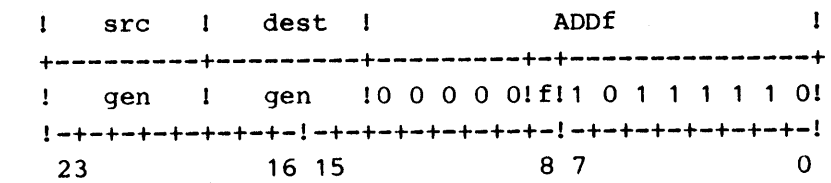
Add Floating

Syntax: **ADDf** **src,** **dest**

gen gen

read.f rmw.f

ADDF
ADDL



The ADDf instruction adds the src and dest operands and places the result in the dest operand location. Results for normalized and zero operands are given in the table below. The symbols "m" and "n" represent any non-zero normalized numbers. The symbols "+z" and "-z" represent positive zero and negative zero, respectively.

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

* These cases, when the result is zero, select the result based on the current rounding mode selected in the FSR. If the "Round toward Negative Infinity" mode is selected, then the result returned is negative zero. Otherwise, the result returned is positive zero.

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.

Examples:

```
1.  ADDF  F0, F7              BE C1 01
2.  ADDL  F2, 16(SB)          BE 80 16 10
```

Example 1 adds the single-precision numbers in registers F0 and F7 and places the result in register F7.

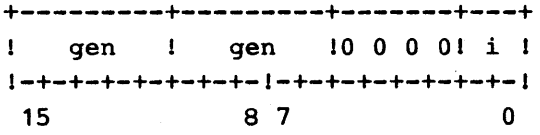
Example 2 adds the double-precision numbers in register pair (F2,F3) and at the address 16(SB) and places the double-precision result at address 16(SB).

These instructions are illustrated below:

	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	F0	40840000 (+4.125)	40840000 (+4.125)
	F7	41D40000 (+26.5)	41F50000 (+30.625)
Ex. 2:	(F2,F3)	41C0200888300000 (+541069584.375)	41C0200888300000 (+541069584.375)
	16(SB)	4114C86300000000 (+340504.75)	41C022A194900000 (+541410089.125)

Add

Syntax: **ADDi** **src,** **dest** ADDB
gen gen ADDW
read.i rmw.i ADDD



The ADDi instruction adds the src and dest operands and places the sum in the dest operand location.

Flags Affected: C is set on a carry from addition, cleared if no carry.
F is set on an overflow from addition, cleared if no overflow.
Integer carry and overflow conditions are defined in Section 3.1

Traps: None.

Examples:

```
1.  ADDB  R0, R1                    40 00
2.  ADDD  4(SB), -4(FP)             03 D6 04 7C
```

Example 1 adds the low-order byte of register R0 to the low-order byte of register R1 and places the result in the low-order byte of register R1. The remaining bytes of R1 are not affected.

Example 2 adds double-words. 4(SB) and -4(FP) specify the operand addresses. The instruction places the double-word sum in memory at the address specified by -4(FP).

These instructions are illustrated below:

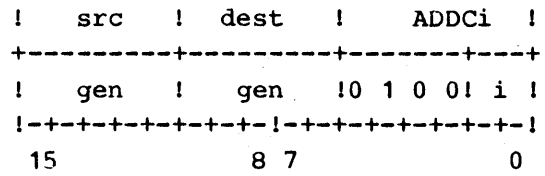
	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	R0	AAAAAA9F (-97)	AAAAAA9F (-97)
	R1	BBBBBB62 (+98)	BBBBBB01 (+1)
	UPSR	nzfxxltc	nz0xxlt1
Ex. 2:	4(SB)	20401110 (+541069584)	20401110 (+541069584)
	-4(FP)	0334A001 (+53780481)	2374B111 (+594850065)
	UPSR	nzfxxltc	nz0xxlt0

Add with Carry

Syntax: **ADDCi** **src,** **dest** ADDCB

gen gen ADDCW

read.i rmw.i ADDCD



The ADDCi instruction adds the src operand, dest operand, and the C flag and places the sum in the dest operand location.

Flags Affected: C is set on a carry from addition, cleared if no carry.
F is set on an overflow from addition, cleared if no overflow.
Integer carry and overflow conditions are defined in Section 3.1.

Traps: None.

Examples:

```
1.  ADDCB   32, R0              10 A0 20
2.  ADDCD   8(SB), R0           13 D0 08
```

Example 1 adds 32, the low-order byte of register R0, and the C flag contents and places the result in the low-order byte of register R0. The remaining bytes of register R0 are unaffected.

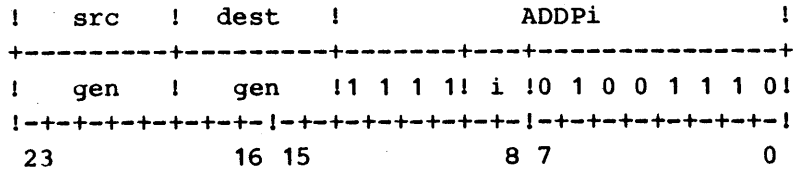
Example 2 adds the double-word at the address specified by 8(SB), the contents of the register R0, and the contents of the C flag and places the result in register R0.

In the following illustration, the C flag is assumed to be 1.

	Operands	Operand Values: Hex (Dec) Before	After
Ex. 1:	32 (immediate)	20 (+32)	--
	R0	AAAAAA0F (+15)	AAAAAA30 (+48)
	UPSR	nzfxxlt1	nz0xxlt0
Ex. 2:	8(SB)	FFFFFFFF (-1)	FFFFFFFF (-1)
	R0	00000030 (+48)	00000030 (+48)
	UPSR	nzfxxlt1	nz0xxlt1

Add Packed Decimal

Syntax: **ADDPi** **src,** **dest** ADDPB
gen gen ADDPW
read.i rmw.i ADDPD



The ADDPi instruction adds the src operand, dest operand, and the C flag and places the result in the dest operand location as a packed decimal (BCD) integer.

The src and dest operands are interpreted as unsigned packed decimal (BCD) integers. If either operand contains invalid digits, the result is undefined. See Section 3.2 for details of packed decimal arithmetic.

Flags Affected: C is set on a carry, cleared if no carry.
F is cleared.
The packed decimal carry condition is defined in Section 3.2.

Traps: None.

Examples:

```
1.  ADDPD   R0, R1                  4E 7F 00
2.  ADDPB   5(SB), TOS              4E FC D5 05
```

Example 1 adds the packed decimal double-word integers contained in registers R0 and R1 and the C flag and places the result in register R1.

Example 2 adds two byte-long packed decimal integers. The integers are at the addresses specified by 5(SB) and TOS. The instruction places the one-byte result on the top of the stack.

In the following illustrations, the C flag value is assumed to be 0.

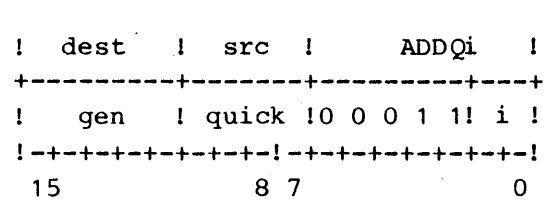
	Operands	Operand Values: Hex * Before	After
Ex. 1:	R0	75308643	75308643
	R1	12345678	87654321
	UPSR	nzfxxlt0	nz0xxlt0
Ex. 2:	5(SB)	99	99
	SP	0000FFDE	0000FFDE
	Stack:		
	0000FFDE	01	00 **
	0000FFDF	AA	AA
	UPSR	nzfxxlt0	nz0xxlt1

* The hexadecimal representation also expresses the decimal interpretation of the value.

** In Example 2, a carry occurs.

Add Quick Integer

Syntax: **ADDQi** **src,** **dest** ADDQB
quick gen ADDQW
rmw.i ADDQD



The ADDQi instruction adds the src and dest operands and places the result in the dest operand location. Before the addition is performed, src is sign-extended to the length of dest.

Flags Affected: C is set on a carry from addition, cleared if no carry.
F is set on an overflow from addition, cleared if no overflow.
Integer carry and overflow conditions are defined in Section 3.1.

Traps: None.

Example:

```
ADDQB  -8, R0                    0C 04
```

The above example adds the quick integer -8 to the low-order byte of register R0. The remaining bytes of R0 are unaffected.

The action of the above instruction is illustrated below.

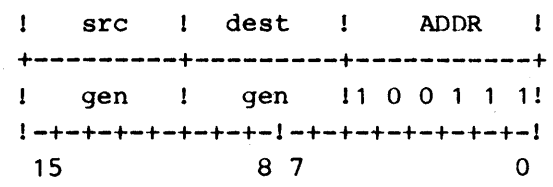
Operands	Operand Values: Hex (Dec) Before	After
-8 (quick)	F8 * (-8)	--
R0	AAAAAA78 (+120)	AAAAAA70 (+112)
UPSR	nzfxxltc	nz0xxlt1

* This shows the internal format of the quick operand after sign-extension to Byte length. The operand is encoded within the instruction as binary 1000.

ADDR

Compute Effective Address

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



The ADDR instruction places the effective address of the src operand into the dest operand location. The src operand itself is not referenced.

NOTE: If the ADDR instruction is used with the External addressing mode for an external procedure, it does not generate the effective address of the procedure entry point, but it can be used instead to copy the procedure's descriptor from the current Link Table into the dest operand location. The ASM16 assembler provides an alternative mnemonic LXPD (Load External Procedure Descriptor, q.v.) for this use.

Flags Affected: None.

Traps: None.

Example:

```
ADDR  4(FP), R0                    27 C0 04
```

This example places the effective address specified as 4(FP) into register R0.

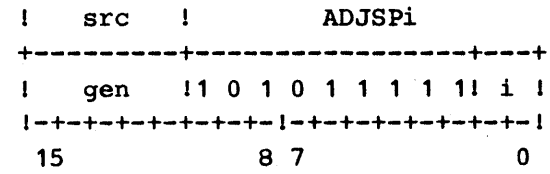
The action of this instruction is illustrated below:

	Operand Values: Hex	
Operands	Before	After
FP	00001000	00001000
R0	AAAAAAAA	00001004 *

* The effective address of 4(FP) is the sum of the contents of the FP register (H'1000) and the displacement 4, as defined for the Frame Memory addressing mode.

Adjust Stack Pointer

Syntax: **ADJSPi** **src** ADJSPB
gen ADJSPW
read.i ADJSPD



The ADJSPi instruction adjusts the value of the current stack pointer by subtracting the src operand from it. This has the effect of lengthening the stack by the number of bytes given in the src operand if positive, and shortening it if src is negative. The S flag in the PSR determines whether the current stack pointer register is SP0 or SP1. Regardless of the length of the src operand, the entire stack pointer is modified. The src operand is interpreted as a signed integer, and is sign-extended to 32 bits before the subtraction is performed.

Flags Affected: None.

Traps: None.

Example:

```
ADJSPD  -4(FP)                    7F C5 7C
```

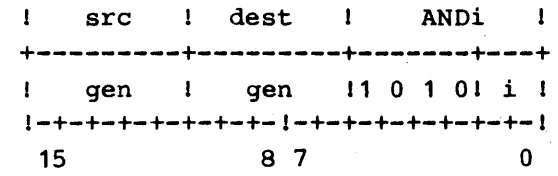
This instruction subtracts the double-word at address -4(FP) from the contents of the current stack pointer, lengthening the stack by that number of bytes.

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

Operands	Operand Values: Hex Before	After
-4(FP)	00000010	00000010
SP1	00001010	00001000

ANDi
And

Syntax: **ANDi** **src**, **dest** ANDB
gen gen ANDW
read.i rmw.i ANDD



The ANDi instruction performs a bit-wise logical AND on the src and dest operands and places the result in the dest operand location.

The instruction ANDs each bit in src with the corresponding dest bit. If two corresponding bits are both "1", the dest bit is set to "1"; otherwise, the dest bit is set to "0".

Flags Affected: None.

Traps: None.

Example:

```
ANDB  R0, R1                          68 00
```

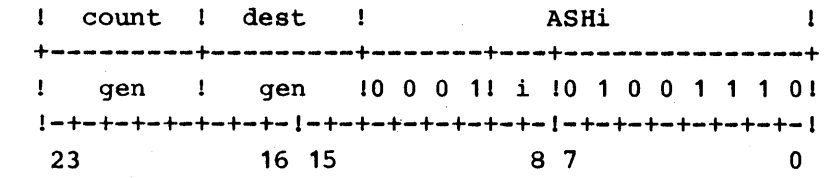
This example ANDs the low-order bytes of registers R0 and R1 and places the result in the low-order byte of R1. The instruction affects only the low-order byte of R1.

The instruction is illustrated below:

Operands	Operand Values: Binary Before	After
R0 (low byte)	10010010	10010010
R1 (low byte)	01110111	00010010

Arithmetic Shift

Syntax: **ASHi** **count,** **dest** ASHB
gen gen ASHW
read.B rmw.i ASHD



The ASHi instruction performs an arithmetic 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 ASHB form, -15 to +15 for the ASHW form, and -31 to +31 for the ASHD form. A positive count specifies a left shift; a negative count specifies a right shift. In an arithmetic left shift, high-order bits (including the sign bit) shifted out of dest are lost, and low-order bit positions emptied by the shift are zero-filled. In an arithmetic right shift, low-order bits shifted out of dest are lost, and all high-order bit positions emptied by the shift are filled from the original sign bit of dest.

The count and dest operands are interpreted as signed integers.

Flags Affected: None.

Traps: None.

Examples:

```
1.  ASHB  2, 16(SB)          4E 84 A6 02 10
2.  ASHB  TOS, 16(SB)        4E 84 BE 10
```

Example 1 shifts the byte specified by 16(SB) two bit positions to the left.

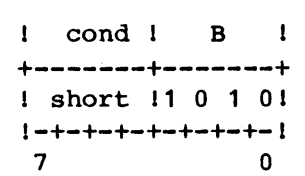
Example 2 pops a byte from the top of the currently-selected stack. Based on this value, it shifts the byte specified by 16(SB) accordingly.

These instructions are illustrated below:

	Operands	Operand Values: Binary (Dec) Before	After
Ex. 1:	2 (immediate)	00000010 (+2)	--
	16(SB)	00011111 (+31)	01111100 (+124)
Ex. 2:	Stack:		
	(48000)	11111110 (-2)	xxxxxxxx
	(48001)	10101010	10101010
	16(SB)	11111000 (-8)	11111110 (-2)
	SP	(48000)	(48001)

Conditional Branch

Syntax: **Bcond** **dest**
disp



The Bcond instruction branches to the location specified as dest if the condition specified by cond is true. If the condition is false, execution continues with the next sequential instruction.

Cond is a two character condition name that specifies the state of a flag or flags in the PSR. If the flag(s) have the specified state, the condition is true; otherwise, the condition is false.

The Conditional Branch instruction may specify the following conditions:

Condition	Condition Name	True State	Short Field
Equal	EQ	Z flag set	0000
Not Equal	NE	Z flag clear	0001
Carry Set	CS	C flag set	0010
Carry Clear	CC	C flag clear	0011
Higher	HI	L flag set	0100
Lower or Same	LS	L flag clear	0101
Greater Than	GT	N flag set	0110
Less Than or Equal	LE	N flag clear	0111
Flag Set	FS	F flag set	1000
Flag Clear	FC	F flag clear	1001
Lower	LO	Z and L flags clear	1010
Higher or Same	HS	Z or L flag set	1011
Less Than	LT	Z and N flags clear	1100
Greater Than or Equal	GE	Z or N flag set	1101

The condition name is appended to the instruction mnemonic as illustrated in the following examples. The name is translated at assembly time to the corresponding 4-bit Short field of the basic instruction.

The interpretation of condition codes is such that the instruction sequence

```
CMPB    A,B
BGT     ERROR
```

will cause a branch if operand A is greater than operand B in the CMPB instruction.

In the machine instruction, dest is specified as a displacement from the current contents of the Program Counter; i.e., from the address of the first byte of this instruction (see Section 4.2.3 for displacement formats). Using the ASM16 assembler, this displacement may be given explicitly in the form *+disp or *-disp, or dest may be specified as a statement label or any addressing expression which evaluates to an address accessible via Program Counter Relative addressing. See the applicable assembler manual for further information.

Flags Affected: None.

Traps: None.

Examples:

1.	BLO LOOP	AA BF 66
2.	BNE *+10	1A 0A

Example 1 passes execution control to the instruction labeled LOOP if the Z and L flags in the PSR are 0.

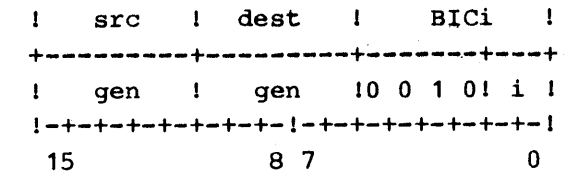
Example 2 passes execution control to a non-sequential instruction if the Z flag is 0. The instruction passes execution control by adding 10 to the PC register.

In the following illustrations, the Z and L flags are assumed to be zero. LOOP is assumed to be the label of a statement beginning at address 9000 Hex.

	Operand	Operand Values: Hex (Dec) Before	After
Ex. 1:	PC	0000909A (37018)	00009000 (36864)
	LOOP (disp)	BF 66 (-154)	--
	UPSR	n0fxx0tc	n0fxx0tc
Ex. 2:	PC	00009FF0 (40944)	00009FFA (40954)
	*+10 (disp)	0A (+10)	--
	UPSR	n0fxx0tc	n0fxx0tc

Bit Clear

Syntax: **BICi** **src,** **dest** BICB
gen gen BICW
read.i rmw.i BICD



The BICi instruction clears (sets to 0) the bits in the dest operand that correspond to the "1" bits in the src operand.

Flags Affected: None.

Traps: None.

Example:

```
BICB  R0, 3(SB)                    88 06 03
```

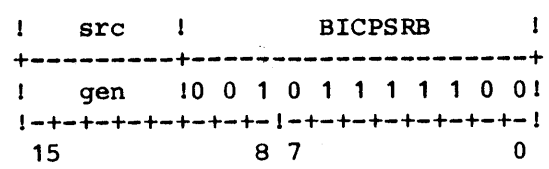
This example clears the bits, in the byte at address 3(SB), corresponding to the "1" bits in the low-order byte of register R0.

The instruction is illustrated below:

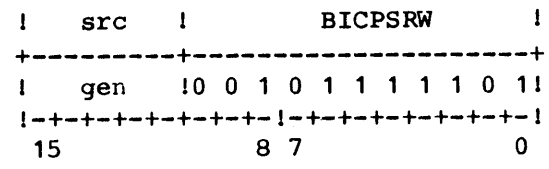
Operands	Operand Values (Binary) Before	After
R0 (low byte)	10011001	10011001
3(SB)	11110000	01100000

BICPSRB
BICPSRW
Bit Clear in PSR

Syntax: **BICPSRB** **src**
gen
read.B



Syntax: **BICPSRW** **src**
gen
read.W



The Bit Clear in PSR instructions clear (set to 0) the bits in the PSR corresponding to the "1" bits in the src operand. The BICPSRB instruction affects only the low-order byte of the PSR; the BICPSRW instruction affects the entire PSR.

Flags Affected: Flags specified by src "1" bits are cleared.

Traps: Illegal Operation Trap (ILL) is activated if a BICPSRW instruction is attempted while the PSR U flag is set.

Example:

```
BICPSRB  B'10100010                7C A1 A2
```

This instruction clears bits 1, 5 and 7 in the low-order byte of the PSR. These are the T, F and N flags, respectively.

The instruction is illustrated below:

Operands	Operand Values (Binary) Before	After
B'10100010 (immediate)	10100010	--
PSR	xxxxipsu/nzfxxltc	xxxxipsu/0z0xx10c

Bit Set in PSR

Syntax: **BISPSRB** **src**
gen
read.B

```
!   src   !        BISPSRB          !
+---------+-------------------------+
!   gen   !0 1 1 0 1 1 1 1 1 0 0 0!
!-+-+-+-+-+-+-+-!-+-+-+-+-+-+-+-!
 15             8 7               0
```

Syntax: **BISPSRW** **src**
gen
read.W

```
!   src   !        BISPSRW          !
+---------+-------------------------+
!   gen   !0 0 1 0 1 1 1 1 1 0 1 0!
!-+-+-+-+-+-+-+-!-+-+-+-+-+-+-+-!
 15             8 7               0
```

The BISPSRB and BISPSRW instructions set the bits in the PSR corresponding to the "1" bits in the src operand.

Flags Affected: Flags specified by src "1" bits are set.

Traps: Illegal Operation Trap (ILL) is activated if a BISPSRW instruction is attempted while the PSR U flag is 1.

Example:

```
BISPSRB  B'10100010                7C A3 A2
```

This instruction sets bits 1, 5 and 7 in the low-order byte of the PSR. These are the T, F and N flags, respectively.

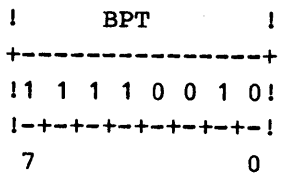
The instruction is illustrated below:

Operands	Operand Values (Binary) Before	After
B'10100010 (immediate)	10100010	--
PSR	xxxxipsu/nzfxxltc	xxxxipsu/1z1xxl1c

BPT

Breakpoint Trap

Syntax: **BPT**



The BPT instruction activates the Breakpoint Trap (BPT). See "Exceptions", Chapter 6. The return address pushed on the Interrupt Stack is the address of the BPT instruction itself.

Flags Affected: None.

Traps: Breakpoint Trap (BPT) is activated.

Example:

```
BPT                              F2
```

Unconditional Branch

Syntax: **BR** **dest**
disp

```
!       BR        !
+-----------------+
!1 1 1 0 1 0 1 0!
!-+-+-+-+-+-+-+-!
 7               0
```

The BR instruction branches to the location specified as dest.

In the machine instruction, dest is specified as a displacement from the current contents of the Program Counter; i.e., from the address of the first byte of this instruction. Using the ASM16 assembler, this displacement may be given explicitly in the form *+disp or *-disp, or dest may be specified as a statement label or any addressing expression which evaluates to an address accessible via Program Counter Relative addressing. See the applicable assembler manual for further information.

Flags Affected: None.

Traps: None.

Examples:

1.	BR ERROR	EA BF 66
2.	BR *+10	EA 0A

Example 1 passes execution control to the instruction labeled ERROR.

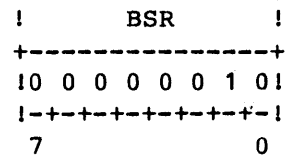
Example 2 passes execution control to a non-sequential instruction by adding 10 to the PC register.

In the following illustration, ERROR is assumed to be the label of a statement beginning at address 9000 Hex.

	Operand	Operand Values: Hex (Dec) Before	After
Ex. 1:	PC	0000909A (37018)	00009000 (36864)
	ERROR (disp)	BF 66 (-154)	--
Ex. 2:	PC	00009FF0 (40944)	00009FFA (40954)
	*+10 (disp)	0A (+10)	--

BSR
Branch To Subroutine

Syntax: **BSR** **dest**
disp



The BSR instruction calls the local procedure at the address specified as dest. It does so by pushing the address of the next sequential instruction onto the currently-selected stack and branching.

In the machine instruction, dest is specified as a displacement from the current contents of the Program Counter; i.e., from the address of the first byte of this instruction. Using the ASM16 assembler, this displacement may be given explicitly in the form *+disp or *-disp, or dest may be specified as a statement label or any addressing expression which evaluates to an address accessible via Program Counter Relative addressing. See the applicable assembler manual for further information.

Flags Affected: None.

Traps: None.

Example:

```
     BSR CALC                              02 10
```

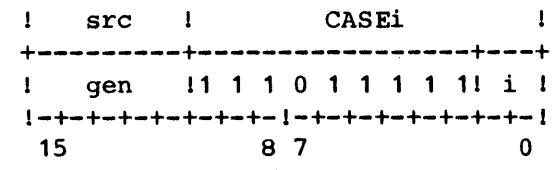
This example causes a program to branch to the local procedure labeled CALC after saving the address of the next sequential instruction on the stack.

The action of the above instruction is illustrated below. The statement labeled CALC is assumed to be at address 9010 Hex, 16 bytes forward from the first byte of this instruction. Since the displacement field is one byte long, the total length of this instruction is two bytes. The return address is therefore the original PC contents plus two.

Operand	Operand Values: Hex (Dec) Before	After
PC	00009000	00009010
CALC (disp)	10 (+16)	--
SP	0000FFD4	0000FFD0
Stack:		
0000FFD0	xxxxxxxx	00009002
0000FFD4	AAAAAAAA	AAAAAAAA

Case Branch

Syntax: **CASEi** **src** CASEB
gen CASEW
read.i CASED



The CASEi instruction branches to a non-sequential instruction by adding the src operand to the PC register. The src operand is interpreted as a signed integer, and is sign-extended to 32 bits before the addition is performed.

A Case Branch instruction, using Scaled Indexing and a table of branch offsets, may be used to implement a multi-way branch. See example below.

Flags Affected: None.

Traps: None.

Example:

```
            CASEB  TABLE[R7:B]      7C E7 DF 04
TABLE:      ; (starts here)
```

This example branches to a non-sequential instruction by adding the byte at the address specified by TABLE[R7:B] to the PC register. The entire contents of register R7 determine the location of the operand to be added.

This instruction is illustrated below:

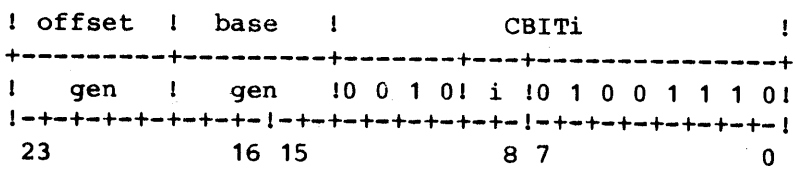
Operand	Operand Values: Hex Before	After
PC	00009000	0000906A
R7	00000005	00000005
TABLE 00009004 *	0A 1A 3A 5A 7A 6A 4A	0A 1A 3A 5A 7A 6A 4A
TABLE[R7:B]	6A	6A

* Address 9004 (Hex) marks the beginning of a table of branch offsets. In this example, the table is located directly after the CASEB instruction, and is accessed via the Program Memory addressing mode, with a displacement of 4. Since register R7 contains 5, the effective address of TABLE[R7:B] is 9009 (Hex). This means the sixth branch offset (6A) is selected to be added to the PC register.

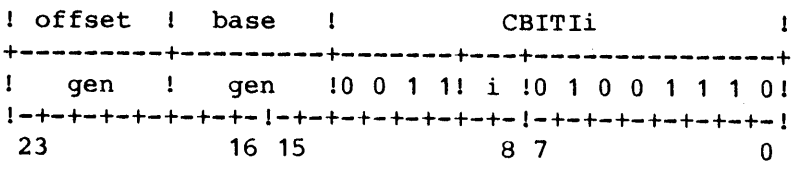
CBITi
CBITIi
Clear Bit, Clear Bit Interlocked

Syntax: **CBITi** **offset,** **base**
gen gen
read.i regaddr

CBITB CBITIB
CBITW CBITIW
CBITD CBITID



Syntax: **CBITIi** **offset,** **base**
gen gen
read.i regaddr



The CBITi and CBITIi instructions clear (set to 0) the register or memory bit specified by base and offset after copying the bit value to the F flag in the PSR.

The CBITIB, CBITIW, and CBITID instructions, in addition, activate the Interlocked Operation output pin on the CPU, which may be used in multi-processor systems to interlock accesses to semaphore bits. See the applicable CPU data sheet for further details.

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.

Clear Bit (continued)

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

Traps: None.

Example:

```
CBITW  R0, 0(R1)                    4E 49 02 00
```

This example clears a bit in memory after copying the bit value to the F flag. For designating the location of the target bit, the low-order word of register R0 supplies the bit offset, and 0(R1) is specified as the base address.

In the following illustration, the target bit is assumed to be 1 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 0(R1)	00001000 (+4096)	--
00001009 * (+4105)	10 [00010000]	00 [00000000]
UPSR	nzfxxltc	nz1xxltc

* The address 1009 (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:

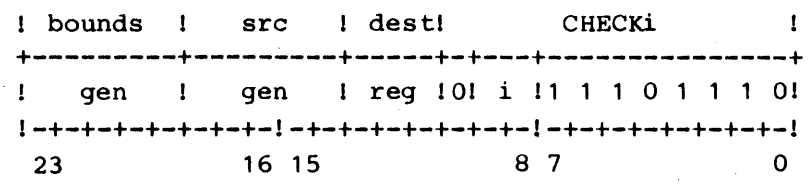
```
base address + (offset DIV 8)
   4096      +       9
   4105 , or 1009 (Hex)  .
```

The bit number within this byte is calculated as:

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

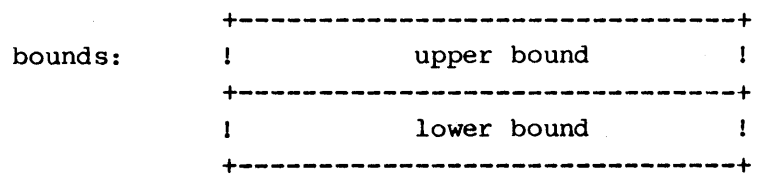
CHECKi
Bounds Check

Syntax: **CHECKi** **dest,** **bounds,** **src** CHECKB
reg gen gen CHECKW
addr read.i CHECKD



The CHECKi instruction compares the src operand against an upper and lower bound from the bounds operand, determining whether it is within those bounds. The instruction then subtracts the lower bound from src, placing the result as a 32-bit value into the general-purpose register specified as dest. This "zero-adjusted" result is usable directly as either an index into a one-dimensional array (within an addressing mode using Scaled Indexing) or as an input value to the INDEX instruction for generating an index into a multi-dimensional array. See Section 3.9 for details of array access.

The bounds operand contains two values -- an upper bound followed by a lower bound -- as shown below:



The upper and lower bounds each have the same length as the src operand. Thus, the entire bounds operand is twice the length of the src operand.

If src is greater than the upper bound or less than the lower bound, it is "out of bounds" and the F flag is set to 1. If src is within the upper and lower bounds, the F flag is cleared.

The instruction places the zero-adjusted result into the dest register. The zero-adjusted value is computed as "src - lower bound". The result is zero-extended to 32 bits. If src is out of bounds, the result placed in dest is undefined.

The src and bounds operands are interpreted as signed integers. The result placed in the dest register is a 32-bit unsigned integer.

Flags Affected: F is set if src is out of bounds, cleared otherwise.

Traps: None.

Bounds Check (continued)

Example:

```
CHECKB  R0, 4(SB), R2              EE 80 D0 04
```

This example compares the low-order byte of register R2 with each of the two one-byte bounds at the address specified by 4(SB). The instruction sets or clears the F flag to indicate the comparison result and then subtracts the lower bound from the low-order byte of R2, placing the result in register R0.

The instruction is illustrated below. An array index in the low-order byte of R2 is being checked against its range of [1..10] and then "zero-adjusted" to its corresponding value for the range [0..9]. The result is being placed into R0 for use in an addressing mode using Scaled Indexing.

Operands	Operand Values: Hex (Dec) Before	After
R0	AAAAAAAA	00000002 * (+2)
4(SB)	0A 01 (+10,+1)	0A 01 (+10,+1)
R2	AAAAAA03 (+3)	AAAAAA03 (+3)
UPSR	nzfxxltc	nz0xxltc

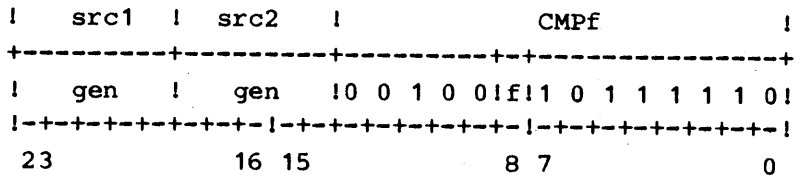
* The result in R0 represents the result of adjusting the value 3 from a range of [1..10] to the zero-based range of [0..9]. The corresponding adjusted value is 2.

CMPf

Compare Floating

Syntax: **CMPf** **src1,** **src2**
gen gen
read.f read.f

CMPF
CMPL



The CMPf instruction compares the src1 and src2 operands and sets the Z and N flags to indicate comparison results. Positive and negative zero are equal.

Flags Affected: Z is set if src1 equals src2, cleared otherwise.
N is set if src1 is greater than src2, cleared otherwise.
L is cleared always.

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.

Example:

```
CMPF  F0, F2                    BE 89 00
```

This example compares the single-precision numbers in registers F0 and F2.

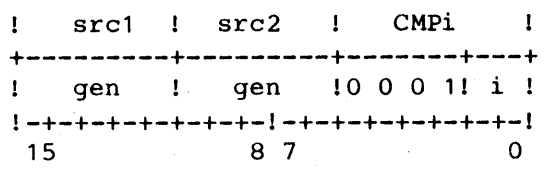
The instruction is illustrated below:

Operands	Operand Values: Hex (Dec) Before	After
F0	42250000 (+41.25)	42250000 (+41.25)
F2	40A00000 (+5.0)	40A00000 (+5.0)
UPSR	nzfxxltc	10fxx0tc

Compare

Syntax: **CMPi** **src1,** **src2**
gen gen
read.i read.i

CMPB
CMPW



The CMPi instruction compares the src1 and src2 operands and sets the Z, N, and L flags in the PSR to indicate the comparison result. The N flag indicates the result of a signed integer comparison; the L flag indicates the result of an unsigned integer comparison. Both types of comparison are performed.

Flags Affected: Z is set if src1 is equal to src2, cleared otherwise.
N is set if src1 is greater than src2 (signed comparison), cleared otherwise.
L is set if src1 is greater than src2 (unsigned comparison), cleared otherwise.

Traps: None.

Example:

```
CMPB  7(SB), 4(R0)                    04 D2 07 04
```

This example compares byte operands. 7(SB) and 4(R0) specify the operand addresses.

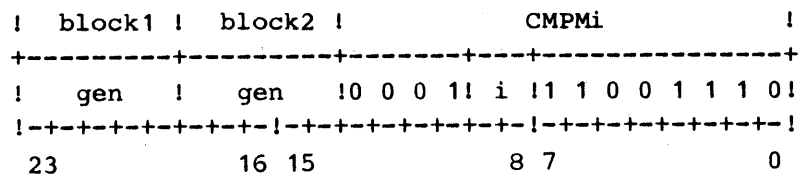
In the following illustration, operand values before instruction execution are assumed; Z, N and L flag values are unknown.

Operands	Operand Values: Hex (Dec) Before	After
7(SB)	FF (signed: -1) (unsigned: +255)	FF (signed: -1) (unsigned: +255)
4(R0)	7F (signed: +127) (unsigned: +127)	7F (signed: +127) (unsigned: +127)
UPSR	nzfxxltc	00fxx1tc

CMPMi

Compare Multiple

Syntax: **CMPMi** **block1,** **block2,** **length** CMPMB
gen gen disp CMPMW
addr addr CMPMD



The CMPMi instruction compares the contents of block1 and block2 and sets the Z, N, and L flags to indicate the comparison result. The blocks are comprised of integers of length i. The number of integers is specified by length.

The instruction compares two integers (one from each block) at a time. If the current integers are equal, the instruction continues with the next two integers; otherwise, the instruction sets the PSR flags and terminates.

The N flag indicates the result of signed integer comparison. The L flag indicates the result of unsigned integer comparison. Both types of comparison are performed.

In assembly language, the length operand is specified as the number of integers in each block. In the machine instruction, however, the length operand is encoded according to the formula

(num - 1) * i

where num is the number of integers in each block, and i is the number of bytes per integer.

A block may not be greater than 16 bytes in length.

Flags Affected: Z is set if block1 and block2 are equal for their entire length; cleared otherwise.

N is set if, in the first unequal pair of integers, the block1 integer is greater than the block2 integer (signed comparison); cleared otherwise.

L is set if, in the first unequal pair of integers, the block1 integer is greater than the block2 integer (unsigned comparison); cleared otherwise.

Traps: None.

Compare Multiple (continued)

Example:

```
CMPMW  10(R0), 16(R1), 4          CE 45 42 0A 10 06
```

This instruction compares four word-long integers from the block starting at the address specified by 10(R0) to the block starting at the address specified by 16(R1).

The instruction is illustrated below:

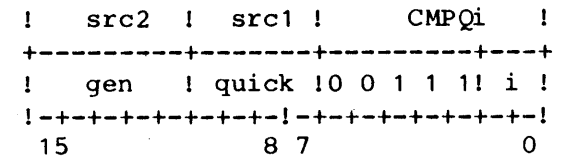
Operands	Operand Values: Hex (Signed) [Unsigned] Before	After
R0	00002000	00002000
R1	0000F000	0000F000
0000200A *	1FBE	1FBE
	10A9	10A9
	8729 (-30935)	8729 (-30935)
	[+34601]	[+34601]
	6511	6511
0000F010 **	1FBE	1FBE
	10A9	10A9
	0839 (+2105)	0839 (+2105)
	[+2105]	[+2105]
	6511	6511
UPSR	nzfxxltc	00fxx1tc

* The address of the first block as specified by 10(R0).
** The address of the second block as specified by 16(R1).

Compare Quick Integer

Syntax: **CMPQi** **src1,** **src2**
quick gen
read.i

CMPQB
CMPQW



The CMPQi instruction compares the src1 and src2 operands and sets the Z, N, and L flags to indicate the comparison result. Before the comparison, src1 is sign-extended to the length of src2. Leading "0"s are supplied for a positive src operand value; leading "1"s for a negative value.

The N flag indicates the result of signed integer comparison. The L flag indicates the result of unsigned integer comparison. Both types of comparison are performed.

Flags Affected: Z is set if src1 is equal to src2, cleared otherwise.
N is set if src1 is greater than src2 (signed comparison), cleared otherwise.
L is set if src1 is greater than src2 (unsigned comparison), cleared otherwise.

Traps: None.

Example:

```
CMPQB  -8, R0                    1C 04
```

This example compares the quick integer -8 with the low-order byte of register R0.

This instruction is illustrated below:

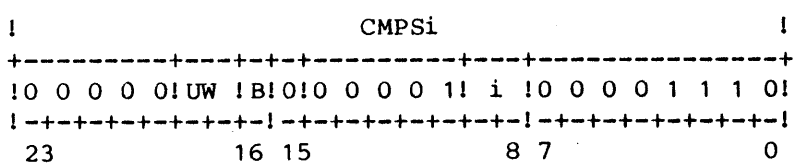
Operands	Operand Values: Hex (Signed) [Unsigned] Before	After
-8 (quick)	F8 * (-8) [+248]	--
R0	AAAAAA00 (0) [0]	AAAAAA00 (0) [0]
UPSR	nzfxxltc	00fxx1tc

* This shows the internal format of the quick operand after sign-extension to Byte length. The operand is encoded within the instruction as binary 1000.

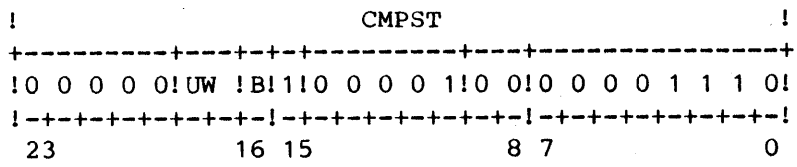
Compare Strings

Syntax: **CMPSi options**

CMPSB
CMPSW
CMPSD
CMPST



Syntax: **CMPST options**



Operands of the CMPSi and CMPST instructions are specified in General Purpose Registers:

R0 - Number of string elements to be processed.
R1 - Address of current String 1 element.
R2 - Address of current String 2 element.
R3 - Address of translation table (CMPST form only).
R4 - Match value (with Until Match or While Match option only).

The CMPSi instruction compares corresponding integer elements from String 1 (address in R1) and String 2 (address in R2) and sets the Z, N and L flags to indicate the comparison results (see "Flags Affected" below). If the current two elements are equal, the instruction compares the next two elements; otherwise, it terminates. After each comparison, the instruction sets register R0 to the number of elements remaining to be compared and sets registers R1 and R2 to the addresses of the next elements to be compared. See Section 3.7 for the exact sequences performed by String instructions.

The N flag indicates the result of signed integer comparison. The L flag indicates the result of unsigned integer comparison. Both types of comparison are performed.

The CMPST instruction compares one-byte elements in String 1, after translation, to one-byte elements in String 2. The translated value to be compared is found by adding the current element from the first string as an unsigned integer to the translation table address found in register R3. The instruction compares elements, sets flags, and sets registers as described above. See Section 3.7 for details of string translation.

Options may be specified by listing the letters B (Backward), U (Until Match) and W (While Match) as operands. The U and W options are mutually exclusive. See Section 3.7 for details of the options available in String instructions.

In the machine instruction, the options are encoded in the B and UW fields as follows:

B field =	0	Forward direction.
	1	Backward direction.
UW field =	00	Neither Until Match nor While Match.
	01	While Match.
	10	(reserved)
	11	Until Match.

String instructions are interruptible. See Section 3.7.

Flags Affected: Z, N and L are affected, as given below.
F is set if the U or W option is specified and the corresponding Until/While condition is met, otherwise it is cleared.

Because of the variety of termination conditions possible in a CMPS instruction, the following sequence is recommended to interpret the flag settings:

1. If the U or W option is specified, then check the F flag. If it is set, then the CMPS instruction has terminated because of the Until Match or While Match test, and the other flag settings are Z=1, N=0, L=0. Register R1 holds the address of the String 1 element which caused termination, and register R2 holds the address of the corresponding element in String 2. Register R0 contains the number of elements left to be processed, including the element which terminated the instruction.

2. If F=0, check the Z flag. If it is set, then the CMPS instruction has terminated because the limit count in R0 has been decremented to zero, and the strings are equal up to that point. Registers R1 and R2 hold the addresses of the next (unprocessed) string elements, and the remaining flag settings are N=0, L=0.

3. If neither the F or Z bit is set (above), then the CMPS instruction has terminated because the strings are unequal. Registers R1 and R2 hold the addresses of the first two string elements that are unequal, and the N and L flags show their relation. Register R0 holds the number of remaining elements, including the element at which the instruction has stopped. If the N bit is set, the element from String 1

Compare Strings (continued)

(indicated by R1) is greater than the element from String 2 (indicated by R2), where both are interpreted as signed integers. If the L bit is set, the element from String 1 is greater than the element from String 2, where both are interpreted as unsigned integers.

Traps: None.

Example:

```
CMPSB                                   0E 04 00
```

This example compares one-byte elements from two strings, until either an unequal pair is found or the limit count in R0 decrements to zero.

The action of the above instruction on two unequal strings is illustrated below. The underlined string elements show the point at which the instruction terminates.

Operands	Operands Values: Hex (Dec) Before	After
R0	00000020 (+32)	0000000F (+15)
R1	00002000	00002011
R2	0000F000	0000F011
UPSR	nzfxxltc	100xx1tc

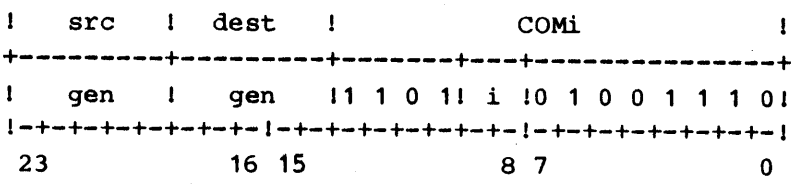
Starting Addresses	String Contents
2000	1E 04 05 1C 0A 14 0C 0B 09 07 1F 0F 17 01 00 11
	1F 1E 1A 09 01 12 14 0E 1E 0A 00 03 09 06 16 18
F000	1E 04 05 1C 0A 14 0C 0B 09 07 1F 0F 17 01 00 11
	1F 1D 1A 09 01 12 14 0E 1E 0A 00 03 09 06 16 18

COMi
Complement

Syntax: **COMi** **src,** **dest**

	gen	gen
	read.i	write.i

COMB
COMW
COMD



The COMi instruction places the one's complement of the src operand in the dest operand location.

The one's complement is the logical NOT operation performed on each bit of src. If the src operand bit is "0", then the corresponding dest operand bit is set to "1". Otherwise, the dest bit is set to "0".

Flags Affected: None.

Traps: None.

Example:

```
COMB  R0, -4(FP)                    4E 34 06 7C
```

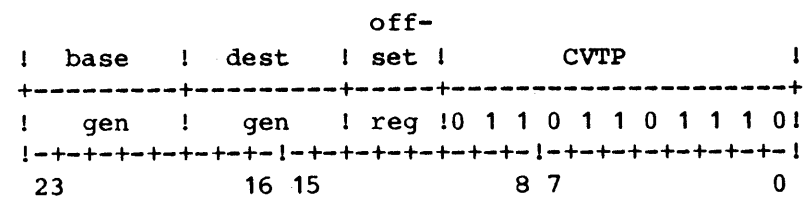
This example places the one's complement of the low-order byte of register R0 in the byte at the address specified by -4(FP).

The instruction is illustrated below:

Operands	Operand Values: Binary Before	After
R0 (low-order)	10101010	10101010
-4(FP)	00000000	01010101

Convert to Bit Pointer

Syntax: **CVTP** **offset,** **base,** **dest**
reg gen gen
addr write.D



The CVTP instruction places in the dest operand location the absolute bit address of the memory bit specified by base and offset. See Section 3.5 for the use of a base and offset in specifying a bit position.

The bit address specifies the number of bits from the first bit in the memory space (bit 0 of the byte at address 0) to the specified bit. The bit address is computed as

$$8 * EA(base) + offset \quad ,$$

where EA(base) is the effective address calculated for base, and offset is a signed byte, word or double-word as given by the operation length.

Flags Affected: None.

Traps: None.

Example:

```
CVTP  R0, 32(SB), R2              6E 83 D0 20
```

This example computes the absolute bit address of the memory bit specified by register R0 and the address 32(SB). The instruction places the resulting bit address into register R2 as a double-word.

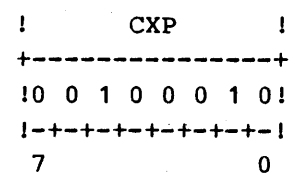
The instruction is illustrated below:

Operands	Operand Values: Hex Before	After
R0	00001234	00001234
SB	00000FE0	00000FE0
base address 32(SB)	00001000	--------
R2	AAAAAAAA	00009234

CXP

Call External Procedure

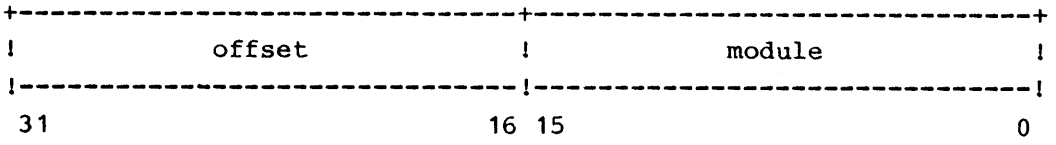
Syntax: **CXP** **index**
disp



The CXP instruction calls a procedure which is outside the current module (an "external" procedure).

The entry point of the external procedure is specified by an external procedure descriptor, which is located in the Link Table (Section 2.7.3) of the current module. The index operand gives the Link Table entry number of the descriptor.

The descriptor is a 32-bit value in the following format:



The descriptor address is the sum of index, multiplied by 4, and the contents of the double-word at memory address MOD+4 (the Link Base pointer, Section 2.7.2), where MOD is the contents of the MOD register.

Once the descriptor has been located, 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 currently-selected 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.

Call External Procedure (continued)

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.

In the machine instruction, index is encoded as a displacement field appended to the basic instruction. In assembly language, index is specified as the name of the external procedure or in the form of an External addressing mode expression.

Flags Affected: None.

Traps: None.

Examples:

```
1.  CXP  OUTSIDE          22 00
2.  CXP  EXT(1)           22 01
```

Example 1 calls the external procedure named OUTSIDE.

Example 2 calls the external procedure whose descriptor is located in the second entry of the current Link Table (entry number 1).

The action of the instruction in example 2 is illustrated below:

Operands	Operand Values: Hex Before	After
1 (index)	01	--
90A4 * (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	

* 90A4 is the descriptor address. It is computed as the sum of the index 1 in the expression EXT(1), scaled by 4, and the Link Table address. The Link Table address is from address 14 (Hex) in the Module Table.

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