction reads the bits in the field from right to left beginning with bit 0. If a bit is "0", the instruction ignores the corresponding register. If a bit is "1", it restores the corresponding register from the stack. Note that the format of the reglist operand is reversed from its format in the ENTER instruction; i.e. bit 0 corresponds to register R7 instead of R0.

```
+---+---+---+---+---+---+---+---+
!R2!R3!R4!R5!R6!R7!
+---+---+---+---+---+---+---+---!
                                0
```

Flags Affected: None.

Traps: None.

Exit Procedure Context (continued)

Example:

```
EXIT  [R0, R2, R7]                92 A1
```

This instruction restores the contents of the listed general-purpose registers, reclaims the frame of the current procedure, and restores the frame of the previous procedure as the current context.

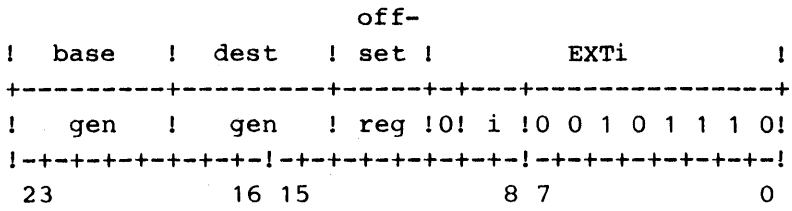
The instruction is illustrated below:

Operands	Operand Values: Hex Before	After
R0	CCCCCCCC	00000010
R2	CCCCCCCC	FFFFFFEF
R7	CCCCCCCC	FFFFF9AB
FP	000010EC	00001000
SP	000010D0	000010F0
Stack:		
000010D0	FFFFF9AB	xxxxxxxx *
000010D4	FFFFFFEF	xxxxxxxx *
000010D8	00000010	xxxxxxxx *
000010DC	BBBBBBBB	xxxxxxxx *
000010E0	BBBBBBBB	xxxxxxxx *
000010E4	BBBBBBBB	xxxxxxxx *
000010E8	BBBBBBBB	xxxxxxxx *
000010EC	00001000	xxxxxxxx *
000010F0	AAAAAAAA	AAAAAAAA

* The EXIT instruction does not itself change the contents of these memory locations. However, information that is outside the stack should be considered unpredictable for other reasons. See Section 2.7.1.

EXTi
Extract Field

Syntax: **EXTi** **offset,** **base,** **dest,** **length** EXTB
reg gen gen disp EXTW
regaddr write.i EXTD



The EXTi instruction copies the bit field specified by base, offset and length to the dest operand location. The field is right-justified in dest. High-order bits are zero-filled if the field is shorter than dest or discarded if the field is longer than dest.

The location of the field is taken from the position of its least-significant bit, given by offset and base as follows:

If base is a register, then the field is within that register, starting at the bit position given by offset. If base is a memory location, then the field starts at bit position

offset MOD 8

within the memory byte whose address is

EA(base) + (offset DIV 8),

where EA(base) is the effective address of base. See Section 3.6 for definitions of the operators MOD and DIV above.

Offset is interpreted as a 32-bit signed integer.

Length specifies the number of bits in the field. It must be in the range 1 through 32.

See Section 3.6 for further details of specifying bit fields.

NOTE: Although a bit field may contain up to 32 bits, an alignment restriction appears for fields containing more than 25 bits: a field may not span more than four bytes. See Section 3.6.

Flags Affected: None.

Traps: None.

Extract Field (continued)

Example:

```
EXTW  R0, 0(R1), R2, 7          2E 81 40 00 07
```

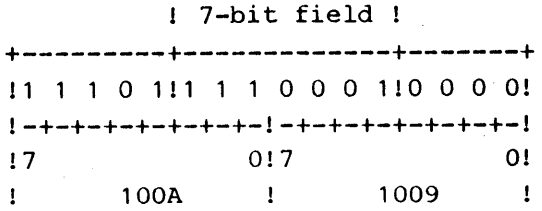
This example copies a 7-bit field from memory into the low-order word of register R2. Bits 7 through 15 of register R2 are set to zero and the remaining bits of R2 are unaffected. For designating the location of the field, register R0 supplies the bit offset, and 0(R1) is specified as the base address.

The instruction is illustrated below:

Operands	Operand Values: Hex (Dec) Before	After
R0 (offset)	0000004C (+76)	0000004C (+76)
R1	00001000 (+4096)	00001000 (+4096)
base address 0(R1)	00001000 (+4096)	--
R2	AAAAAAAA	AAAA0071
00001009 * (+4105)	EF10	EF10 **

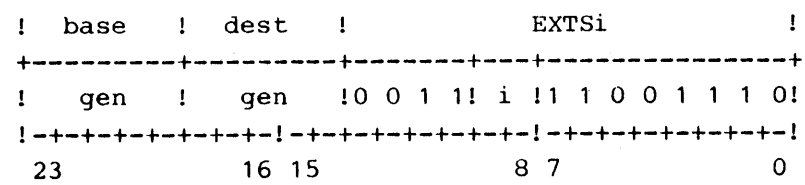
* The address 1009 (Hex) is the effective address of the byte containing the least-significant bit of the specified field. This address is computed as 4096 + (76 DIV 8) = 4105, where 4096 is the base address specified by 0(R1) and 76 is the bit offset given by the contents of register R0.

** The bit field starts at bit position 4 (= 76 MOD 8) in the byte at address 1009 (Hex) and is seven bits long as illustrated below.



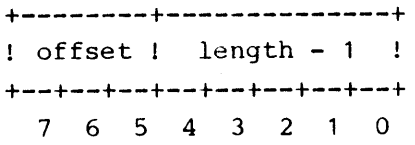
EXTSi
Extract Field Short

Syntax: **EXTSi** **base,** **dest,** **offset, length** EXTSB
gen gen !------imm-----! EXTSW
regaddr write.i EXTSD



The EXTSi instruction copies the bit field specified by base, offset, and length to the dest operand location. The field is right-justified in dest. High-order bits are zero-filled if the field is shorter than dest or discarded if the field is longer than dest.

The offset and length operands are encoded together as an immediate byte appended to the basic instruction. The offset is encoded as the high-order three bits of this byte; the length operand, minus one, is encoded as the low-order five bits. The byte has the following form:



The offset value must be in the range 0 through 7. The length value specifies the number of bits in the field. It must be in the range 1 through 32.

The location of the field is taken from the position of its least-significant bit. If base is a register, then the field is within that register, starting at the bit position given by offset. If base is a memory location, then the field starts at the bit position given by offset within the memory byte whose address is given as base.

See Section 3.6 for further details of specifying bit fields.

NOTE: Although a bit field may contain up to 32 bits, an alignment restriction appears for fields containing more than 25 bits: a field may not span more than four bytes. See Section 3.6.

Flags Affected: None.

Traps: None.

Extract Field Short (continued)

Example:

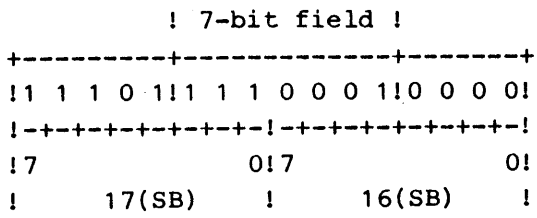
```
EXTSW  16(SB), R2, 4, 7          CE 8D D0 10 86
```

This example copies a bit field to the low-order word of register R2. The field begins at bit position 4 of the byte at the address specified as 16(SB) and is seven bits long.

The instruction is illustrated below:

Operands	Operand Values: Hex Before	After
R2	AAAAAAAA	AAAA0071 **
16(SB)	EF10	EF10 *

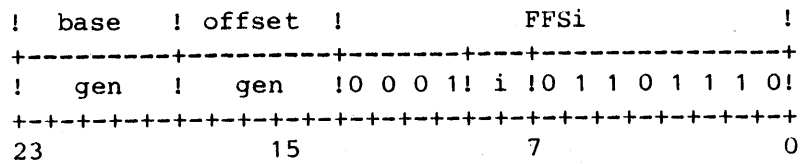
* The bit field starts at bit number 4 in the byte at address 16(SB) and is seven bits long as illustrated below:



** The bit field is right-justified in the low-order word of register R2. Nine leading zero bits are added to the bit field to fill the low-order word.

FFSi
Find First Set Bit

Syntax: **FFSi** **base,** **offset** FFSB
gen gen FFSW
read.i rmw.B



The FFSi instruction searches for the first "1" bit in the base operand. The search starts at the bit specified by the offset operand and proceeds in ascending order to the first "1" bit or to the last bit in base.

If a "1" bit is found, the instruction sets the offset operand value to the bit number of the first "1" bit found in the base operand and clears the F flag in the PSR.

If a "1" bit is not found, the instruction sets the offset operand value to zero and sets the F flag in the PSR to 1.

The offset is interpreted as an unsigned number. Its value must be within the range 0 to 7 (in FFSB instruction), 0 to 15 (in FFSW instruction), and 0 to 31 (in FFSD instruction), otherwise the result placed in the offset operand and in the F bit is undefined.

Note: If the FFSi instruction finds a "1" bit and it is desired to scan the remaining portion of the base operand, the offset operand must be incremented first or the "1" bit previously found must be cleared. Otherwise, the FFSi instruction will detect that bit again.

Flags Affected: F is set if a "1" bit is not found, cleared if found.

Traps: None.

Examples:

```
1.  FFSW  8(SB), R0          6E 05 D0 08
2.  FFSB  -4(FP), TOS        6E C4 C5 7C
```

Example 1 searches the word at the address specified by 8(SB) for the first "1" bit. The search begins at the bit specified by the low-order byte of register R0, and the result is placed in the low-order byte of R0. The remaining portion of R0 is not used or affected.

Example 2 searches the byte at the address specified by -4(FP). The search begins at the bit specified by the byte on the top of the stack, which is replaced by the resulting bit number.

Find First Set Bit (continued)

The preceding instructions are illustrated below:

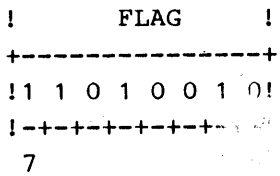
	Operands	Operand Values: Hex (Dec) [Binary] Before	After
Ex. 1:	R0	AAAAAA05 (+5)	AAAAAA08 (+8)
	8(SB)	EF10 [1110111100010000]	EF10 [1110111100010000]
	UPSR	nzfxxltc	nz0xxltc
Ex. 2:	SP	0000FFDE	0000FFDE
	Stack: 0000FFDE	05 (+5)	00 (0)
	-4(FP)	10 [00010000]	10 [00010000]
	UPSR	nzfxxltc	nz1xxltc

In example 1, the instruction finds the first "1" bit at bit position 8.

In example 2, the instruction finds no "1" bits; that is, the bits at bit positions 5, 6, and 7 are all "0" bits.

FLAG
Trap on Flag

Syntax: **FLAG**



The FLAG instruction activates the Flag Trap (FLG) if the F flag in the PSR is set. The Flag Trap passes control to the Flag service procedure (see Chapter 6). The return address pushed on the Interrupt Stack is the address of the FLAG instruction itself (see Chapter 6). If the F flag is not set, program execution continues with the next sequential instruction.

Flags Affected: None.

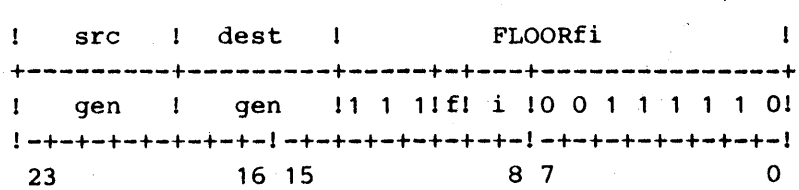
Traps: The Flag Trap (FLG) is activated if the F flag is set.

Example:

```
FLAG                    D2
```

Floor Floating to Integer

Syntax: **FLOORfi** **src,** **dest** FLOORFB FLOORLB
gen gen FLOORFW FLOORLW
read.f write.i FLOORFD FLOORLD



The FLOORfi instruction rounds the src operand to the nearest integer less than or equal to it (i.e., toward negative infinity) and places the result in the dest operand location as a signed integer.

Flags Affected: No PSR flags. One FSR flag may be affected:
IF is set on an inexact result; unaffected otherwise.
See Section 3.3 for floating-point exception definitions.

Traps: Undefined Instruction Trap (UND) is activated if the F bit in the CFG register is clear.

Floating-Point Trap (FPU) is activated if a floating-point exception is detected. See Section 3.3. Particularly relevant to this instruction is the Overflow exception, which is caused by attempting to convert a floating-point number that is too great in absolute value to be held in a signed integer of the size specified for dest.

Examples:

```
1.  FLOORFB  F0, R0              3E 3C 00
2.  FLOORLD  F2, 16(SB)          3E BB 16 10
```

Example 1 rounds the single-precision number in register F0 to a byte-long integer and copies the integer to the low-order byte of register R0.

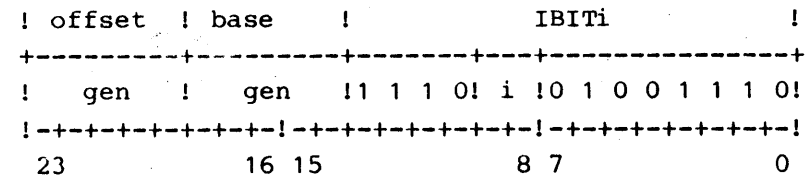
Example 2 rounds the double-precision number in register pair (F2,F3) to a double-word-long integer and copies the integer to address 16(SB).

These instructions are illustrated below:

		Operand Values: Hex (Dec)	
	Operands	Before	After
Ex. 1:	F0	C0280000 (-2.65)	C0280000 (-2.65)
	R0	AAAAAAAA	AAAAAAFD (-3)
Ex. 2:	(F2,F3)	41C0200888700000 (+541069584.875)	41C0200888700000 (+541069584.875)
	16(SB)	AAAAAAAA	20401110 (+541069584)

Invert Bit

Syntax: **IBITi** **offset,** **base** IBITB
gen gen IBITW
read.i regaddr IBITD



The IBITi instruction inverts (complements) the register or memory bit specified by base and offset after copying the bit value to the F flag in the PSR.

The location of the bit is determined from offset and base. Offset is a general operand, whose length is given by the operation length suffix. Base is an addressing expression giving a byte address from which offset specifies a bit position. See Section 3.5 for details of specifying bit positions.

If base is a register, then the bit is within that register, at the bit position given by the offset operand. If base is a memory location, then the bit is at bit position

offset MOD 8

within the memory byte whose address is

EA(base) + (offset DIV 8),

where EA(base) is the effective address of base. See Section 3.5 for definitions of the operators MOD and DIV above, and for further details of bit instructions.

Offset is interpreted as a signed integer.

Flags Affected: F is set to the original value of the specified bit.

Traps: None.

Example:

```
IBITW  R0, 1(R1)                    4E 79 02 01
```

This example inverts a bit in memory after copying the bit value into the F flag. For designating the location of the target bit, the low-order word of register R0 supplies the bit offset, and 1(R1) is specified as the base address.

In the following illustration, the target bit is assumed to be 0 prior to instruction execution.

Operands	Operand Values: Hex (Dec) [Binary] Before	After
R0 (offset)	AAAA004C (+76)	AAAA004C (+76)
R1	00001000 (+4096)	00001000 (+4096)
base address 1(R1)	00001001 (+4097)	--
0000100A * (+4106)	EF [11101111]	FF [11111111]
UPSR	nzfxxltc	nz0xxltc

* The address 100A (Hex) is the effective address of the byte containing the desired bit. This address is computed from the offset and the base address as follows:

```
base address + (offset DIV 8)
   4097      +       9
   4106, or 100A (Hex)  .
```

The bit number within this byte is calculated as:

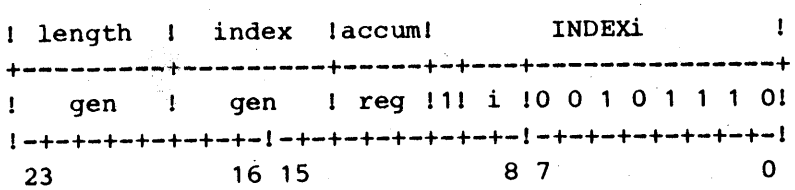
```
offset MOD 8
  76   MOD 8
     4  .
```

Calculate Index

Syntax: **INDEXi** **accum,** **length,** **index**

	reg	
	gen	gen
	read.i	read.i

INDEXB
INDEXW
INDEXD



The INDEXi instruction assists the programmer in accessing multi-dimensional arrays by providing a one-dimensional index which can subsequently be used directly in an addressing mode with Scaled Indexing. The one-dimensional index is calculated from the values of the indices along each dimension of the array.

This instruction is intended to be executed iteratively, as discussed in Section 3.9, once for each dimension except the first. Each iteration accumulates its result into the general-purpose register specified as accum. The length operand defines the length of the current dimension, giving the difference between the upper and lower index bounds (this is the actual dimension length minus one). The index operand is the zero-adjusted value along the current dimension. The result placed in the accum register is:

```
accum * (length + 1) + index .
```

The length and index operands are interpreted as unsigned integers, and are zero-extended to 32 bits internally before use. The accum operand is interpreted as an unsigned 32-bit integer.

Flags Affected: None.

Traps: None.

Example:

```
INDEXB  R0, 20(SB), -4(FP)          2E 04 D6 14 7C
```

This example performs one step of an index calculation. R0 is the accum operand, memory location 20(SB) holds a byte defining the length of the current array dimension, and memory location -4(FP) holds the index value along this dimension.

The case below shows the application of the above instruction to calculate the one-dimensional index of array element A[I,J], where A has been declared (in the Pascal language) as being of dimensions [1..7, 0..16]. The array is assumed to be stored in row major order (Section 3.9). Since it is an array of only two dimensions, one INDEXi instruction serves to calculate the one-dimensional index.

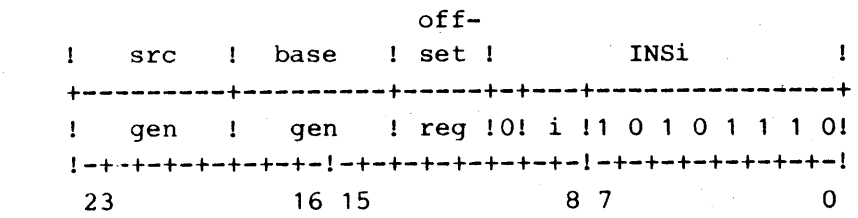
The value of index I (assumed to be 4) has been zero-adjusted to 3 by a CHECK instruction (q.v.), and the result placed in register R0 as a double-word. The value of index J, held in one byte at address -4(FP), is assumed to be 3. The byte at location 20(SB) holds the length operand for the second dimension of the array (16 - 0 = 16).

The result in R0, 54, is the final one-dimensional index of element [4,3] of array A. This value can be used directly in any addressing mode with a Scaled Indexing modifier to access this array element.

Operands	Operand Values: Hex (Dec) Before	After
R0	00000003 (+3)	00000036 (+54)
20(SB)	10 (+16)	10 (+16)
-4(FP)	03 (+3)	03 (+3)

Insert Field

Syntax: **INSi** **offset,** **src,** **base,** **length** INSB
reg gen gen disp INSW
read.i regaddr INSD



The INSi instruction inserts the src operand into the bit field specified by base, offset, and length. The src operand is right-justified in the field. High-order bits are zero-filled if src is shorter than the field or discarded if src is longer than the field.

The location of the field is taken as the position of its least-significant bit, given by offset and base as follows:

If base is a register, then the field is within that register, starting at the bit position given by offset. If base is a memory location, then the field starts at bit position

offset MOD 8

within the memory byte whose address is

EA(base) + (offset DIV 8),

where EA(base) is the effective address of base. See Section 3.6 for definitions of the operators MOD and DIV above.

Offset is interpreted as a 32-bit signed integer.

Length specifies the number of bits in the field. It must be in the range 1 through 32.

See Section 3.6 for further details of specifying bit fields.

NOTE: Although a bit field may contain up to 32 bits, an alignment restriction appears for fields containing more than 25 bits: a field may not span more than four bytes. See Section 3.6.

Flags Affected: None.

Traps: None.

Example:

```
INSW  R0, R2, 0(R1), 7          AE 41 12 00 07
```

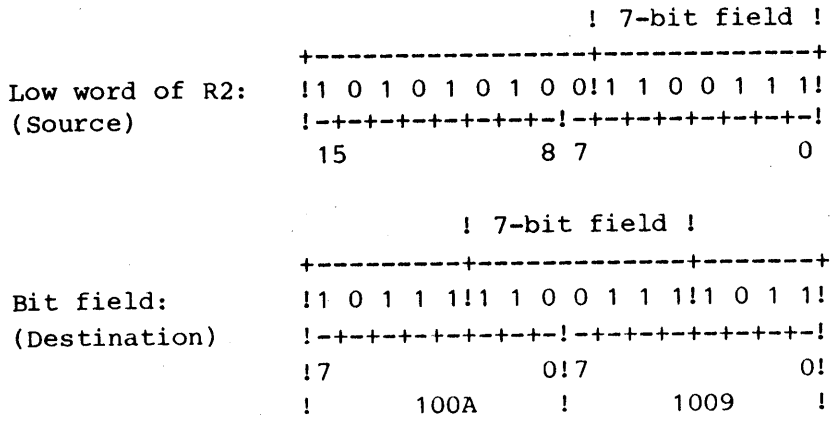
This example inserts 7 bits from the low-order word of register R2 into a bit field in memory. For specifying the location of the field, register R0 supplies the bit offset, and 0(R1) is specified as the base address.

The instruction is illustrated below:

Operands	Operand Values: Hex Before	After
R0 (offset)	0000004C (+76)	0000004C (+76)
R1	00001000 (+4096)	00001000 (+4096)
base address 0(R1)	00001000 (+4096)	00001000 (+4096)
R2	AAAAAA67	AAAAAA67
00001009 * (+4105)	BBBB	BE7B **

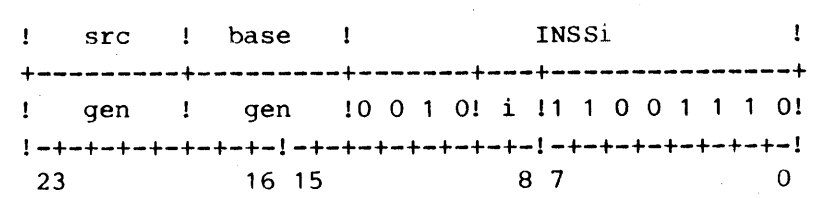
* The address 1009 (Hex) is the effective address of the byte containing the least-significant bit of the specified field. This address is computed as 4096 + (76 DIV 8) = 4105, where 4096 is the address specified by 0(R1) and 76 is the bit offset given by the contents of register R0.

** The bit field starts at bit position 4 (= 76 MOD 8) in the byte at address 1009 (Hex) and is seven bits long as illustrated below.



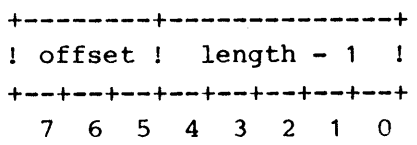
Insert Field Short

Syntax: **INSSi** **src,** **base** **offset, length** INSSB
gen gen !-----imm------! INSSW
read.i regaddr INSSD



The INSSi instruction inserts the src operand into the bit field specified by base, offset, and length. The src operand is right-justified in the field. High-order bits are zero-filled if src is shorter than the field or discarded if src is longer than the field.

The offset and length operands are encoded together as an immediate byte appended to the basic instruction. The offset is encoded as the high-order three bits of this byte; the length operand, minus one, is encoded as the low-order five bits. The byte has the following form:



The offset value must be in the range 0 through 7. The length value specifies the number of bits in the field. It must be in the range 1 through 32.

The location of the field is taken from the position of its least-significant bit. If base is a register, then the field is within that register, starting at the bit position given by offset. If base is a memory location, then the field starts at the bit position given by offset within the memory byte whose address is given as base.

See Section 3.6 for further details of specifying bit fields.

NOTE: Although a bit field may contain up to 32 bits, an alignment restriction appears for fields containing more than 25 bits: a field may not span more than four bytes. See Section 3.6.

Flags Affected: None.

Traps: None.

Example:

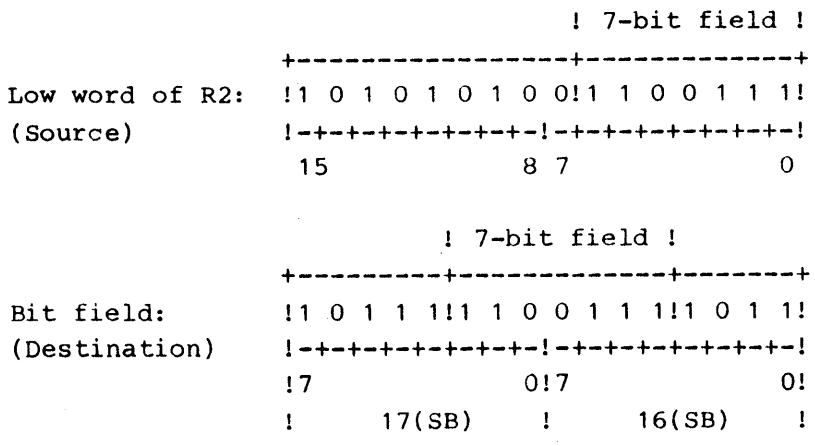
```
INSSW  R2, 16(SB), 4, 7          CE 89 16 10 86
```

This example inserts 7 bits from the low-order word of register R2 into a bit field in memory. The bit field begins at bit position 4 in the byte at the address specified by 16(SB) and is 7 bits long.

The instruction is illustrated below:

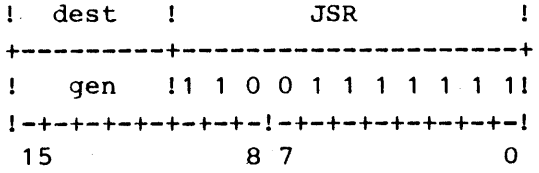
Operands	Operand Values: Hex Before	After
R2	AAAAAA67	AAAAAA67
16(SB)	BBBB	BE7B *

* The bit field starts at bit number 4 in the byte at address 16(SB) and is seven bits long as illustrated below:



Jump to Subroutine

Syntax: **JSR dest**
gen
addr



The JSR instruction jumps to the procedure at the address specified by dest after saving the return address on the stack. The return address is the address of the next sequential instruction.

Flags Affected: None.

Traps: None.

Example:

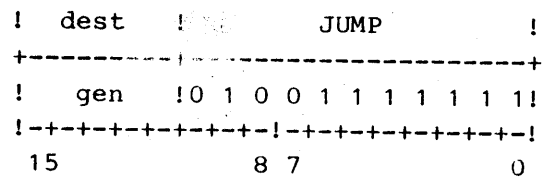
```
JSR 0(4(SB))                    7F 96 04 00
```

This example causes the program to jump to a procedure at the address held within a double-word at address 4(SB). This is accomplished via the Static Memory Relative addressing mode. The instruction saves the address of the next sequential instruction on the stack.

The instruction is illustrated below:

Operand	Operand Values: Hex Before	After
PC	00009000	00001FFF
4(SB)	00001FFF	00001FFF
SP	0000FFD4	0000FFD0
Stack:		
0000FFD0	xxxxxxxx	00009004
0000FFD4	AAAAAAAA	AAAAAAAA

Syntax: **JUMP** **dest**
gen
addr



The JUMP instruction jumps to the address specified by dest by loading the effective address of dest into the PC register.

Flags Affected: None.

Traps: None.

Example:

```
JUMP 0(-8(FP))                    7F 82 78 00
```

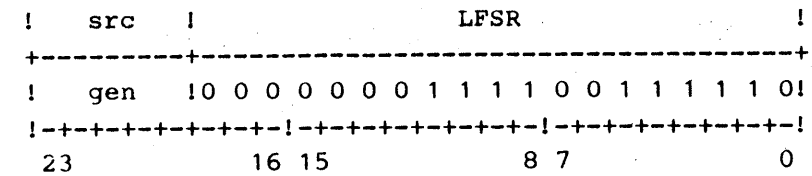
This example loads the address held in the double-word at address -8(FP) into the PC register. This is accomplished via the Frame Memory Relative addressing mode. Program execution continues at that address.

The instruction is illustrated below:

Operand	Operand Values: Hex Before	After
-8(FP)	00001004	00001004
PC	0000909A	00001004

Load Floating-Point Status Register (FSR)

Syntax: **LFSR** **src**
gen
read.D



The LFSR instruction copies the double-word specified by src to the Floating Point Status register (FSR). See Section 2.4.2 for the format of the FSR.

Flags Affected: No PSR flags. All FSR flags are affected.

Traps: Undefined Instruction Trap (UND) is activated if the F bit in the CFG register is clear.

Example:

```
LFSR  R0                          3E 0F 00
```

The above example copies the contents of register R0 into the FSR.

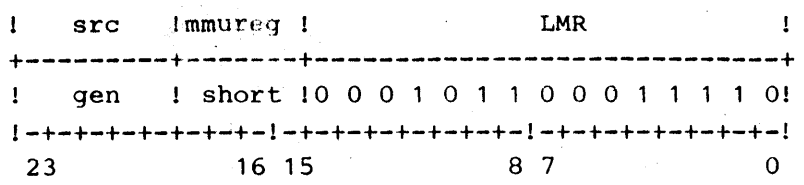
The instruction is illustrated below:

Operands	Operand Values: Hex Before	After
R0	00000028	00000028
FSR	xxxx0129	xxxx0028

LMR

Load Memory Management Register

Syntax: **LMR** **mmureg, src**
short gen
read.D



The LMR instruction copies the src operand to the Memory Management register specified by mmureg.

The LMR instruction may load the following registers. The short field of the basic instruction holds a four-bit value which addresses the corresponding Memory Management register as shown below.

Register	mmureg	short field
Breakpoint Register 0	BPR0	0000
Breakpoint Register 1	BPR1	0001
Program Flow Register 0	PF0	0100
Program Flow Register 1	PF1	0101
Sequential Count Registers	SC	1000
Memory Management Status Register	MSR	1010
Breakpoint Count Register	BCNT	1011
Page Table Base Register 0	PTB0	1100
Page Table Base Register 1	PTB1	1101
Error/Invalidate Address Register	EIA	1111

Flags Affected: None.

Traps: Undefined Instruction Trap (UND) is activated if the M bit in the CFG register is clear. The instruction is not executed.

Illegal Instruction Trap (ILL) is activated if the U flag is set. The instruction is not executed.

Load Memory Management Register (continued)

Example:

```
LMR  BCNT, R0                    1E 8B 05
```

This example copies the contents of register R0 to the Breakpoint Count Register.

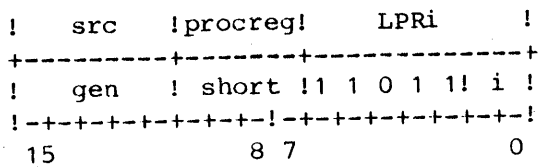
The instruction is illustrated below:

Operands	Operand Values: Hex Before	After
R0	00009000	00009000
BCNT	xxAAAAAA	xx009000

LPRi
Load Processor Register

Syntax: **LPRi** **procreg, src**
short gen
read.i

LPRB
LPRW
LPRD



The LPRi instruction copies the src operand to the dedicated register specified by procreg. See Section 2.2 for the formats of these registers.

The src operand is right-justified in the register. High-order register bits are zero-filled if src is shorter than the register. High-order src bits are discarded if src is longer than the register.

The Load Processor Register instruction may load the following registers. The specified procreg corresponds to the 4-bit short field in the basic instruction as shown below.

Register	procreg	short	
User PSR	UPSR	0000	(Note 3)
Frame Pointer	FP	1000	
Stack Pointer	SP	1001	(Note 1)
Static Base Register	SB	1010	(Note 2)
Processor Status Register	PSR	1101	(Note 3)
Interrupt Base Register	INTBASE	1110	(Note 3)
Module Register	MOD	1111	

NOTES:
1. If SP is specified in the instruction and the S flag in the PSR is set, the instruction copies the src operand to the SP1 register. If the S flag is clear, the instruction copies the operand to the SP0 register.

2. The SB register should not be loaded using LPRi except during system initialization after a Reset, because it is automatically reloaded from the current Module Table entry whenever an external procedure returns, or a trap or interrupt service procedure returns.

3. Access to the PSR or INTBASE register is privileged. If the U flag in the PSR is "1" when this is attempted, the Illegal Operation trap (ILL) occurs instead. If UPSR is specified, however, only the low-order eight bits of the PSR are affected. This is not a privileged operation, and no trap occurs.

Flags Affected: All flags are affected when the PSR register is specified. The N, Z, F, L, T and C flags are affected when UPSR is specified. No flags are affected otherwise.

Load Processor Register (continued)

Traps: Illegal Operation Trap (ILL) is activated if the U flag is set and PSR or INTBASE is specified.

Examples:

```
1.  LPRD  FP, R0          6F 04
2.  LPRW  MOD, 4(SB)      ED D7 04
```

Example 1 loads the entire FP register from register R0.

Example 2 copies the word at address 4(SB) into the MOD register.

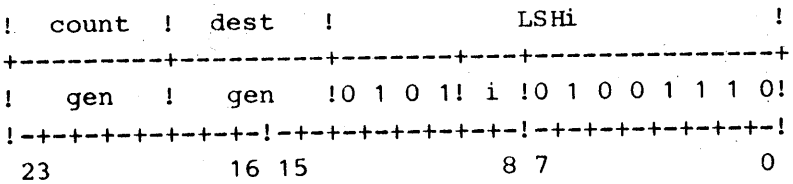
These instructions are illustrated below:

	Operands	Operand Values: Hex Before	After
Ex. 1:	FP	00AAAAAA	00543210
	R0	00543210	00543210
Ex. 2:	MOD	AAAA	0030
	4(SB)	0030	0030

LSHi
Logical Shift

Syntax: **LSHi** **count,** **dest**
gen gen
read.B rmw.i

LSHB
LSHW
LSHD



The LSHi instruction performs a logical shift on the dest operand in the manner specified by the count operand. The sign of count determines the direction of the shift. The absolute value of count gives the number of bit positions to shift the dest operand.

The count operand value must be within the range -7 to +7 for the LSHB form, -15 to +15 for the LSHW form, and -31 to +31 for the LSHD form. A positive count specifies a left shift; a negative count specifies a right shift. In a logical shift, all bits shifted out of dest are lost, and all bit positions emptied by the shift are zero-filled.

The count operand is interpreted as a signed integer. The dest operand is interpreted as an unsigned integer.

Flags Affected: None.

Traps: None.

Logical Shift (continued)

Examples:

```
1.  LSHB   4, 8(SB)          4E 94 A6 04 08
2.  LSHB   -4(FP), 8(SB)     4E 94 C6 7C 08
```

Example 1 shifts the one-byte operand at address 8(SB) four bit positions to the left.

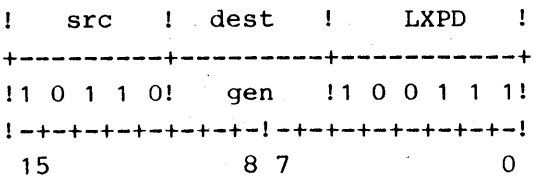
Example 2 shifts the operand at address 8(SB) according to the count given by the byte at address -4(FP). This value, -1, causes a one-bit logical right shift.

These instructions are illustrated below:

	Operands	Operand Values: Binary (Dec) Before	After
Ex. 1:	4 (immediate)	00000100 (+4)	--
	8(SB)	11111110	11100000
Ex. 2:	-4(FP)	11111111 (-1)	11111111 (-1)
	8(SB)	11111110	01111111

LXPD
Load External Procedure Descriptor

Syntax: **LXPD** **src, dest**
gen gen
addr write.D



The LXPD instruction is a specific form of the ADDR instruction whose effect is to copy an external procedure descriptor (Section 2.7.3) from a specified entry of the current Link Table into the dest operand location. The mnemonic "LXPD" is provided by the ASM16 assembler as an alternative to "ADDR" when this use is intended. See the NS16000 Cross-Assembler Reference Manual for further details.

Flags Affected: None.

Traps: None.

Example:

```
LXPD  EXT(3),TOS                    E7 B5 03 00
```

This example copies the external procedure descriptor from Link Table entry number 3, pushing it as a double-word onto the currently-selected stack.

The instruction is illustrated below:

Operands	Operand Values: Hex Before	After
Link Table Entry 3	01000030 (module 0030) (offset 0100)	01000030
SP	0000FFE4	0000FFE0
Stack:		
0000FFE0	xxxxxxxx	01000030
0000FFE4	AAAAAAAA	AAAAAAAA

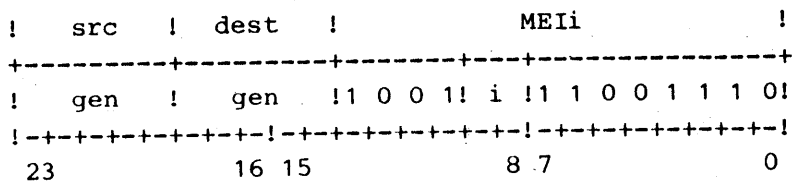
Multiply Extended Integer

Syntax: **MEIi src, dest**

gen gen

read.i rmw.2i

MEIB
MEIW
MEID



The MEIi instruction multiplies the src operand and the low-order half of the dest operand and places the result in the entire dest operand location.

The src and dest operands are interpreted as unsigned integers.

The dest operand may be specified as an even-odd general-purpose register pair. In such cases, the instruction reads the even-numbered register of the pair and places the low-order half (1, 2 or 4 bytes) of the result in the even register and the high-order half in the next consecutive register. The register pair must be specified in assembly language by the name of the even register of the pair.

If the Top of Stack (TOS) addressing mode is used for the dest operand, the Stack Pointer contents do not change. Note that this is not the same as popping a value of length "i" and pushing a result of length "2i". Space must already have been allocated on the stack to accommodate the entire result.

Flags Affected: None.

Traps: None.

Examples:

1.	MEIW R2, 10(SB)	CE A5 16 0A
2.	MEIW R2, R0	CE 25 10

Example 1 multiplies the low-order word of register R2 and the word at the dest operand address 10(SB) and places the double-word result at the dest operand address 10(SB).

Example 2 multiplies the low-order word of register R2 and the low-order word of register R0. The result is a double-word. The low-order word of the result is written to the low-order word of register R0, the high-order word is written to the low-order word of register R1.

These instructions are illustrated below:

	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	R2	AAAA0020 (+32)	AAAA0020 (+32)
	10(SB)	BBBB1001 (+4097)	00020020 (+131104)
Ex. 2:	R2	AAAA0020 (+32)	AAAA0020 (+32)
	R0	BBBB1001	BBBB0020
	R1	CCCCCCCC (+4097)	CCCC0002 (+131104)