

ADD

Addition

Assembler Formats

```
ADD  DR, SR1, SR2
ADD  DR, SR1, imm5
```

Encodings

15	12	11	9	8	6	5	4	3	2	0
0001	DR	SR1	0	00	SR2					

15	12	11	9	8	6	5	4			0
0001	DR	SR1	1	imm5						

Operation

```
if (bit[5] == 0)
    DR = SR1 + SR2;
else
    DR = SR1 + SEXT(imm5);
setcc();
```

Description

If bit [5] is 0, the second source operand is obtained from SR2. If bit [5] is 1, the second source operand is obtained by sign-extending the imm5 field to 16 bits. In both cases, the second source operand is added to the contents of SR1 and the result stored in DR. The condition codes are set, based on whether the result is negative, zero, or positive.

Examples

```
ADD  R2, R3, R4    ; R2 ← R3 + R4
ADD  R2, R3, #7     ; R2 ← R3 + 7
```

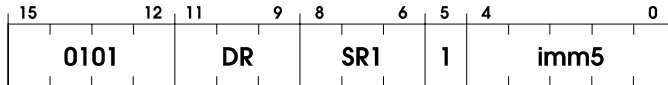
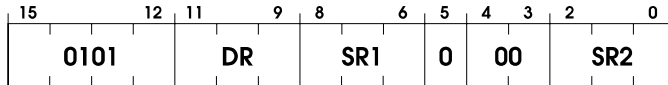
AND

Bit-wise Logical AND

Assembler Formats

```
AND DR, SR1, SR2
AND DR, SR1, imm5
```

Encodings



Operation

```
if (bit[5] == 0)
    DR = SR1 AND SR2;
else
    DR = SR1 AND SEXT(imm5);
setcc();
```

Description

If bit [5] is 0, the second source operand is obtained from SR2. If bit [5] is 1, the second source operand is obtained by sign-extending the imm5 field to 16 bits. In either case, the second source operand and the contents of SR1 are bit-wise ANDed, and the result 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.

Examples

```
AND R2, R3, R4    ;R2 ← R3 AND R4
AND R2, R3, #7     ;R2 ← R3 AND 7
```

BR

Conditional Branch

Assembler Formats

BRn LABEL BRzp LABEL

BRz LABEL BRnp LABEL

BRp LABEL BRnz LABEL

BR[†] LABEL BRnzp 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

BRzp LOOP ; Branch to LOOP if the last result was zero or positive.

BR[†] NEXT ; Unconditionally branch to NEXT.

[†]The assembly language opcode BR is interpreted the same as BRnzp; that is, always branch to the target address.

[‡]This is the incremented PC.

JMP

RET

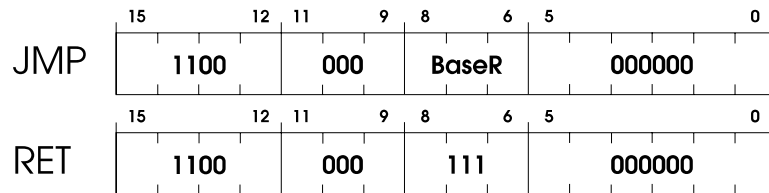
Jump

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.

JSR

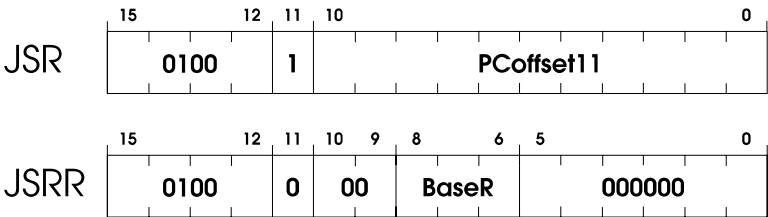
JSRR

Jump to Subroutine

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.
```

† This is the incremented PC.

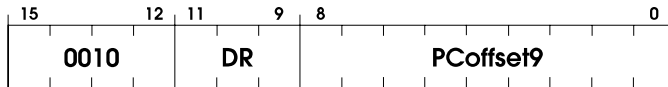
LD

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]
```

[†]This is the incremented PC.

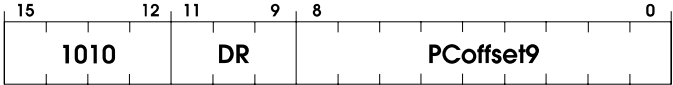
LDI

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]]

[†]This is the incremented PC.

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]
```

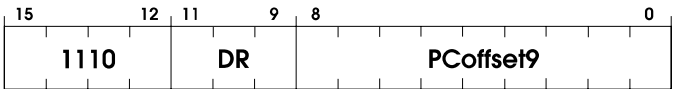

LEA

Load Effective Address

Assembler Format

LEA DR, LABEL

Encoding



Operation

$DR = PC^{\dagger} + SEXT(PCOffset9);$

~~LEA no longer sets condition codes~~

LEA no longer sets condition codes

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 condition codes are set, ~~LEA no longer sets condition codes~~

^{not}Example

LEA R4, TARGET ; R4 ← address of TARGET.

[†]This is the incremented PC.
[‡]The LEA instruction does not read memory to obtain the information to load into DR. The address itself is loaded into DR.

NOT

Bit-Wise Complement

Assembler Format

NOT DR, SR

Encoding

15	12	11	9	8	6	5	4	3	2	0
1001	DR	SR	1	11111						

Operation

$DR = \text{NOT}(SR) ;$
 $\text{setcc}();$

Description

The bit-wise 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 \leftarrow NOT(R2)

RET[†]

Return from Subroutine

Assembler Format

RET

Encoding

15	12	11	9	8	6	5	0
1100		000		111		000000	

Operation

PC = R7;

Description

The PC is loaded with the value in R7. This causes a return from a previous JSR instruction.

Example

RET ; PC ← R7

[†] The RET instruction is a specific encoding of the JMP instruction. See also JMP.

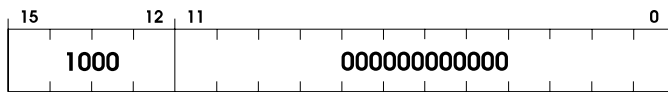
RTI

Return from Interrupt

Assembler Format

RTI

Encoding



Operation

```
if (PSR[15] == 0)
    PC = mem[R6]; R6 is the SSP
    R6 = R6+1;
    TEMP = mem[R6];
    R6 = R6+1;
    PSR = TEMP; the privilege mode and condition codes of
the interrupted process are restored
else
    Initiate a privilege mode exception;
```

Description

If the processor is running in Supervisor mode, the top two elements on the Supervisor Stack are popped and loaded into PC, PSR. If the processor is running in User mode, a privilege mode violation exception occurs.

Example

RTI ; PC, PSR ← top two values popped off stack.

Note

On an external interrupt or an internal exception, the initiating sequence first changes the privilege mode to Supervisor mode (PSR[15] = 0). Then the PSR and PC of the interrupted program are pushed onto the Supervisor Stack before loading the PC with the starting address of the interrupt or exception service routine. Interrupt and exception service routines run with Supervisor privilege. The last instruction in the service routine is RTI, which returns control to the interrupted program by popping two values off the Supervisor Stack to restore the PC and PSR. In the case of an interrupt, the PC is restored to the address of the instruction that was about to be processed when the interrupt was initiated. In the case of an exception, the PC is restored to either the address of the instruction that caused the exception or the address of the following instruction, depending on whether the instruction that caused the exception is to be re-executed. In the case of an interrupt, the PSR is restored to the value it had when the interrupt was initiated. In the case of an exception, the PSR is restored to the value it had when the exception occurred or to some modified value, depending on the exception. See also Section A.4.

If the processor is running in User mode, a privilege mode violation exception occurs. Section A.4 describes what happens in this case.

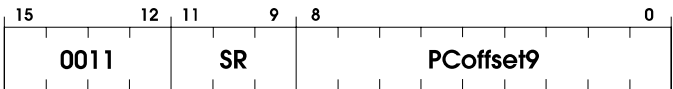
ST

Store

Assembler Format

ST SR, LABEL

Encoding



Operation

$\text{mem}[\text{PC}^\dagger + \text{SEXT}(\text{PCoffset9})] = \text{SR};$

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

[†]This is the incremented PC.

STI

Store Indirect

Assembler Format

STI SR, LABEL

Encoding



Operation

$\text{mem}[\text{mem}[\text{PC}^\dagger + \text{SEXT}(\text{PCoffset9})]] = \text{SR};$

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 ; $\text{mem}[\text{mem}[\text{NOT_HERE}]] \leftarrow \text{R4}$

[†] This is the incremented PC.

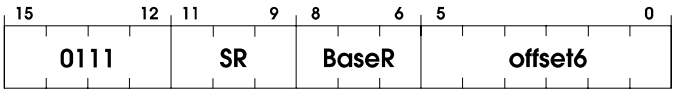
STR

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