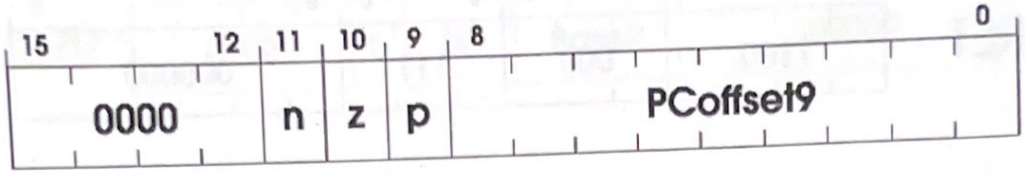
BR - Conditional Branch

Assembler Formats

```
BRn LABEL
BRp LABEL
BRz LABEL
BRzp LABEL
BRnp LABEL
BRnz LABEL
BRnzp LABEL
BR  LABEL
```

Encoding



Operation

```
if ((n AND N) OR (z AND Z) OR (p AND P)) {
      PC = PC‡ + SEXT(PCoffset9);
}
```

Description

The condition codes specified by the state of bits [11:9] are tested. If bit [11] is set, N is tested; if bit [11] is clear, N is not tested. If bit [10] is set, Z is tested, etc. If any of the condition codes tested is set, the program branches to the location specified by adding the sign-extended PCoffset9 field to the incremented PC.

Examples

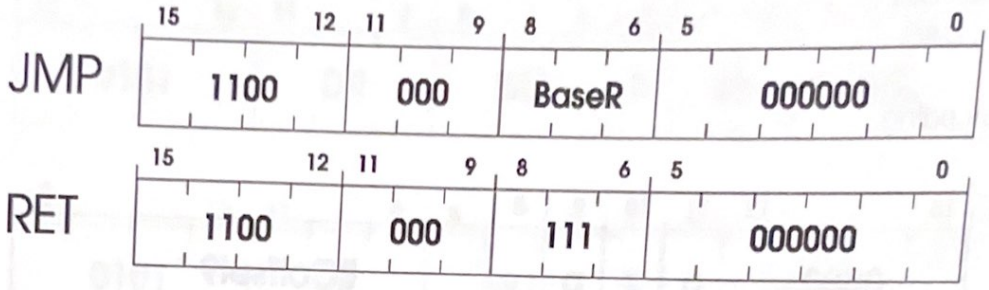
```
BRz LOOP    ; Branch to LOOP if the last result was zero
BR  NEXT    ; Unconditionally branch to NEXT.
```

JMP – Jump, RET - Return from Subroutine

Assembler Formats

```
JMP BaseR
RET
```

Encoding



Operation

```
PC = BaseR;
```

Description

The program unconditionally jumps to the location specified by the contents of the base register. Bits [8:6] identify the base register.

Examples

JMP R2;	PC ← R2
RET;	PC ← R7

Note

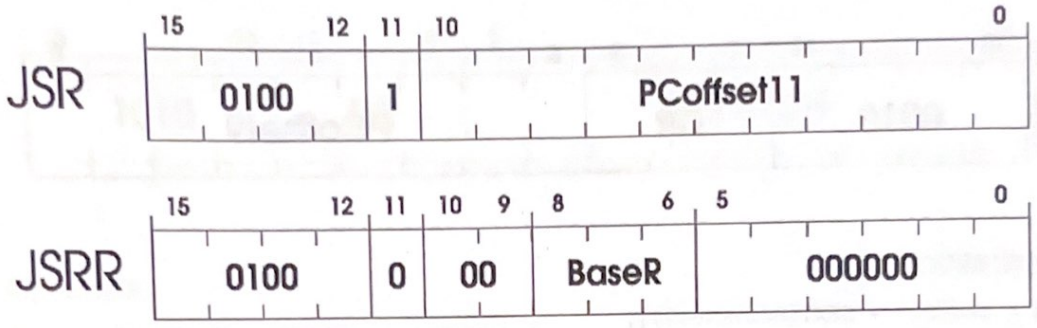
The RET instruction is a special case of the JMP instruction. The PC is loaded with the contents of R7, which contains the linkage back to the instruction following the subroutine call instruction.

JSRR

Assembler Formats

```
JSR LABEL
JSRR BaseR
```

Encoding



Operation

```
R7 = PC;†
if (bit[11] == 0)
      PC = BaseR;
else
      PC = PC + SEXT(PCoffset11);
```

Description

First, the incremented PC is saved in R7. This is the linkage back to the calling routine. Then the PC is loaded with the address of the first instruction of the subroutine, causing an unconditional jump to that address. The address of the subroutine is obtained from the base register (if bit [11] is 0), or the address is computed by sign-extending bits [10:0] and adding this value to the incremented PC (if bit [11] is 1).

Examples

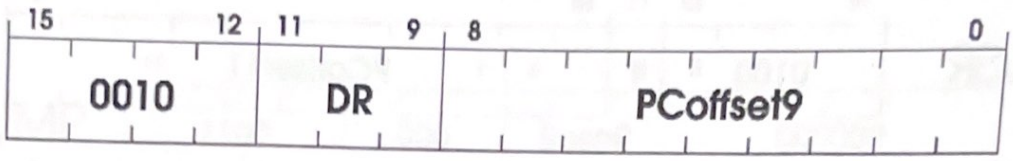
```
JSR QUEUE ; Put the address of the instruction following JSR into R7;
          ; Jump to QUEUE.
JSRR R3   ; Put the address following JSRR into R7; Jump to the
          ; address contained in R3.
```

LD - Load

Assembler Format

```
LD DR, LABEL
```

Encoding



Operation

```
DR = mem[PC† + SEXT(PCoffset9)];
setcc();
```

Description

An address is computed by sign-extending bits [8:0] to 16 bits and adding this value to the incremented PC. The contents of memory at this address are loaded into DR. The condition codes are set, based on whether the value loaded is negative, zero, or positive.

Example

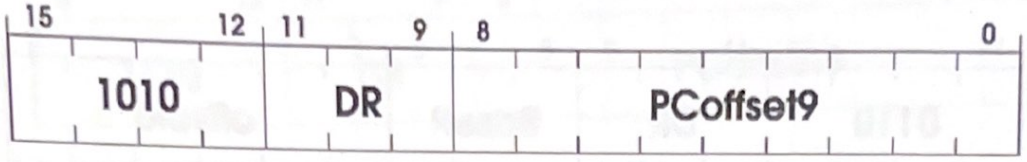
```
LD R4, VALUE        ; R4 ← mem[VALUE]
```

LDI - Load Indirect

Assembler Format

```
LDI DR, LABEL
```

Encoding



Operation

```
DR = mem[mem[PC† + SEXT(PCoffset9)]];
setcc();
```

Description

An address is computed by sign-extending bits [8:0] to 16 bits and adding this value to the incremented PC. What is stored in memory at this address is the address of the data to be loaded into DR. The condition codes are set, based on whether the value loaded is negative, zero, or positive.

Example

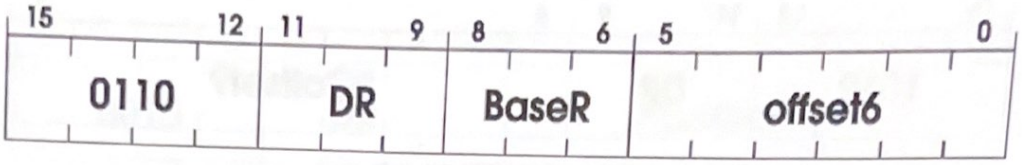
```
LDI R4, ONEMORE     ; R4 ← mem[mem[ONEMORE]]
```

LDR - Load Base+offset

Assembler Format

```
LDR DR, BaseR, offset6
```

Encoding



Operation

```
DR = mem[BaseR + SEXT(offset6)];
setcc();
```

Description

An address is computed by sign-extending bits [5:0] to 16 bits and adding this value to the contents of the register specified by bits [8:6]. The contents of memory at this address are loaded into DR. The condition codes are set, based on whether the value loaded is negative, zero, or positive.

Example

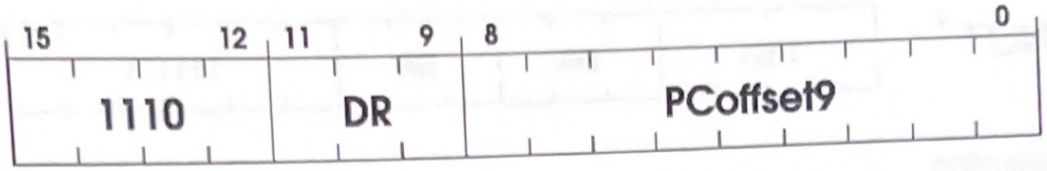
```
LDR R4, R2, #-5 ;       R4 ← mem[R2 - 5]
```

LEA - Load Effective Address

Assembler Format

```
LEA DR, LABEL
```

Encoding



Operation

```
DR = PC + SEXT(PCoffset9);
setcc();
```

Description

An address is computed by sign-extending bits [8:0] to 16 bits and adding this value to the incremented PC. This address is loaded into DR. The LEA instruction does not read memory to obtain the information to load into DR. The address itself is loaded into DR. The condition codes are set, based on whether the value loaded is negative, zero, or positive.

Example

```
LEA R4, TARGET      ; R4 ← address of TARGET.
```

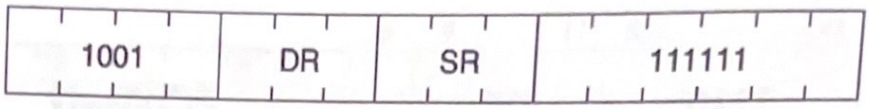
NOT - Bitwise Complement

Assembler Format

```
NOT DR, SR
```

Encoding

NOT+



Operation

```
DR = NOT(SR);
setcc();
```

Description

The bitwise complement of the contents of SR is stored in DR. The condition codes are set, based on whether the binary value produced, taken as a 2's complement integer, is negative, zero, or positive.

Example

```
NOT R4, R2     ; R4 ← NOT(R2)
```

RET - Return from Subroutine

This instruction is not used in X16

RTI - Return from Interrupt

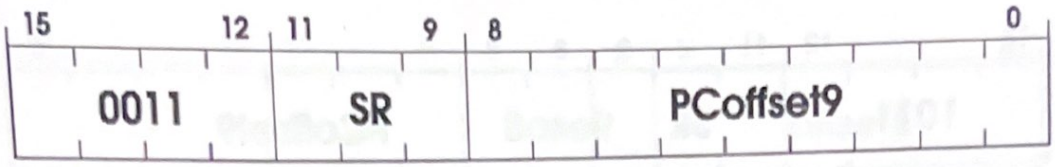
This instruction is not used in X16

ST - Store

Assembler Format

```
ST SR, LABEL
```

Encoding



Operation

```
mem[PC + SEXT(PCoffset9)] = SR;
```

Description

The contents of the register specified by SR are stored in the memory location whose address is computed by sign-extending bits [8:0] to 16 bits and adding this value to the incremented PC.

Example

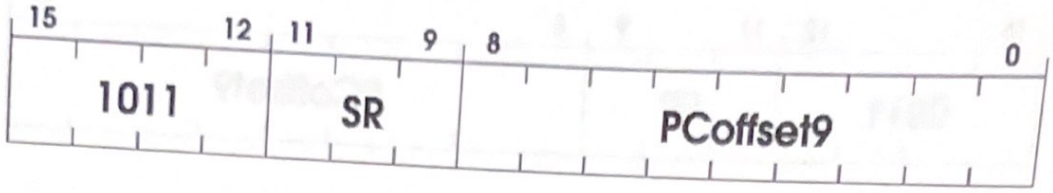
```
ST R4, HERE     ; mem[HERE] ← R4
```

STI - Store Indirect

Assembler Format

```
STI SR, LABEL
```

Encoding



Operation

```
mem[mem[PC† + SEXT(PCoffset9)]] = SR;
```

Description

The contents of the register specified by SR are stored in the memory location whose address is obtained as follows: Bits [8:0] are sign-extended to 16 bits and added to the incremented PC. What is in memory at this address is the address of the location to which the data in SR is stored.

Example

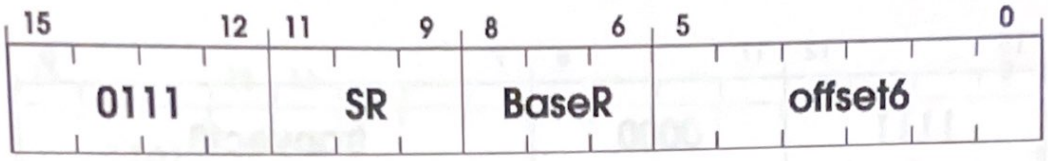
```
STI R4, NOT_HERE      ; mem[mem[NOT_HERE]] ← R4
```

STR - Store Base+offset

Assembler Format

```
STR SR, BaseR, offset6
```

Encoding



Operation

```
mem[BaseR + SEXT(offset6)] = SR;
```

Description

The contents of the register specified by SR are stored in the memory location whose address is computed by sign-extending bits [5:0] to 16 bits and adding this value to the contents of the register specified by bits [8:6].

Example

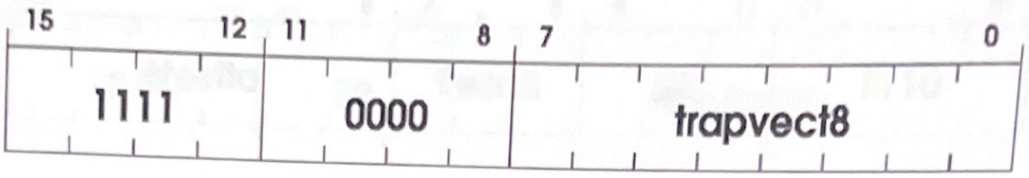
```
STR R4, R2, #5    ; mem[R2 + 5] ← R4
```

TRAP - System Call

Assembler Format

```
TRAP trapvector8
```

Encoding



Operation

```
R7 = PC;†
PC = mem[ZEXT(trapvect8)];
```

Description

First R7 is loaded with the incremented PC. (This enables a return to the instruction physically following the TRAP instruction in the original program after the service routine has completed execution.) Then the PC is loaded with the starting address of the system call specified by trapvector8. The starting address is contained in the memory location whose address is obtained by zero-extending trapvector8 to 16 bits.

Example

```
TRAP x23 ; Directs the operating system to execute the IN system call.
         ; The starting address of this system call is contained in
         ; memory location x0023.
```

Note

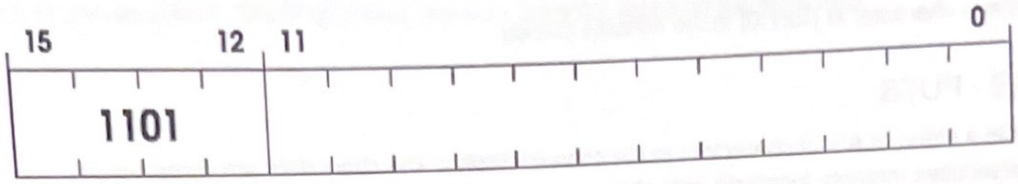
Memory locations x0000 through x00FF, 256 in all, are available to contain starting addresses for system calls specified by their corresponding trap vectors. This region of memory is called the Trap Vector Table. The trap service section describes the functions performed by the service routines corresponding to trap vectors x20 to x25.

Unused Opcode

Assembler Format

None

Encoding



Operation

Initiate an illegal opcode exception (halt the processor).

Description

If an illegal opcode is encountered, an illegal opcode exception occurs.

Note

The opcode 1101 has been reserved for future use. It is currently not defined. If the instruction currently executing has bits [15:12] = 1101, an illegal opcode exception occurs.

Trap Service Routines

x20 - GETC

Read a single character from the keyboard. The character is not echoed onto the console. Its ASCII code is copied into R0. The high eight bits of R0 are cleared.

x21 - PUTC

Write a character in R0[7:0] to the console display.

x22 - PUTS

Write a string of ASCII characters to the console display. The characters are contained in consecutive memory locations, one character per memory location, starting with the address specified in R0. Writing terminates with the occurrence of x0000 in a memory location.

x23 - ENTER

Print a prompt on the screen and read a single character from the keyboard. The character is echoed onto the console monitor, and its ASCII code is copied into R0. The high eight bits of R0 are cleared.

x24 - PUTSP

Write a string of ASCII characters to the console. The characters are contained in consecutive memory locations, two characters per memory location, starting with the address specified in R0. The ASCII code contained in bits [7:0] of a memory location is written to the console first. Then the ASCII code contained in bits [15:8] of that memory location is written to the console. (A character string consisting of an odd number of characters to be written will have x00 in bits [15:8] of the memory location containing the last character to be written.) Writing terminates with the occurrence of x0000 in a memory location.

x25 - HALT

Halt execution and print a message on the console.

Memory Mapped IO and Device Register Assignments

xFE00 - Keyboard status register

Also known as KBSR. The ready bit (bit [15]) indicates if the keyboard has received a new character.

xFE02 - Keyboard data register

Also known as KBDR. Bits [7:0] contain the last character typed on the keyboard.

Interrupt Vector Table

The Interrupt Vector Table is not used in X16.