3.4 Instruction Set

The complete MSP430 instruction set consists of 27 core instructions and 24 emulated instructions. The core instructions are instructions that have unique op-codes decoded by the CPU. The emulated instructions are instructions that make code easier to write and read, but do not have op-codes themselves, instead they are replaced automatically by the assembler with an equivalent core instruction. There is no code or performance penalty for using emulated instruction.

There are three core-instruction formats:

- Dual-operand
- Single-operand
- Jump

All single-operand and dual-operand instructions can be byte or word instructions by using .B or .W extensions. Byte instructions are used to access byte data or byte peripherals. Word instructions are used to access word data or word peripherals. If no extension is used, the instruction is a word instruction.

The source and destination of an instruction are defined by the following fields:

- src: The source operand defined by As and S-reg
- dst: The destination operand defined by Ad and D-reg
- As: The addressing bits responsible for the addressing mode used for the source (src)
- S-reg: The working register used for the source (src)
- Ad: The addressing bits responsible for the addressing mode used for the destination (dst)
- D-reg: The working register used for the destination (dst)
- B/W: Byte or word operation:
  - 0: word operation
  - 1: byte operation

**Note: Destination Address**

Destination addresses are valid anywhere in the memory map. However, when using an instruction that modifies the contents of the destination, the user must ensure the destination address is writable. For example, a masked-ROM location would be a valid destination address, but the contents are not modifiable, so the results of the instruction would be lost.
### 3.4.1 Double-Operand (Format I) Instructions

Figure 3-9 illustrates the double-operand instruction format.

**Figure 3-9. Double Operand Instruction Format**

![Double Operand Instruction Format](image)

Table 3-11 lists and describes the double operand instructions.

**Table 3-11. Double Operand Instructions**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>S-Reg, D-Reg</th>
<th>Operation</th>
<th>Status Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV(.B)</td>
<td>src,dst</td>
<td>src → dst</td>
<td>V N Z C</td>
</tr>
<tr>
<td>ADD(.B)</td>
<td>src,dst</td>
<td>src + dst → dst</td>
<td>* * *</td>
</tr>
<tr>
<td>ADDC(.B)</td>
<td>src,dst</td>
<td>src + dst + C → dst</td>
<td>* * *</td>
</tr>
<tr>
<td>SUB(.B)</td>
<td>src,dst</td>
<td>dst + .not.src + 1 → dst</td>
<td>* * *</td>
</tr>
<tr>
<td>SUBC(.B)</td>
<td>src,dst</td>
<td>dst + .not.src + C → dst</td>
<td>* * *</td>
</tr>
<tr>
<td>CMP(.B)</td>
<td>src,dst</td>
<td>dst - src</td>
<td>* * *</td>
</tr>
<tr>
<td>DADD(.B)</td>
<td>src,dst</td>
<td>src + dst + C → dst (decimally)</td>
<td>* * *</td>
</tr>
<tr>
<td>BIT(.B)</td>
<td>src,dst</td>
<td>src .and. dst</td>
<td>0 * * *</td>
</tr>
<tr>
<td>BIC(.B)</td>
<td>src,dst</td>
<td>.not.src .and. dst → dst</td>
<td>- - -</td>
</tr>
<tr>
<td>BIS(.B)</td>
<td>src,dst</td>
<td>src .or. dst → dst</td>
<td>- - -</td>
</tr>
<tr>
<td>XOR(.B)</td>
<td>src,dst</td>
<td>src .xor. dst → dst</td>
<td>* * *</td>
</tr>
<tr>
<td>AND(.B)</td>
<td>src,dst</td>
<td>src .and. dst → dst</td>
<td>0 * * *</td>
</tr>
</tbody>
</table>

* The status bit is affected
– The status bit is not affected
0 The status bit is cleared
1 The status bit is set

**Note:** Instructions **CMP** and **SUB**

The instructions **CMP** and **SUB** are identical except for the storage of the result. The same is true for the **BIT** and **AND** instructions.
### 3.4.2 Single-Operand (Format II) Instructions

Figure 3–10 illustrates the single-operand instruction format.

**Figure 3–10. Single Operand Instruction Format**

![Figure 3–10. Single Operand Instruction Format](image)

Table 3–12 lists and describes the single operand instructions.

**Table 3–12. Single Operand Instructions**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>S-Reg, D-Reg</th>
<th>Operation</th>
<th>Status Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRC(.B)</td>
<td>dst</td>
<td>C → MSB → LSB → C</td>
<td>*  *  *  *</td>
</tr>
<tr>
<td>RRA(.B)</td>
<td>dst</td>
<td>MSB → MSB → LSB → C</td>
<td>0  *  *  *</td>
</tr>
<tr>
<td>PUSH(.B)</td>
<td>src</td>
<td>SP − 2 → SP, src → @SP</td>
<td>− − − −</td>
</tr>
<tr>
<td>SWPB</td>
<td>dst</td>
<td>Swap bytes</td>
<td>− − − −</td>
</tr>
<tr>
<td>CALL</td>
<td>dst</td>
<td>SP − 2 → SP, PC+2 → @SP, dst → PC</td>
<td>− −</td>
</tr>
<tr>
<td>RETI</td>
<td></td>
<td>TOS → SR, SP + 2 → SP</td>
<td>*  *  *  *</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TOS → PC,SP + 2 → SP</td>
<td></td>
</tr>
<tr>
<td>SXT</td>
<td>dst</td>
<td>Bit 7 → Bit 8.......Bit 15</td>
<td>0  *  *  *</td>
</tr>
</tbody>
</table>

* The status bit is affected
− The status bit is not affected
0 The status bit is cleared
1 The status bit is set

All addressing modes are possible for the CALL instruction. If the symbolic mode (ADDRESS), the immediate mode (#N), the absolute mode (&EDE) or the indexed mode x(RN) is used, the word that follows contains the address information.
### 3.4.3 Jumps

Figure 3−11 shows the conditional-jump instruction format.

**Figure 3−11. Jump Instruction Format**

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op-code</td>
<td>C</td>
<td>10-Bit PC Offset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3−13 lists and describes the jump instructions.

**Table 3−13. Jump Instructions**

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>S-Reg, D-Reg</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>JEQ/JZ</td>
<td>Label</td>
<td>Jump to label if zero bit is set</td>
</tr>
<tr>
<td>JNE/JNZ</td>
<td>Label</td>
<td>Jump to label if zero bit is reset</td>
</tr>
<tr>
<td>JC</td>
<td>Label</td>
<td>Jump to label if carry bit is set</td>
</tr>
<tr>
<td>JNC</td>
<td>Label</td>
<td>Jump to label if carry bit is reset</td>
</tr>
<tr>
<td>JN</td>
<td>Label</td>
<td>Jump to label if negative bit is set</td>
</tr>
<tr>
<td>JGE</td>
<td>Label</td>
<td>Jump to label if (N .XOR. V) = 0</td>
</tr>
<tr>
<td>JL</td>
<td>Label</td>
<td>Jump to label if (N .XOR. V) = 1</td>
</tr>
<tr>
<td>JMP</td>
<td>Label</td>
<td>Jump to label unconditionally</td>
</tr>
</tbody>
</table>

Conditional jumps support program branching relative to the PC and do not affect the status bits. The possible jump range is from −511 to +512 words relative to the PC value at the jump instruction. The 10-bit program-counter offset is treated as a signed 10-bit value that is doubled and added to the program counter:

\[ \text{PC}_{\text{new}} = \text{PC}_{\text{old}} + 2 + \text{PC}_{\text{offset}} \times 2 \]
**ADC**[.W]  Add carry to destination  
**ADC.B**  Add carry to destination

**Syntax**  
ADC dst  or  ADC.W dst  
ADC.B dst

**Operation**  
dst + C -> dst

**Emulation**  
ADDC #0,dst  
ADDC.B #0,dst

**Description**  
The carry bit (C) is added to the destination operand. The previous contents of the destination are lost.

**Status Bits**  
N: Set if result is negative, reset if positive  
Z: Set if result is zero, reset otherwise  
C: Set if dst was incremented from 0FFFFh to 0000, reset otherwise  
    Set if dst was incremented from 0FFh to 00, reset otherwise  
V: Set if an arithmetic overflow occurs, otherwise reset

**Mode Bits**  
OSCOFF, CPUOFF, and GIE are not affected.

**Example**  
The 16-bit counter pointed to by R13 is added to a 32-bit counter pointed to by R12.  
ADD @R13,0(R12) ; Add LSDs  
ADC 2(R12) ; Add carry to MSD

**Example**  
The 8-bit counter pointed to by R13 is added to a 16-bit counter pointed to by R12.  
ADD.B @R13,0(R12) ; Add LSDs  
ADC.B 1(R12) ; Add carry to MSD
**Instruction Set**

**ADD.W**
Add source to destination

**ADD.B**
Add source to destination

**Syntax**
ADD src,dst or ADD.W src,dst
ADD.B src,dst

**Operation**
src + dst -> dst

**Description**
The source operand is added to the destination operand. The source operand is not affected. The previous contents of the destination are lost.

**Status Bits**
N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Set if there is a carry from the result, cleared if not
V: Set if an arithmetic overflow occurs, otherwise reset

**Mode Bits**
OSCOFF, CPUOFF, and GIE are not affected.

**Example**
R5 is increased by 10. The jump to TONI is performed on a carry.

```
ADD #10,R5
JC TONI ; Carry occurred
...... ; No carry
```

**Example**
R5 is increased by 10. The jump to TONI is performed on a carry.

```
ADD.B #10,R5 ; Add 10 to Lowbyte of R5
JC TONI ; Carry occurred, if (R5) ≥ 246 [0Ah+0F6h]
...... ; No carry
```
**Instruction Set**

**ADDC.W**
Add source and carry to destination

**ADDC.B**
Add source and carry to destination

**Syntax**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDC src,dst</td>
<td>src + dst + C -&gt; dst</td>
</tr>
<tr>
<td>ADDC.W src,dst</td>
<td>Add source and carry to destination</td>
</tr>
<tr>
<td>ADDC.B src,dst</td>
<td>Add source and carry to destination</td>
</tr>
</tbody>
</table>

**Operation**

src + dst + C -> dst

**Description**
The source operand and the carry bit (C) are added to the destination operand. The source operand is not affected. The previous contents of the destination are lost.

**Status Bits**

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Set if result is negative, reset if positive</td>
</tr>
<tr>
<td>Z</td>
<td>Set if result is zero, reset otherwise</td>
</tr>
<tr>
<td>C</td>
<td>Set if there is a carry from the MSB of the result, reset otherwise</td>
</tr>
<tr>
<td>V</td>
<td>Set if an arithmetic overflow occurs, otherwise reset</td>
</tr>
</tbody>
</table>

**Mode Bits**

OSCOFF, CPUOFF, and GIE are not affected.

**Example**
The 32-bit counter pointed to by R13 is added to a 32-bit counter, eleven words (20/2 + 2/2) above the pointer in R13.

```
ADD @R13+,20(R13) ; ADD LSDs with no carry in
ADDC @R13+,20(R13) ; ADD MSDs with carry
... ; resulting from the LSDs
```

**Example**
The 24-bit counter pointed to by R13 is added to a 24-bit counter, eleven words above the pointer in R13.

```
ADD.B @R13+,10(R13) ; ADD LSDs with no carry in
ADDC.B @R13+,10(R13) ; ADD medium Bits with carry
ADDC.B @R13+,10(R13) ; ADD MSDs with carry
... ; resulting from the LSDs
```
**Instruction Set**

**AND.W**  
Source AND destination

**AND.B**  
Source AND destination

**Syntax**  
AND src,dst or AND.W src,dst  
AND.B src,dst

**Operation**  
src .AND. dst -> dst

**Description**  
The source operand and the destination operand are logically ANDed. The result is placed into the destination.

**Status Bits**  
N: Set if result MSB is set, reset if not set  
Z: Set if result is zero, reset otherwise  
C: Set if result is not zero, reset otherwise ( = .NOT. Zero)  
V: Reset

**Mode Bits**  
OSCOFF, CPUOFF, and GIE are not affected.

**Example**  
The bits set in R5 are used as a mask (#0AA55h) for the word addressed by TOM. If the result is zero, a branch is taken to label TONI.

```
MOV #0AA55h,R5 ; Load mask into register R5
AND R5,TOM ; mask word addressed by TOM with R5
JZ TONI ; Result is not zero
...... ; or
AND #0AA55h,TOM
JZ TONI
```

**Example**  
The bits of mask #0A5h are logically ANDed with the low byte TOM. If the result is zero, a branch is taken to label TONI.

```
AND.B #0A5h,TOM ; mask Lowbyte TOM with 0A5h
JZ TONI ; Result is not zero
......
```
**BIC [W]**
Clear bits in destination

**BIC [B]**
Clear bits in destination

**Syntax**
- \( \text{BIC} \ src,\text{dst} \) or \( \text{BIC.W} \ src,\text{dst} \)
- \( \text{BIC.B} \ src,\text{dst} \)

**Operation**
\( \text{.NOT.} src \ \text{.AND.} \ dst \rightarrow \text{dst} \)

**Description**
The inverted source operand and the destination operand are logically ANDed. The result is placed into the destination. The source operand is not affected.

**Status Bits**
Status bits are not affected.

**Mode Bits**
OSCOFF, CPUOFF, and GIE are not affected.

**Example**
The six MSBs of the RAM word LEO are cleared.

\[ \text{BIC} \ #0FC00h,\text{LEO} \; ; \text{Clear 6 MSBs in MEM(LEO)} \]

**Example**
The five MSBs of the RAM byte LEO are cleared.

\[ \text{BIC.B} \ #0F8h,\text{LEO} \; ; \text{Clear 5 MSBs in Ram location LEO} \]
### Instruction Set

<table>
<thead>
<tr>
<th>BIS[W]</th>
<th>Set bits in destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIS.B</td>
<td>Set bits in destination</td>
</tr>
</tbody>
</table>

#### Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>BIS src,dst or BIS.W src,dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIS.B</td>
<td>src,dst</td>
</tr>
</tbody>
</table>

#### Operation

src .OR. dst -> dst

#### Description

The source operand and the destination operand are logically ORed. The result is placed into the destination. The source operand is not affected.

#### Status Bits

Status bits are not affected.

#### Mode Bits

OSCOFF, CPUOFF, and GIE are not affected.

#### Example

The six LSBs of the RAM word TOM are set.

BIS #003Fh,TOM; set the six LSBs in RAM location TOM

#### Example

The three MSBs of RAM byte TOM are set.

BIS.B #0E0h,TOM ; set the 3 MSBs in RAM location TOM
**Instruction Set**

**BIT[.W]**  
Test bits in destination

**BIT.B**  
Test bits in destination

**Syntax**  
BIT src,dst or BIT.W src,dst

**Operation**  
sr .AND. dst

**Description**  
The source and destination operands are logically ANDed. The result affects only the status bits. The source and destination operands are not affected.

**Status Bits**  
N: Set if MSB of result is set, reset otherwise  
Z: Set if result is zero, reset otherwise  
C: Set if result is not zero, reset otherwise (.NOT. Zero)  
V: Reset

**Mode Bits**  
OSCOFF, CPUOFF, and GIE are not affected.

**Example**  
If bit 9 of R8 is set, a branch is taken to label TOM.

```assembly
BIT #0200h,R8 ; bit 9 of R8 set?
JNZ TOM ; Yes, branch to TOM
... ; No, proceed
```

**Example**  
If bit 3 of R8 is set, a branch is taken to label TOM.

```assembly
BIT.B #8,R8
JC TOM
```

**Example**  
A serial communication receive bit (RCV) is tested. Because the carry bit is equal to the state of the tested bit while using the BIT instruction to test a single bit, the carry bit is used by the subsequent instruction; the read information is shifted into register RECBUF.

```assembly
; Serial communication with LSB is shifted first:

BIT.B #RCV,RCCTL ; Bit info into carry
RRC RECBUF ; Carry -> MSB of RECBUF
... ; repeat previous two instructions
... ; 8 times
... ; cccc cccc
; ^ ^
; MSB LSB

; Serial communication with MSB shifted first:

BIT.B #RCV,RCCTL ; Bit info into carry
RLC.B RECBUF ; Carry -> LSB of RECBUF
... ; repeat previous two instructions
... ; 8 times
... ; cccc cccc
; | LSB
; MSB
```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR, BRANCH</td>
<td>Branch to ........ destination</td>
</tr>
</tbody>
</table>

**Syntax**

```
BR    dst
```

**Operation**

```
dst -> PC
```

**Emulation**

```
MOV    dst,PC
```

**Description**

An unconditional branch is taken to an address anywhere in the 64K address space. All source addressing modes can be used. The branch instruction is a word instruction.

**Status Bits**

Status bits are not affected.

**Example**

Examples for all addressing modes are given.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR #EXEC</td>
<td><code>BR #EXEC ; Branch to label EXEC or direct branch (e.g. #0A4h)</code></td>
</tr>
<tr>
<td></td>
<td><code>; Core instruction MOV @PC+,PC</code></td>
</tr>
<tr>
<td>BR EXEC</td>
<td><code>BR EXEC ; Branch to the address contained in EXEC</code></td>
</tr>
<tr>
<td></td>
<td><code>; Core instruction MOV X(PC),PC</code></td>
</tr>
<tr>
<td></td>
<td><code>; Indirect address</code></td>
</tr>
<tr>
<td>BR &amp;EXEC</td>
<td><code>BR &amp;EXEC ; Branch to the address contained in absolute</code></td>
</tr>
<tr>
<td></td>
<td><code>; address EXEC</code></td>
</tr>
<tr>
<td></td>
<td><code>; Core instruction MOV X(0),PC</code></td>
</tr>
<tr>
<td></td>
<td><code>; Indirect address</code></td>
</tr>
<tr>
<td>BR R5</td>
<td><code>BR R5 ; Branch to the address contained in R5</code></td>
</tr>
<tr>
<td></td>
<td><code>; Core instruction MOV R5,PC</code></td>
</tr>
<tr>
<td></td>
<td><code>; Indirect R5</code></td>
</tr>
<tr>
<td>BR @R5</td>
<td><code>BR @R5 ; Branch to the address contained in the word</code></td>
</tr>
<tr>
<td></td>
<td><code>; pointed to by R5.</code></td>
</tr>
<tr>
<td></td>
<td><code>; Core instruction MOV @R5,PC</code></td>
</tr>
<tr>
<td></td>
<td><code>; Indirect, indirect R5</code></td>
</tr>
<tr>
<td>BR @R5+</td>
<td><code>BR @R5+ ; Branch to the address contained in the word pointed</code></td>
</tr>
<tr>
<td></td>
<td><code>; to by R5 and increment pointer in R5 afterwards.</code></td>
</tr>
<tr>
<td></td>
<td><code>; The next time—S/W flow uses R5 pointer—it can</code></td>
</tr>
<tr>
<td></td>
<td><code>; alter program execution due to access to</code></td>
</tr>
<tr>
<td></td>
<td><code>; next address in a table pointed to by R5</code></td>
</tr>
<tr>
<td></td>
<td><code>; Core instruction MOV @R5,PC</code></td>
</tr>
<tr>
<td></td>
<td><code>; Indirect, indirect R5 with autoincrement</code></td>
</tr>
<tr>
<td>BR X(R5)</td>
<td><code>BR X(R5) ; Branch to the address contained in the address</code></td>
</tr>
<tr>
<td></td>
<td><code>; pointed to by R5 + X (e.g. table with address</code></td>
</tr>
<tr>
<td></td>
<td><code>; starting at X). X can be an address or a label</code></td>
</tr>
<tr>
<td></td>
<td><code>; Core instruction MOV X(R5),PC</code></td>
</tr>
<tr>
<td></td>
<td><code>; Indirect, indirect R5 + X</code></td>
</tr>
</tbody>
</table>
CALL Subroutine

Syntax

CALL dst

Operation

dst -> tmp  dst is evaluated and stored
SP - 2 -> SP
PC -> @SP  PC updated to TOS
tmp -> PC  dst saved to PC

Description

A subroutine call is made to an address anywhere in the 64K address space. All addressing modes can be used. The return address (the address of the following instruction) is stored on the stack. The call instruction is a word instruction.

Status Bits

Status bits are not affected.

Example

Examples for all addressing modes are given.

CALL #EXEC ; Call on label EXEC or immediate address (e.g. #0A4h)
; SP−2 → SP, PC+2 → @SP, @PC+ → PC

CALL EXEC ; Call on the address contained in EXEC
; SP−2 → SP, PC+2 → @SP, X(PC) → PC
; Indirect address

CALL &EXEC ; Call on the address contained in absolute address
; EXEC
; SP−2 → SP, PC+2 → @SP, X(0) → PC
; Indirect address

CALL R5 ; Call on the address contained in R5
; SP−2 → SP, PC+2 → @SP, R5 → PC
; Indirect R5

CALL @R5 ; Call on the address contained in the word
; pointed to by R5
; SP−2 → SP, PC+2 → @SP, @R5 → PC
; Indirect, indirect R5

CALL @R5+ ; Call on the address contained in the word
; pointed to by R5 and increment pointer in R5.
; The next time—S/W flow uses R5 pointer—
; it can alter the program execution due to
; access to next address in a table pointed to by R5
; SP−2 → SP, PC+2 → @SP, @R5 → PC
; Indirect, indirect R5 with autoincrement

CALL X(R5) ; Call on the address contained in the address pointed
; to by R5 + X (e.g. table with address starting at X)
; X can be an address or a label
; SP−2 → SP, PC+2 → @SP, X(R5) → PC
; Indirect, indirect R5 + X
### Instruction Set

| **CLR.W** | Clear destination |
| **CLR.B** | Clear destination |

#### Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR dst or CLR.W dst</td>
</tr>
<tr>
<td>CLR.B dst</td>
</tr>
</tbody>
</table>

#### Operation

<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 -&gt; dst</td>
</tr>
</tbody>
</table>

#### Emulation

<table>
<thead>
<tr>
<th>Emulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV #0,dst</td>
</tr>
<tr>
<td>MOV.B #0,dst</td>
</tr>
</tbody>
</table>

#### Description

The destination operand is cleared.

#### Status Bits

Status bits are not affected.

#### Example

RAM word TONI is cleared.

```assembly
CLR TONI ; 0 -> TONI
```

Register R5 is cleared.

```assembly
CLR R5
```

RAM byte TONI is cleared.

```assembly
CLR.B TONI ; 0 -> TONI
```
* CLRC

Clear carry bit

Syntax

CLRC

Operation

0 -> C

Emulation

BIC #1,SR

Description

The carry bit (C) is cleared. The clear carry instruction is a word instruction.

Status Bits

N: Not affected
Z: Not affected
C: Cleared
V: Not affected

Mode Bits

OSCOFF, CPUOFF, and GIE are not affected.

Example

The 16-bit decimal counter pointed to by R13 is added to a 32-bit counter pointed to by R12.

```
CLRC ; C=0: defines start
DADD @R13,0(R12) ; add 16-bit counter to low word of 32-bit counter
DADC 2(R12) ; add carry to high word of 32-bit counter
```
**Instruction Set**

<table>
<thead>
<tr>
<th><em>CLRN</em></th>
<th>Clear negative bit</th>
</tr>
</thead>
</table>

**Syntax**

CLRN

**Operation**

0 → N  

or  

(.NOT.src .AND. dst −> dst)

**Emulation**

BIC #4,SR

**Description**

The constant 04h is inverted (0FFFBh) and is logically ANDed with the destination operand. The result is placed into the destination. The clear negative bit instruction is a word instruction.

**Status Bits**

- **N**: Reset to 0  
- **Z**: Not affected  
- **C**: Not affected  
- **V**: Not affected

**Mode Bits**

- OSCOFF, CPUOFF, and GIE are not affected.

**Example**

The Negative bit in the status register is cleared. This avoids special treatment with negative numbers of the subroutine called.

```
CLRN
CALL SUBR
......
......
SUBR JN SUBRET ; If input is negative: do nothing and return
......
......
SUBRET RET
```
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLRZ</td>
<td>Clear zero bit</td>
</tr>
</tbody>
</table>

**Syntax**
CLRZ

**Operation**
0 → Z
or
(.NOT.src .AND. dst → dst)

**Emulation**
BIC #2,SR

**Description**
The constant 02h is inverted (0FFFDh) and logically ANDed with the destination operand. The result is placed into the destination. The clear zero bit instruction is a word instruction.

**Status Bits**
- N: Not affected
- Z: Reset to 0
- C: Not affected
- V: Not affected

**Mode Bits**
OSCOFF, CPUOFF, and GIE are not affected.

**Example**
The zero bit in the status register is cleared.
CLRZ
### Instruction Set

**CMP[.W]**  
Compare source and destination

**CMP.B**  
Compare source and destination

### Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP src,dst or CMP.W src,dst</td>
<td>dst + .NOT.src + 1</td>
</tr>
<tr>
<td>CMP.B src,dst</td>
<td>(dst - src)</td>
</tr>
</tbody>
</table>

### Operation

\[\text{dst} + \text{.NOT. src} + 1\]

or

\[(\text{dst} - \text{src})\]

### Description

The source operand is subtracted from the destination operand. This is accomplished by adding the 1s complement of the source operand plus 1. The two operands are not affected and the result is not stored; only the status bits are affected.

### Status Bits

<table>
<thead>
<tr>
<th>Status Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N:</td>
<td>Set if result is negative, reset if positive (src &gt;= dst)</td>
</tr>
<tr>
<td>Z:</td>
<td>Set if result is zero, reset otherwise (src = dst)</td>
</tr>
<tr>
<td>C:</td>
<td>Set if there is a carry from the MSB of the result, reset otherwise</td>
</tr>
<tr>
<td>V:</td>
<td>Set if an arithmetic overflow occurs, otherwise reset</td>
</tr>
</tbody>
</table>

### Mode Bits

OSCOFF, CPUOFF, and GIE are not affected.

### Example

R5 and R6 are compared. If they are equal, the program continues at the label EQUAL.

```
CMP R5,R6 ; R5 = R6?
JEQ EQUAL ; YES, JUMP
```

### Example

Two RAM blocks are compared. If they are not equal, the program branches to the label ERROR.

```
MOV #NUM,R5 ; number of words to be compared
MOV #BLOCK1,R6 ; BLOCK1 start address in R6
MOV #BLOCK2,R7 ; BLOCK2 start address in R7
L$1 CMP @R6+,0(R7) ; Are Words equal? R6 increments
JNZ ERROR ; No, branch to ERROR
INCD R7 ; Increment R7 pointer
DEC R5 ; Are all words compared?
JNZ L$1 ; No, another compare
```

### Example

The RAM bytes addressed by EDE and TONI are compared. If they are equal, the program continues at the label EQUAL.

```
CMP.B EDE,TONI ; MEM(EDE) = MEM(TONI)?
JEQ EQUAL ; YES, JUMP
```
* DADC[.W] Add carry decimally to destination
* DADC.B Add carry decimally to destination

Syntax

- DADC dst or DADC.W src,dst
- DADC.B dst

Operation
dst + C -> dst (decimally)

Emulation

- DADD #0,dst
- DADD.B #0,dst

Description
The carry bit (C) is added decimally to the destination.

Status Bits

- N: Set if MSB is 1
- Z: Set if dst is 0, reset otherwise
- C: Set if destination increments from 9999 to 0000, reset otherwise
  - Set if destination increments from 99 to 00, reset otherwise
- V: Undefined

Mode Bits
OSCOFF, CPUOFF, and GIE are not affected.

Example
The four-digit decimal number contained in R5 is added to an eight-digit decimal number pointed to by R8.

```assembly
CLRC ; Reset carry
DADD R5,0(R8) ; Add LSDs + C
DADC 2(R8) ; Add carry to MSD
```

Example
The two-digit decimal number contained in R5 is added to a four-digit decimal number pointed to by R8.

```assembly
CLRC ; Reset carry
DADD.B R5,0(R8) ; Add LSDs + C
DADC 1(R8) ; Add carry to MSDs
```
**DADD[.W]**
Source and carry added decimally to destination

**DADD.B**
Source and carry added decimally to destination

**Syntax**
- `DADD src,dst` or `DADD.W src,dst`
- `DADD.B src,dst`

**Operation**
`src + dst + C -> dst` (decimally)

**Description**
The source operand and the destination operand are treated as four binary coded decimals (BCD) with positive signs. The source operand and the carry bit (C) are added decimally to the destination operand. The source operand is not affected. The previous contents of the destination are lost. The result is not defined for non-BCD numbers.

**Status Bits**
- **N**: Set if the MSB is 1, reset otherwise
- **Z**: Set if result is zero, reset otherwise
- **C**: Set if the result is greater than 9999
  - Set if the result is greater than 99
- **V**: Undefined

**Mode Bits**
OSCOFF, CPUOFF, and GIE are not affected.

**Example**
The eight-digit BCD number contained in R5 and R6 is added decimally to an eight-digit BCD number contained in R3 and R4 (R6 and R4 contain the MSDs).

```
CLRC ; clear carry
DADD R5,R3 ; add LSDs
DADD R6,R4 ; add MSDs with carry
JC OVERFLOW ; If carry occurs go to error handling routine
```

**Example**
The two-digit decimal counter in the RAM byte CNT is incremented by one.

```
CLRC ; clear carry
DADD.B #1,CNT ; increment decimal counter
```

or

```
SETC
DADD.B #0,CNT ; ≡ DADC.B CNT
```
Instruction Set

* DEC[W]
* DEC.B

Decrement destination
Decrement destination

Syntax

DEC dst or DEC.W dst
DEC.B dst

Operation

dst − 1 → dst

Emulation

SUB #1,dst
SUB.B #1,dst

Description

The destination operand is decremented by one. The original contents are lost.

Status Bits

N: Set if result is negative, reset if positive
Z: Set if dst contained 1, reset otherwise
C: Reset if dst contained 0, set otherwise
V: Set if an arithmetic overflow occurs, otherwise reset.
   Set if initial value of destination was 08000h, otherwise reset.
   Set if initial value of destination was 080h, otherwise reset.

Mode Bits

OSCOFF, CPUOFF, and GIE are not affected.

Example

R10 is decremented by 1

DEC R10 ; Decrement R10

; Move a block of 255 bytes from memory location starting with EDE to memory location starting with
; TONI. Tables should not overlap: start of destination address TONI must not be within the range EDE
; to EDE+0FEh
;
; MOV     #EDE,R6
MOV     #255,R10
L$1     MOV.B  @R6+,TONI−EDE−1(R6)
DEC     R10
JNZ     L$1

; Do not transfer tables using the routine above with the overlap shown in Figure 3−12.

Figure 3−12. Decrement Overlap

RISC 16−Bit CPU 3-37
**Instruction Set**

* DECD.[W] Double-decrement destination
* DECD.B Double-decrement destination

**Syntax**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DECD</td>
<td>dst</td>
</tr>
<tr>
<td>DECD.W</td>
<td>dst</td>
</tr>
<tr>
<td>DECD.B</td>
<td>dst</td>
</tr>
</tbody>
</table>

**Operation**

\[ \text{dst} - 2 \rightarrow \text{dst} \]

**Emulation**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SUB</td>
<td>#2,dst</td>
</tr>
<tr>
<td>SUB.B</td>
<td>#2,dst</td>
</tr>
</tbody>
</table>

**Description**

The destination operand is decremented by two. The original contents are lost.

**Status Bits**

- **N**: Set if result is negative, reset if positive
- **Z**: Set if dst contained 2, reset otherwise
- **C**: Reset if dst contained 0 or 1, set otherwise
- **V**: Set if an arithmetic overflow occurs, otherwise reset.
  - Set if initial value of destination was 08001 or 08000h, otherwise reset.
  - Set if initial value of destination was 081 or 080h, otherwise reset.

**Mode Bits**

OSCOFF, CPUOFF, and GIE are not affected.

**Example**

R10 is decremented by 2.

```
DECD R10 ; Decrement R10 by two
```

Move a block of 255 words from memory location starting with EDE to memory location starting with TONI.

- Tables should not overlap: start of destination address TONI must not be within the range EDE to EDE+0FEh

```
MOV #EDE,R6
MOV #510,R10
L$1 MOV @R6+,TONI-EDE-2(R6)
DECD R10
JNZ L$1
```

**Example**

Memory at location LEO is decremented by two.

```
DECD.B LEO ; Decrement MEM(LEO)
```

Decrement status byte STATUS by two.

```
DECD.B STATUS
```
* DINT

Disable (general) interrupts

**Syntax**

DINT

**Operation**

\[ 0 \rightarrow \text{GIE} \]

or

\[ (\text{0FFFF7h .AND. SR} \rightarrow \text{SR} / \text{.NOT.src .AND. dst } \rightarrow \text{dst}) \]

**Emulation**

BIC \#8,SR

**Description**

All interrupts are disabled. The constant 08h is inverted and logically ANDed with the status register (SR). The result is placed into the SR.

**Status Bits**

Status bits are not affected.

**Mode Bits**

GIE is reset. OSCOFF and CPUOFF are not affected.

**Example**

The general interrupt enable (GIE) bit in the status register is cleared to allow a nondisrupted move of a 32-bit counter. This ensures that the counter is not modified during the move by any interrupt.

```
DINT ; All interrupt events using the GIE bit are disabled
NOP
MOV COUNTHI,R5 ; Copy counter
MOV COUNTLO,R6
EINT ; All interrupt events using the GIE bit are enabled
```

---

**Note: Disable Interrupt**

If any code sequence needs to be protected from interruption, the DINT should be executed at least one instruction before the beginning of the uninterruptible sequence, or should be followed by a NOP instruction.
* EINT

Enable (general) interrupts

Syntax

EINT

Operation

1 → GIE
or
(0008h .OR. SR → SR / .src .OR. dst → dst)

Emulation

BIS #8,SR

Description

All interrupts are enabled. The constant #08h and the status register SR are logically ORed. The result is placed into the SR.

Status Bits

Status bits are not affected.

Mode Bits

GIE is set. OSCOFF and CPUOFF are not affected.

Example

The general interrupt enable (GIE) bit in the status register is set.

; Interrupt routine of ports P1.2 to P1.7
; P1IN is the address of the register where all port bits are read. P1IFG is the address of
; the register where all interrupt events are latched.

PUSH.B &P1IN
BIC.B @SP,&P1IFG ; Reset only accepted flags
EINT ; Preset port 1 interrupt flags stored on stack
; other interrupts are allowed
BIT #Mask,@SP
JEQ MaskOK ; Flags are present identically to mask: jump
......
MaskOK
BIC #Mask,@SP
......
INCD SP ; Housekeeping: inverse to PUSH instruction
; at the start of interrupt subroutine. Corrects
; the stack pointer.
RETI

Note: Enable Interrupt

The instruction following the enable interrupt instruction (EINT) is always executed, even if an interrupt service request is pending when the interrupts are enable.
**INC[.W]**  Increment destination

**INC.B**  Increment destination

**Syntax**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>INC dst</td>
<td>dst + 1 → dst</td>
</tr>
<tr>
<td>INC.W dst</td>
<td></td>
</tr>
<tr>
<td>INC.B dst</td>
<td></td>
</tr>
</tbody>
</table>

**Operation**

dst + 1 → dst

**Emulation**

ADD #1, dst

**Description**

The destination operand is incremented by one. The original contents are lost.

**Status Bits**

- **N:** Set if result is negative, reset if positive
- **Z:** Set if dst contained 0FFFFh, reset otherwise
  - Set if dst contained 0FFh, reset otherwise
- **C:** Set if dst contained 0FFFFh, reset otherwise
  - Set if dst contained 0FFh, reset otherwise
- **V:** Set if dst contained 07FFFh, reset otherwise
  - Set if dst contained 07Fh, reset otherwise

**Mode Bits**

OSCOFF, CPUOFF, and GIE are not affected.

**Example**

The status byte, STATUS, of a process is incremented. When it is equal to 11, a branch to OVFL is taken.

```
INC.B STATUS
CMP.B #11,STATUS
JEQ OVFL
```
* INCD.W
    Double-increment destination
* INCD.B
    Double-increment destination

Syntax
INCD dst or INCD.W dst
INCD.B dst

Operation
dst + 2 -> dst

Emulation
ADD #2,dst
ADD.B #2,dst

Example
The destination operand is incremented by two. The original contents are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if dst contained 0FFFEh, reset otherwise
    Set if dst contained 0FEh, reset otherwise
C: Set if dst contained 0FFFEh or 0FFFFh, reset otherwise
    Set if dst contained 0FEh or 0FFh, reset otherwise
V: Set if dst contained 07FFEh or 07FFFh, reset otherwise
    Set if dst contained 07Eh or 07Fh, reset otherwise

Mode Bits
OSCOFF, CPUOFF, and GIE are not affected.

Example
The item on the top of the stack (TOS) is removed without using a register.

........
PUSH R5 ; R5 is the result of a calculation, which is stored
         ; in the system stack
INCD SP ; Remove TOS by double-increment from stack
         ; Do not use INCD.B, SP is a word-aligned
         ; register
RET

Example
The byte on the top of the stack is incremented by two.

INCD.B 0(SP) ; Byte on TOS is incremented by two
* INV.W  Invert destination
* INV.B Invert destination

Syntax  INV dst
        INV.B dst

Operation  .NOT.dst -> dst

Emulation  XOR  #0FFFFh,dst
            XOR.B #0FFh,dst

Description  The destination operand is inverted. The original contents are lost.

Status Bits
N: Set if result is negative, reset if positive
Z: Set if dst contained 0FFFFh, reset otherwise
C: Set if result is not zero, reset otherwise ( = .NOT. Zero)
V: Set if initial destination operand was negative, otherwise reset

Mode Bits  OSCOFF, CPUOFF, and GIE are not affected.

Example  Content of R5 is negated (twos complement).
          MOV  #00AEh,R5 ; R5 = 000AEh
          INV  R5 ; Invert R5, R5 = 0FF51h
          INC  R5 ; R5 is now negated, R5 = 0FF52h

Example  Content of memory byte LEO is negated.
          MOV.B #0AEh,LEO ; MEM(LEO) = 0AEh
          INV.B LEO ; Invert LEO, MEM(LEO) = 051h
          INC.B LEO ; MEM(LEO) is negated, MEM(LEO) = 052h
### Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JC</td>
<td>Jump if carry set</td>
</tr>
<tr>
<td>JHS</td>
<td>Jump if higher or same</td>
</tr>
</tbody>
</table>

#### Syntax

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>JC</td>
<td>JC label</td>
</tr>
<tr>
<td>JHS</td>
<td>JHS label</td>
</tr>
</tbody>
</table>

#### Operation

- If C = 1: \( PC + 2 \times \text{offset} \rightarrow PC \)
- If C = 0: execute following instruction

#### Description

The status register carry bit (C) is tested. If it is set, the 10-bit signed offset contained in the instruction LSBs is added to the program counter. If C is reset, the next instruction following the jump is executed. JC (jump if carry/higher or same) is used for the comparison of unsigned numbers (0 to 65536).

#### Status Bits

Status bits are not affected.

#### Example

The P1IN.1 signal is used to define or control the program flow.

```assembly
BIT #01h,&P1IN ; State of signal -> Carry
JC PROGA ; If carry=1 then execute program routine A
...... ; Carry=0, execute program here
```

#### Example

R5 is compared to 15. If the content is higher or the same, branch to LABEL.

```assembly
CMP #15,R5
JHS LABEL ; Jump is taken if R5 ≥ 15
...... ; Continue here if R5 < 15
```
**Instruction Set**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>JEQ, JZ</strong></td>
<td>Jump if equal, jump if zero</td>
</tr>
</tbody>
</table>

**Syntax**

JEQ label, JZ label

**Operation**

If $Z = 1$: $PC + 2 \times \text{offset} \rightarrow PC$
If $Z = 0$: execute following instruction

**Description**

The status register zero bit ($Z$) is tested. If it is set, the 10-bit signed offset contained in the instruction LSBs is added to the program counter. If $Z$ is not set, the instruction following the jump is executed.

**Status Bits**

Status bits are not affected.

**Example**

Jump to address TONI if R7 contains zero.

```
TST R7 ; Test R7
JZ TONI ; if zero: JUMP
```

Jump to address LEO if R6 is equal to the table contents.

```
CMP R6,Table(R5) ; Compare content of R6 with content of MEM (table address + content of R5)
JEQ LEO ; Jump if both data are equal
...... ; No, data are not equal, continue here
```

Branch to LABEL if R5 is 0.

```
TST R5
JZ LABEL
......
```
**Instruction Set**

**JGE**
Jump if greater or equal

**Syntax**
JGE label

**Operation**
If $(N \oplus V) = 0$ then jump to label: $PC + 2 \times$ offset $\rightarrow PC$
If $(N \oplus V) = 1$ then execute the following instruction

**Description**
The status register negative bit $(N)$ and overflow bit $(V)$ are tested. If both $N$ and $V$ are set or reset, the 10-bit signed offset contained in the instruction LSBs is added to the program counter. If only one is set, the instruction following the jump is executed.

This allows comparison of signed integers.

**Status Bits**
Status bits are not affected.

**Example**
When the content of $R6$ is greater or equal to the memory pointed to by $R7$, the program continues at label $EDE$.

```
CMP @R7,R6 ; R6 $\geq$ (R7)?, compare on signed numbers
JGE EDE ; Yes, R6 $\geq$ (R7)
...... ; No, proceed
......
......
```
**JL**  
Jump if less  

**Syntax**  
JL label  

**Operation**  
If \((N \text{ .XOR. } V) = 1\) then jump to label: \(PC + 2 \times \text{ offset} \to PC\)  
If \((N \text{ .XOR. } V) = 0\) then execute following instruction  

**Description**  
The status register negative bit (N) and overflow bit (V) are tested. If only one is set, the 10-bit signed offset contained in the instruction LSBs is added to the program counter. If both N and V are set or reset, the instruction following the jump is executed.  

This allows comparison of signed integers.  

**Status Bits**  
Status bits are not affected.  

**Example**  
When the content of R6 is less than the memory pointed to by R7, the program continues at label EDE.  

```
CMP @R7,R6 ; R6 < (R7)?, compare on signed numbers
JL EDE ; Yes, R6 < (R7)
...... ; No, proceed
......
```

**RISC 16–Bit CPU**
### Instruction Set

**JMP**

<table>
<thead>
<tr>
<th>Jump unconditionally</th>
</tr>
</thead>
</table>

**Syntax**

<table>
<thead>
<tr>
<th>JMP label</th>
</tr>
</thead>
</table>

**Operation**

<table>
<thead>
<tr>
<th>PC + 2 × offset → PC</th>
</tr>
</thead>
</table>

**Description**

The 10-bit signed offset contained in the instruction LSBs is added to the program counter.

**Status Bits**

Status bits are not affected.

**Hint:**

This one-word instruction replaces the BRANCH instruction in the range of −511 to +512 words relative to the current program counter.
### JN

**Jump if negative**

**Syntax**

JN label

**Operation**

if N = 1: PC + 2 × offset -> PC

if N = 0: execute following instruction

**Description**

The negative bit (N) of the status register is tested. If it is set, the 10-bit signed offset contained in the instruction LSBs is added to the program counter. If N is reset, the next instruction following the jump is executed.

**Status Bits**

Status bits are not affected.

**Example**

The result of a computation in R5 is to be subtracted from COUNT. If the result is negative, COUNT is to be cleared and the program continues execution in another path.

```
SUB R5,COUNT ; COUNT − R5 −> COUNT
JN L$1 ; If negative continue with COUNT=0 at PC=L$1
 ...... ; Continue with COUNT≥0
 ......
 ......
L$1 CLR COUNT
 ......
 ......
 ......
```
Instruction Set

JNC  Jump if carry not set
JLO  Jump if lower

Syntax
- JNC label
- JLO label

Operation
- if $C = 0$: PC + 2 $\times$ offset $\rightarrow$ PC
- if $C = 1$: execute following instruction

Description
The status register carry bit ($C$) is tested. If it is reset, the 10-bit signed offset contained in the instruction LSBs is added to the program counter. If $C$ is set, the next instruction following the jump is executed. JNC (jump if no carry/lower) is used for the comparison of unsigned numbers (0 to 65536).

Status Bits
Status bits are not affected.

Example
The result in R6 is added in BUFFER. If an overflow occurs, an error handling routine at address ERROR is used.

```
ADD R6, BUFFER ; BUFFER + R6 $\rightarrow$ BUFFER
JNC CONT ; No carry, jump to CONT
```

Example
Branch to STL2 if byte STATUS contains 1 or 0.

```
CMP.B #2, STATUS
JLO STL2 ; STATUS $\leq$ 2
...... ; STATUS $\geq$ 2, continue here
```
### Instruction Set

#### JNE
- **Jump if not equal**

#### JNZ
- **Jump if not zero**

#### Syntax
- **JNE** label
- **JNZ** label

#### Operation
- If $Z = 0$: PC + 2 × offset $\rightarrow$ PC
- If $Z = 1$: execute following instruction

#### Description
The status register zero bit (Z) is tested. If it is reset, the 10-bit signed offset contained in the instruction LSBs is added to the program counter. If Z is set, the next instruction following the jump is executed.

#### Status Bits
Status bits are not affected.

#### Example
Jump to address TONI if R7 and R8 have different contents.

```
CMP R7, R8 ; COMPARE R7 WITH R8
JNE TONI ; if different: jump
...... ; if equal, continue
```
### Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV.W</td>
<td>Move source to destination</td>
</tr>
<tr>
<td>MOV.B</td>
<td>Move source to destination</td>
</tr>
</tbody>
</table>

#### Syntax

- **MOV src,dst** or **MOV.W src,dst**
- **MOV.B src,dst**

#### Operation

src → dst

#### Description

- The source operand is moved to the destination.
- The source operand is not affected. The previous contents of the destination are lost.

#### Status Bits

Status bits are not affected.

#### Mode Bits

OSCOFF, CPUOFF, and GIE are not affected.

#### Example

The contents of table EDE (word data) are copied to table TOM. The length of the tables must be 020h locations.

```
MOV  #EDE,R10          ; Prepare pointer
MOV  #020h,R9          ; Prepare counter
Loop
  MOV  @R10+,TOM-EDE-2(R10) ; Use pointer in R10 for both tables
  DEC  R9               ; Decrement counter
  JNZ  Loop             ; Counter ≠ 0, continue copying
......  ; Copying completed
......
......
```

The contents of table EDE (byte data) are copied to table TOM. The length of the tables should be 020h locations.

```
MOV  #EDE,R10          ; Prepare pointer
MOV  #020h,R9          ; Prepare counter
Loop
  MOV.B  @R10+,TOM-EDE-1(R10) ; Use pointer in R10 for both tables
  DEC  R9               ; Decrement counter
  JNZ  Loop             ; Counter ≠ 0, continue copying
......  ; Copying completed
......
```

---

*RISC 16−Bit CPU*
### * NOP

No operation

**Syntax**

NOP

**Operation**

None

**Emulation**

MOV #0, R3

**Description**

No operation is performed. The instruction may be used for the elimination of instructions during the software check or for defined waiting times.

**Status Bits**

Status bits are not affected.

The NOP instruction is mainly used for two purposes:

- To fill one, two, or three memory words
- To adjust software timing

---

**Note: Emulating No-Operation Instruction**

Other instructions can emulate the NOP function while providing different numbers of instruction cycles and code words. Some examples are:

- MOV #0,R3 ; 1 cycle, 1 word
- MOV 0(R4),0(R4) ; 6 cycles, 3 words
- MOV @R4,0(R4) ; 5 cycles, 2 words
- BIC #0,EDE(R4) ; 4 cycles, 2 words
- JMP $+2 ; 2 cycles, 1 word
- BIC #0,R5 ; 1 cycle, 1 word

However, care should be taken when using these examples to prevent unintended results. For example, if MOV 0(R4), 0(R4) is used and the value in R4 is 120h, then a security violation will occur with the watchdog timer (address 120h) because the security key was not used.
**Instruction Set**

* POP[W]  Pop word from stack to destination  
* POP.B  Pop byte from stack to destination

**Syntax**

<table>
<thead>
<tr>
<th>POP</th>
<th>dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>POP.B</td>
<td>dst</td>
</tr>
</tbody>
</table>

**Operation**

@SP  -> temp  
SP + 2  -> SP  
temp  -> dst

**Emulation**

MOV  @SP+,dst  or  MOV.W  @SP+,dst  
MOV.B  @SP+,dst

**Description**

The stack location pointed to by the stack pointer (TOS) is moved to the destination. The stack pointer is incremented by two afterwards.

**Status Bits**

Status bits are not affected.

**Example**

The contents of R7 and the status register are restored from the stack.

POP  R7  ; Restore R7  
POP  SR  ; Restore status register

**Example**

The contents of RAM byte LEO is restored from the stack.

POP.B  LEO  ; The low byte of the stack is moved to LEO.

**Example**

The contents of R7 is restored from the stack.

POP.B  R7  ; The low byte of the stack is moved to R7, the high byte of R7 is 00h

**Example**

The contents of the memory pointed to by R7 and the status register are restored from the stack.

POP.B  0(R7)  ; The low byte of the stack is moved to the byte which is pointed to by R7  
; Example:  R7 = 203h  
; Mem(R7) = low byte of system stack  
; Example:  R7 = 20Ah  
; Mem(R7) = low byte of system stack  
POP  SR  ; Last word on stack moved to the SR

**Note:** The System Stack Pointer

The system stack pointer (SP) is always incremented by two, independent of the byte suffix.
### Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PUSH[W]</strong></td>
<td>Push word onto stack</td>
</tr>
<tr>
<td><strong>PUSH.B</strong></td>
<td>Push byte onto stack</td>
</tr>
</tbody>
</table>

#### Syntax

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PUSH src</code></td>
<td><code>SP ← SP</code></td>
</tr>
<tr>
<td><code>PUSH.W src</code></td>
<td><code>SP ← SP</code></td>
</tr>
<tr>
<td><code>PUSH.B src</code></td>
<td><code>@SP ← src</code></td>
</tr>
</tbody>
</table>

#### Operation

- `SP − 2 → SP`
- `src → @SP`

#### Description

The stack pointer is decremented by two, then the source operand is moved to the RAM word addressed by the stack pointer (TOS).

#### Status Bits

- Status bits are not affected.

#### Mode Bits

- OSCOFF, CPUOFF, and GIE are not affected.

#### Example

The contents of the status register and R8 are saved on the stack.

```
PUSH SR ; save status register
PUSH R8 ; save R8
```

#### Example

The contents of the peripheral TCDAT is saved on the stack.

```
PUSH.B &TCDAT ; save data from 8-bit peripheral module, address TCDAT, onto stack
```

#### Note: The System Stack Pointer

The system stack pointer (SP) is always decremented by two, independent of the byte suffix.
### Instruction Set

<table>
<thead>
<tr>
<th>* RET</th>
<th>Return from subroutine</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Syntax</strong></td>
<td>RET</td>
</tr>
<tr>
<td><strong>Operation</strong></td>
<td>@SP→ PC  &lt;br&gt; SP + 2 → SP</td>
</tr>
<tr>
<td><strong>Emulation</strong></td>
<td>MOV @SP+,PC</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>The return address pushed onto the stack by a CALL instruction is moved to the program counter. The program continues at the code address following the subroutine call.</td>
</tr>
<tr>
<td><strong>Status Bits</strong></td>
<td>Status bits are not affected.</td>
</tr>
</tbody>
</table>
**RETIR**

**Return from interrupt**

**Syntax**

RETIR

**Operation**

TOS → SR  
SP + 2 → SP  
TOS → PC  
SP + 2 → SP

**Description**

The status register is restored to the value at the beginning of the interrupt service routine by replacing the present SR contents with the TOS contents. The stack pointer (SP) is incremented by two.

The program counter is restored to the value at the beginning of interrupt service. This is the consecutive step after the interrupted program flow. Restoration is performed by replacing the present PC contents with the TOS memory contents. The stack pointer (SP) is incremented.

**Status Bits**

N: restored from system stack  
Z: restored from system stack  
C: restored from system stack  
V: restored from system stack

**Mode Bits**

OSCOFF, CPUOFF, and GIE are restored from system stack.

**Example**

Figure 3–13 illustrates the main program interrupt.

*Figure 3–13. Main Program Interrupt*
Instruction Set

* RLA[W]
  Rotate left arithmetically

* RLA.B
  Rotate left arithmetically

Syntax

RLA dst or RLA.W dst
RLA.B dst

Operation

C ← MSB ← MSB−1 .... LSB+1 ← LSB ← 0

Emulation

ADD dst,dst
ADD.B dst,dst

Description

The destination operand is shifted left one position as shown in Figure 3−14. The MSB is shifted into the carry bit (C) and the LSB is filled with 0. The RLA instruction acts as a signed multiplication by 2.

An overflow occurs if dst ≥ 04000h and dst < 0C000h before operation is performed: the result has changed sign.

Figure 3−14. Destination Operand—Arithmetic Shift Left

An overflow occurs if dst ≥ 040h and dst < 0C0h before the operation is performed: the result has changed sign.

Status Bits

N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Loaded from the MSB
V: Set if an arithmetic overflow occurs:
   the initial value is 04000h ≤ dst < 0C000h; reset otherwise
Set if an arithmetic overflow occurs:
   the initial value is 040h ≤ dst < 0C0h; reset otherwise

Mode Bits

OSCOFF, CPUOFF, and GIE are not affected.

Example

R7 is multiplied by 2.

RLA R7 ; Shift left R7 (× 2)

Example

The low byte of R7 is multiplied by 4.

RLA.B R7 ; Shift left low byte of R7 (× 2)
RLA.B R7 ; Shift left low byte of R7 (× 4)

Note: RLA Substitution

The assembler does not recognize the instruction:

RLA @R5+, RLA.B @R5+, or RLA(.B) @R5

It must be substituted by:

ADD @R5+,−2(R5) ADD.B @R5+,−1(R5) or ADD(.B) @R5
* RLC[W]  Rotate left through carry
* RLC.B  Rotate left through carry

**Syntax**
RLC   dst or RLC.W   dst
RLC.B  dst

**Operation**
C <- MSB <- MSB−1 .... LSB+1 <- LSB <- C

**Emulation**
ADDC   dst,dst

**Description**
The destination operand is shifted left one position as shown in Figure 3–15. The carry bit (C) is shifted into the LSB and the MSB is shifted into the carry bit (C).

*Figure 3–15. Destination Operand—Carry Left Shift*

---

**Status Bits**
- N: Set if result is negative, reset if positive
- Z: Set if result is zero, reset otherwise
- C: Loaded from the MSB
- V: Set if an arithmetic overflow occurs
  - the initial value is 04000h ≤ dst < 0C000h; reset otherwise
  - Set if an arithmetic overflow occurs:
    - the initial value is 040h ≤ dst < 0C0h; reset otherwise

**Mode Bits**
OSCOFF, CPUOFF, and GIE are not affected.

**Example**
R5 is shifted left one position.

RLC   R5   ; (R5 x 2) + C -> R5

**Example**
The input P1IN.1 information is shifted into the LSB of R5.

BIT.B #2,&P1IN   ; Information -> Carry
RLC   R5   ; Carry=P0in.1 -> LSB of R5

**Example**
The MEM(LEO) content is shifted left one position.

RLC.B  LEO   ; Mem(LEO) x 2 + C -> Mem(LEO)

---

**Note: RLC and RLC.B Substitution**
The assembler does not recognize the instruction:
RLC @R5+,   RLC.B @R5+, or RLC(.B) @R5
It must be substituted by:
ADDC @R5+,−2(R5)  ADDC.B @R5+,−1(R5) or ADDC(.B) @R5
**Instruction Set**

**RRA.W**
Rotate right arithmetically

**RRA.B**
Rotate right arithmetically

**Syntax**

<table>
<thead>
<tr>
<th>RRA dst</th>
<th>RRA.W dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRA.B dst</td>
<td></td>
</tr>
</tbody>
</table>

**Operation**

MSB → MSB, MSB → MSB−1, ... LSB+1 → LSB, LSB → C

**Description**
The destination operand is shifted right one position as shown in Figure 3−16. The MSB is shifted into the MSB, the MSB is shifted into the MSB−1, and the LSB+1 is shifted into the LSB.

**Figure 3−16. Destination Operand—Arithmetic Right Shift**

![](image)

**Status Bits**

N: Set if result is negative, reset if positive
Z: Set if result is zero, reset otherwise
C: Loaded from the LSB
V: Reset

**Mode Bits**

OSCOFF, CPUOFF, and GIE are not affected.

**Example**

R5 is shifted right one position. The MSB retains the old value. It operates equal to an arithmetic division by 2.

```
RRA R5 ; R5/2 → R5
; The value in R5 is multiplied by 0.75 (0.5 + 0.25).
```

```
PUSH R5 ; Hold R5 temporarily using stack
RRA R5 ; R5 × 0.5 → R5
ADD @SP+,R5 ; R5 × 0.5 + R5 = 1.5 × R5 → R5
RRA R5 ; (1.5 × R5) × 0.5 = 0.75 × R5 → R5
......
```

**Example**

The low byte of R5 is shifted right one position. The MSB retains the old value. It operates equal to an arithmetic division by 2.

```
RRA.B R5 ; R5/2 → R5: operation is on low byte only
; High byte of R5 is reset
PUSH.B R5 ; R5 × 0.5 → TOS
RRA.B @SP ; TOS × 0.5 = 0.5 × R5 × 0.5 = 0.25 × R5 → TOS
ADD.B @SP+,R5 ; R5 × 0.5 + R5 × 0.25 = 0.75 × R5 → R5
......
```
### RRC[.W]
- **Operation**: Rotate right through carry

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRC dst</td>
<td>The carry bit (C) is shifted into the MSB, the LSB is shifted into the carry bit (C).</td>
</tr>
<tr>
<td>RRC.W dst</td>
<td></td>
</tr>
</tbody>
</table>

### Status Bits
- N: Set if result is negative, reset if positive
- Z: Set if result is zero, reset otherwise
- C: Loaded from the LSB
- V: Reset

### Mode Bits
- OSCOFF, CPUOFF, and GIE are not affected.

### Example
- R5 is shifted right one position. The MSB is loaded with 1.
  ```
  SETC ; Prepare carry for MSB
  RRC R5 ; R5/2 + 8000h −> R5
  ```

### Example
- R5 is shifted right one position. The MSB is loaded with 1.
  ```
  SETC ; Prepare carry for MSB
  RRC.B R5 ; R5/2 + 80h −> R5; low byte of R5 is used
  ```
* SBC[W]  Subtract source and borrow/.NOT. carry from destination
* SBC.B  Subtract source and borrow/.NOT. carry from destination

Syntax

SBC       dst   or   SBC.W  dst
SBC.B     dst

Operation

dst + 0FFFFh + C -> dst
dst + 0FFh + C -> dst

Emulation

SUBC       #0,dst
SUBC.B     #0,dst

Description

The carry bit (C) is added to the destination operand minus one. The previous contents of the destination are lost.

Status Bits

N:  Set if result is negative, reset if positive
Z:  Set if result is zero, reset otherwise
C:  Set if there is a carry from the MSB of the result, reset otherwise.
    Set to 1 if no borrow, reset if borrow.
V:  Set if an arithmetic overflow occurs, reset otherwise.

Mode Bits

OSCOFF, CPUOFF, and GIE are not affected.

Example

The 16-bit counter pointed to by R13 is subtracted from a 32-bit counter pointed to by R12.

SUB       @R13,0(R12) ; Subtract LSDs
SBC       2(R12)    ; Subtract carry from MSD

Example

The 8-bit counter pointed to by R13 is subtracted from a 16-bit counter pointed to by R12.

SUB.B     @R13,0(R12) ; Subtract LSDs
SBC.B     1(R12)     ; Subtract carry from MSD

Note:  Borrow Implementation.

The borrow is treated as a .NOT. carry:

<table>
<thead>
<tr>
<th>Borrow</th>
<th>Carry bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>No</td>
<td>1</td>
</tr>
</tbody>
</table>
**SETC**

Set carry bit

**Syntax**

SETC

**Operation**

1 → C

**Emulation**

BIS #1,SR

**Description**

The carry bit (C) is set.

**Status Bits**

N: Not affected
Z: Not affected
C: Set
V: Not affected

**Mode Bits**

OSCOFF, CPUOFF, and GIE are not affected.

**Example**

Emulation of the decimal subtraction:
Subtract R5 from R6 decimally
Assume that R5 = 03987h and R6 = 04137h

DSUB

ADD #06666h,R5 ; Move content R5 from 0–9 to 6–0Fh
; R5 = 03987h + 06666h = 09FEDh
INV R5 ; Invert this (result back to 0–9)
; R5 = .NOT. R5 = 06012h
SETC ; Prepare carry = 1
DADD R5,R6 ; Emulate subtraction by addition of:
; (010000h − R5 − 1)
; R6 = R6 + R5 + 1
; R6 = 0150h
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Syntax</th>
<th>Operation</th>
<th>Emulation</th>
<th>Status Bits</th>
<th>Mode Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETN</td>
<td>The negative bit (N) is set.</td>
<td>SETN</td>
<td>1 (\rightarrow) N</td>
<td>BIS #4,SR</td>
<td>N: Set</td>
<td>OSCOFF, CPUOFF, and GIE are not affected.</td>
</tr>
</tbody>
</table>

**Status Bits**
- **N**: Set
- **Z**: Not affected
- **C**: Not affected
- **V**: Not affected
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>* SETZ</td>
<td>Set zero bit</td>
</tr>
</tbody>
</table>

**Syntax**

```
SETZ
```

**Operation**

```
1 -> Z
```

**Emulation**

```
BIS #2,SR
```

**Description**

The zero bit (Z) is set.

**Status Bits**

- N: Not affected
- Z: Set
- C: Not affected
- V: Not affected

**Mode Bits**

OSCOFF, CPUOFF, and GIE are not affected.
SUB[.W]  Subtract source from destination
SUB.B  Subtract source from destination

Syntax  SUB src,dst or SUB.W src,dst
        SUB.B src,dst

Operation  dst + .NOT.src + 1 -> dst
        or
        [(dst − src -> dst)]

Description  The source operand is subtracted from the destination operand by adding the
              source operand’s 1s complement and the constant 1. The source operand is
              not affected. The previous contents of the destination are lost.

Status Bits  N:  Set if result is negative, reset if positive
Z:  Set if result is zero, reset otherwise
C:  Set if there is a carry from the MSB of the result, reset otherwise.
    Set to 1 if no borrow, reset if borrow.
V:  Set if an arithmetic overflow occurs, otherwise reset

Mode Bits  OSCOFF, CPUOFF, and GIE are not affected.

Example  See example at the SBC instruction.

Example  See example at the SBC.B instruction.

Note:  Borrow Is Treated as a .NOT.

<table>
<thead>
<tr>
<th>Borrow</th>
<th>Carry bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>No</td>
<td>1</td>
</tr>
</tbody>
</table>
---|---
Subtract source and borrow/.NOT. carry from destination
Subtract source and borrow/.NOT. carry from destination

**Syntax**

<table>
<thead>
<tr>
<th>SUBC</th>
<th>src,dst</th>
<th>SUBC[W]</th>
<th>src,dst</th>
<th>or</th>
</tr>
</thead>
<tbody>
<tr>
<td>SBB</td>
<td>src,dst</td>
<td>SBB[W]</td>
<td>src,dst</td>
<td></td>
</tr>
<tr>
<td>SUBC[B]</td>
<td>src,dst</td>
<td>SBB[B]</td>
<td>src,dst</td>
<td></td>
</tr>
</tbody>
</table>

**Operation**

dst + .NOT.src + C -> dst
or
(dst – src – 1 + C -> dst)

**Description**
The source operand is subtracted from the destination operand by adding the source operand’s 1s complement and the carry bit (C). The source operand is not affected. The previous contents of the destination are lost.

**Status Bits**

- **N**: Set if result is negative, reset if positive.
- **Z**: Set if result is zero, reset otherwise.
- **C**: Set if there is a carry from the MSB of the result, reset otherwise.
  - Set to 1 if no borrow, reset if borrow.
- **V**: Set if an arithmetic overflow occurs, reset otherwise.

**Mode Bits**
OSCOFF, CPUOFF, and GIE are not affected.

**Example**

Two floating point mantissas (24 bits) are subtracted. LSBs are in R13 and R10, MSBs are in R12 and R9.

```
SUB.W R13,R10 ; 16-bit part, LSBs
SUBC.B R12,R9  ; 8-bit part, MSBs
```

**Example**
The 16-bit counter pointed to by R13 is subtracted from a 16-bit counter in R10 and R11(MSD).

```
SUB.B @R13+,R10 ; Subtract LSDs without carry
SUBC.B @R13,R11 ; Subtract MSDs with carry
... ; resulting from the LSDs
```

**Note:** Borrow Implementation

<table>
<thead>
<tr>
<th>The borrow is treated as a .NOT. carry</th>
<th>Borrow</th>
<th>Carry bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>No</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**Instruction Set**

### SWPB

**Syntax**

SWPB dst

**Operation**

Bits 15 to 8 $\leftrightarrow$ bits 7 to 0

**Description**

The destination operand high and low bytes are exchanged as shown in Figure 3–18.

**Status Bits**

Status bits are not affected.

**Mode Bits**

OSCOFF, CPUOFF, and GIE are not affected.

#### Figure 3–18. Destination Operand Byte Swap

![Diagram of byte swap](image)

**Example**

```
MOV #040BFh,R7 ; 0100000010111111 $\rightarrow$ R7
SWPB R7 ; 101111101000000 in R7
```

**Example**

The value in R5 is multiplied by 256. The result is stored in R5,R4.

```
SWPB R5 ;
MOV R5,R4 ;Copy the swapped value to R4
BIC #0FF00h,R5 ;Correct the result
BIC #00FFh,R4 ;Correct the result
```
SXT  

Extend Sign

Syntax 

SXT    dst

Operation 

Bit 7 -> Bit 8 ......... Bit 15

Description 

The sign of the low byte is extended into the high byte as shown in Figure 3–19.

Status Bits 

N: Set if result is negative, reset if positive  
Z: Set if result is zero, reset otherwise  
C: Set if result is not zero, reset otherwise (.NOT. Zero)  
V: Reset

Mode Bits 

OSCOFF, CPUOFF, and GIE are not affected.

Figure 3–19. Destination Operand Sign Extension

Example 

R7 is loaded with the P1IN value. The operation of the sign-extend instruction expands bit 8 to bit 15 with the value of bit 7.  
R7 is then added to R6.

MOV.B &P1IN,R7  ; P1IN = 080h: ...... ...... 1000 0000  
SXT         R7  ; R7 = 0FF80h: 1111 1111 1000 0000
### Instruction Set

**TST[W]**
- Test destination

**TST.B**
- Test destination

#### Syntax
- `TST dst` or `TST.W dst`
- `TST.B dst`

#### Operation
- `dst + 0FFFFh + 1`
- `dst + 0FFh + 1`

#### Emulation
- `CMP #0,dst`
- `CMP .B #0,dst`

#### Description
The destination operand is compared with zero. The status bits are set according to the result. The destination is not affected.

#### Status Bits
- **N**: Set if destination is negative, reset if positive
- **Z**: Set if destination contains zero, reset otherwise
- **C**: Set
- **V**: Reset

#### Mode Bits
OSCOFF, CPUOFF, and GIE are not affected.

#### Example
R7 is tested. If it is negative, continue at R7NEG; if it is positive but not zero, continue at R7POS.

```assembly
TST R7          ; Test R7
JN R7NEG        ; R7 is negative
JZ R7ZERO       ; R7 is zero
```

```
R7POS ......    ; R7 is positive but not zero
R7NEG ......    ; R7 is negative
R7ZERO ......   ; R7 is zero
```

#### Example
The low byte of R7 is tested. If it is negative, continue at R7NEG; if it is positive but not zero, continue at R7POS.

```assembly
TST.B R7        ; Test low byte of R7
JN R7NEG        ; Low byte of R7 is negative
JZ R7ZERO       ; Low byte of R7 is zero
```

```
R7POS ......    ; Low byte of R7 is positive but not zero
R7NEG ......    ; Low byte of R7 is negative
R7ZERO ......   ; Low byte of R7 is zero
```
### XOR[.W]
Exclusive OR of source with destination

### XOR.B
Exclusive OR of source with destination

#### Syntax
- `XOR src,dst` or `XOR.W src,dst`
- `XOR.B src,dst`

#### Operation
- `src .XOR. dst -> dst`

#### Description
The source and destination operands are exclusive ORed. The result is placed into the destination. The source operand is not affected.

#### Status Bits
- **N**: Set if result MSB is set, reset if not set
- **Z**: Set if result is zero, reset otherwise
- **C**: Set if result is not zero, reset otherwise ( = .NOT. Zero)
- **V**: Set if both operands are negative

#### Mode Bits
OSCOFF, CPUOFF, and GIE are not affected.

#### Example
The bits set in R6 toggle the bits in the RAM word TONI.
```
XOR R6,TONI ; Toggle bits of word TONI on the bits set in R6
```

#### Example
The bits set in R6 toggle the bits in the RAM byte TONI.
```
XOR.B R6,TONI ; Toggle bits of byte TONI on the bits set in low byte of R6
```

#### Example
Reset to 0 those bits in low byte of R7 that are different from bits in RAM byte EDE.
```
XOR.B EDE,R7 ; Set different bit to “1s”
INV.B R7 ; Invert Lowbyte, Highbyte is 0h
```