Chapter 4

Logical instructions

\[
\text{and } \text{wb, ws, wd } \Rightarrow \text{ Reg }[\text{wb}] \land \text{ Reg }[\text{ws}] \rightarrow \text{ Reg }[\text{wd}] \text{ (16 bits are transferred)}
\]

\[
\text{and.b } \text{wb, ws, wd } \Rightarrow \text{ Reg }[\text{wb}] \land \text{ Reg }[\text{ws}] \rightarrow \text{ Reg }[\text{wd}] \text{ (8 bits to LSB of wd)}
\]

\[
\text{and } \text{wb, #lit5, wd } \Rightarrow \text{ Reg }[\text{wb}] \land \#\text{lit5} \rightarrow \text{ Reg }[\text{wd}] \text{ (16 bits are transferred)}
\]

\[
\text{and.b } \text{wb, #lit5, wd } \Rightarrow \text{ Reg }[\text{wb}] \land \#\text{lit5} \rightarrow \text{ Reg }[\text{wd}] \text{ (8 bits to LSB of wd)}
\]

\[
\text{and } \#\text{lit10, wn } \Rightarrow \text{ Reg }[\text{wn}] \land \#\text{lit10} \rightarrow \text{ Reg }[\text{wn}] \text{ (16 bits are transferred)}
\]

\[
\text{and.b } \#\text{lit10, wn } \Rightarrow \text{ Reg }[\text{wn}] \land \#\text{lit10} \rightarrow \text{ Reg }[\text{wn}] \text{ (8 bits to LSB of wn)}
\]

\[
\text{and } f \Rightarrow \text{ Reg }[\text{w0}] \land \text{ mem } (f) \rightarrow \text{ mem } (f) \text{ (16 bits are transferred)}
\]

\[
\text{and.b } f \Rightarrow \text{ Reg }[\text{w0}] \land \text{ mem } (f) \rightarrow \text{ mem } (f) \text{ (8 bits to LSB of data memory)}
\]

\[
\text{and } f, \text{ wreg } \Rightarrow \text{ Reg }[\text{w0}] \land \text{ mem } (f) \rightarrow \text{ Reg }[\text{w0}] \text{ (16 bits are transferred)}
\]

\[
\text{and.b } f, \text{ wreg } \Rightarrow \text{ Reg }[\text{w0}] \land \text{ mem } (f) \rightarrow \text{ Reg }[\text{w0}] \text{ (8 bits to LSB of w0)}
\]
ior  wb, ws, wd  ⇒  Reg [wb] | Reg [ws] → Reg [wd] (16 bits are transferred)
ior.b wb, ws, wd  ⇒  Reg [wb] | Reg [ws] → Reg [wd] (8 bits to LSB of wd)

ior  wb, #lit5, wd  ⇒  Reg [wb] | #lit5 → Reg [wd] (16 bits are transferred)
ior.b wb, #lit5, wd  ⇒  Reg [wb] | #lit5 → Reg [wd] (8 bits to LSB of wd)

ior  #lit10, wn  ⇒  Reg [wn] | #lit10 → Reg [wn] (16 bits are transferred)
ior.b #lit10, wn  ⇒  Reg [wn] | #lit10 → Reg [wn] (8 bits to LSB of wn)

ior  f  ⇒  Reg [w0] | mem (f) → mem (f) (16 bits are transferred)
ior.b f  ⇒  Reg [w0] | mem (f) → mem (f) (8 bits to LSB of data memory)

ior  f , wreg  ⇒  Reg [w0] | mem (f) → Reg [w0] (16 bits are transferred)
ior.b f , wreg  ⇒  Reg [w0] | mem (f) → Reg [w0] (8 bits to LSB of w0)
xor wb, ws, wd ⇒ Reg[wb]^Reg[ws] → Reg[wd] (16 bits are transferred)
xor.b wb, ws, wd ⇒ Reg[wb]^Reg[ws] → Reg[wd] (8 bits to LSB of wd)

xor wb, #lit5, wd ⇒ Reg[wb]^#lit5 → Reg[wd] (16 bits are transferred)
xor.b wb, #lit5, wd ⇒ Reg[wb]^#lit5 → Reg[wd] (8 bits to LSB of wd)

xor #lit10, wn ⇒ Reg[wn]^#lit10 → Reg[wn] (16 bits are transferred)
xor.b #lit10, wn ⇒ Reg[wn]^#lit10 → Reg[wn] (8 bits to LSB of wn)

xor f ⇒ Reg[w0]^mem(f) → mem(f) (16 bits are transferred)
xor.b f ⇒ Reg[w0]^mem(f) → mem(f) (8 bits to LSB of data memory)

xor f, wreg ⇒ Reg[w0]^mem(f) → Reg[w0] (16 bits are transferred)
xor.b f, wreg ⇒ Reg[w0]^mem(f) → Reg[w0] (8 bits to LSB of w0)
com ws, wd ⇒ ¬ Reg [ws] → Reg [wd] (16 bits are transferred)
com.b ws, wd ⇒ ¬ Reg [ws] → Reg [wd] (LSB is transferred)

com f ⇒ ¬ mem (f) → mem (f) (16 bits are transferred)
com.b f ⇒ ¬ mem (f) → mem (f) (LSB is transferred)

com f, wreg ⇒ ¬ mem (f) → Reg [w0] (16 bits are transferred)
com.b f, wreg ⇒ ¬ mem (f) → Reg [w0] (LSB is transferred)
clr  wd  ⇒  0x0000 → Reg [wd] (16 bits are cleared)
clr.b wd  ⇒  0x0000 → Reg [wd] (LSB is cleared)

clr  wreg  ⇒  0x0000 → Reg [w0] (16 bits are cleared)
clr.b wreg  ⇒  0x0000 → Reg [w0] (LSB is cleared)

clr  f  ⇒  0x0000 → mem (f) (16 bits are cleared)
clr.b f  ⇒  0x0000 → mem (f) (LSB is cleared)
setm  wd  \Rightarrow  0xFFFF \rightarrow \text{Reg }[\text{wd}] \ (16 \text{ bits are set})
setm.b \ wd  \Rightarrow  0xFFFF \rightarrow \text{Reg }[\text{wd}] \ (\text{LSB is set})

setm  \ wreg  \Rightarrow  0xFFFF \rightarrow \text{Reg }[\text{w0}] \ (16 \text{ bits are set})
setm.b \ wreg  \Rightarrow  0xFFFF \rightarrow \text{Reg }[\text{w0}] \ (\text{LSB is set})

setm \ f  \Rightarrow  0xFFFF \rightarrow \text{mem } (f) \ (16 \text{ bits are set})
setm.b \ f  \Rightarrow  0xFFFF \rightarrow \text{mem } (f) \ (\text{LSB is set})

\textbf{NOTE:} Instructions containing ws and wd are eligible for \textbf{indirect} addressing.
**Example:** Let $\text{mem (0x0800)} = 0x2C$ (data memory is byte addressable)

(a) Execute $\text{mem (0x0800)} \& 0x0F \rightarrow \text{mem (0x0800)}$

mov.b #0x0F, w0 \implies \#0x0F \rightarrow \text{Reg [w0] (only to LSB of w0)}
and.b 0x0800 \implies \text{mem (0x0800)} \& \text{Reg [w0]} \rightarrow \text{mem (0x0800)} (\text{only LSB})
0010 1100 \& 0000 1111 \rightarrow 0000 1100 = 0x0C \rightarrow \text{mem (0x0800)}$

(b) Execute $\text{mem (0x0800)} \oplus 0x0F \rightarrow \text{mem (0x0800)}$

mov.b #0x0F, w0 \implies \#0x0F \rightarrow \text{Reg [w0] (only to LSB of w0)}
xor.b 0x0800 \implies \text{mem (0x0800)} \oplus \text{Reg [w0]} \rightarrow \text{mem (0x0800)} (\text{only LSB})
0010 1100 \oplus 0000 1111 \rightarrow 0010 0011 = 0x23 \rightarrow \text{mem (0x0800)}$

(c) Execute $\sim \text{mem (0x0800)} \rightarrow \text{mem (0x0800)}$

com.b 0x0800 \implies \sim \text{mem (0x0800)} \rightarrow \text{mem (0x0800)} (\text{only LSB})
$\sim 0010 1100 \rightarrow 1101 0011 = 0xD3 \rightarrow \text{mem (0x0800)}$

(d) Execute $\#0x00 \rightarrow \text{mem (0x0800)}$

clr.b 0x0800 \implies \#0x00 \rightarrow \text{mem (0x0800)} (\text{only LSB})$
Logical instructions dealing with a single bit

\[
\begin{align*}
\text{bset} &\quad w, \ #\text{bitj} \Rightarrow \text{bitj of Reg} [w] = 1 \ (\text{for all 16 bits}) \\
\text{bset.b} &\quad w, \ #\text{bitj} \Rightarrow \text{bitj of Reg} [w] = 1 \ (\text{for only least significant 8 bits}) \\
\text{bset} &\quad f, \ #\text{bitj} \Rightarrow \text{bitj of mem} (f) = 1 \ (\text{for all 16 bits}) \\
\text{bset.b} &\quad f, \ #\text{bitj} \Rightarrow \text{bitj of mem} (f) = 1 \ (\text{for only least significant 8 bits}) \\
\text{bclr} &\quad w, \ #\text{bitj} \Rightarrow \text{bitj of Reg} [w] = 0 \ (\text{for all 16 bits}) \\
\text{bclr.b} &\quad w, \ #\text{bitj} \Rightarrow \text{bitj of Reg} [w] = 0 \ (\text{for only least significant 8 bits}) \\
\text{bclr} &\quad f, \ #\text{bitj} \Rightarrow \text{bitj of mem} (f) = 0 \ (\text{for all 16 bits}) \\
\text{bclr.b} &\quad f, \ #\text{bitj} \Rightarrow \text{bitj of mem} (f) = 0 \ (\text{for only least significant 8 bits}) \\
\text{btg} &\quad w, \ #\text{bitj} \Rightarrow \neg \text{bitj of Reg} [w] = \text{bitj of Reg} [w] \ (\text{for all 16 bits}) \\
\text{btg.b} &\quad w, \ #\text{bitj} \Rightarrow \neg \text{bitj of Reg} [w] = \text{bitj of Reg} [w] \ (\text{for only LS 8 bits}) \\
\text{btg} &\quad f, \ #\text{bitj} \Rightarrow \neg \text{bitj of mem} (f) = \text{bitj of mem} (f) \ (\text{for all 16 bits}) \\
\text{btg.b} &\quad f, \ #\text{bitj} \Rightarrow \neg \text{bitj of mem} (f) = \text{bitj of mem} (f) \ (\text{for only LS 8 bits})
\end{align*}
\]
Example: Let Reg \([w1]\) = 0x00F0 and mem (0x4F01) = 0xFF0F

(a) Execute bset \(w1, \#0\) \(\Rightarrow\) bit 0 of Reg \([w1]\) = 1 \(\Rightarrow\) Reg \([w1]\) = 0x00F1

(b) Execute bset.b \(0x4F01, \#4\) \(\Rightarrow\) bit 4 of mem \((0x4F01)\) = 1
\hspace{1cm} mem \((0x4F01)\) = 0xFF1F

(c) Execute bclr.b \(0x4F01, \#1\) \(\Rightarrow\) bit 1 of mem \((0x4F01)\) = 0
\hspace{1cm} mem \((0x4F01)\) = 0xFF0D

(d) Execute btg \(w1, \#15\) \(\Rightarrow\) \(~\) bit 15 of Reg \([w1]\) = bit 15 of Reg \([w1]\)
\hspace{1cm} Reg \([w1]\) = 0x80F0
**Status Register (SR)**

Status register shows the status of the machine output due to an instruction after it is executed.

<table>
<thead>
<tr>
<th>15</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>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>DC</td>
<td>IPL2</td>
<td>IPL1</td>
<td>IPL0</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>RA</td>
<td>N</td>
<td>OV</td>
<td>Z</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td>C</td>
</tr>
</tbody>
</table>

- **DC** = sets to 1 if decimal carry occurs
- **IPL [2:0]** = Interrupt Priority Level bits set by the user
- **RA** = sets to 1 if while a repeat/loop instruction takes place
- **N** = sets to 1 if the instruction produces a negative result
- **OV** = sets to 1 if the instruction produces an overflow bit
- **Z** = sets to 1 if the instruction produces a zero result
- **C** = sets to 1 if the instruction produces a carry-out bit
Example: Observe what status bits are produced after these operations

(a) Determine C, Z, OV and N flags of the SR after 0x01 + 0xFF

\[
\begin{array}{c}
0x01 \\
+ \\
0xFF \\
\hline \\
0x00 \\
\end{array}
\rightarrow
\begin{cases}
C = 1 \\
Z = 1 \\
OV = 1 \\
N = 0
\end{cases}
\]

(b) Determine C, Z, OV and N flags of the SR after 0x20 + 0xF0

\[
\begin{array}{c}
0x20 \\
+ \\
0xF0 \\
\hline \\
0x10 \\
\end{array}
\rightarrow
\begin{cases}
C = 1 \\
Z = 0 \\
OV = 1 \\
N = 0
\end{cases}
\]
(c) Determine C, Z, OV and N flags of the SR after 0x80 + 0x7F

\[
\begin{array}{c}
0x80 \\
+ \ \\
0x7F \\
\hline \\
0xFF \\
\rightarrow \\
\begin{array}{l}
C = 0 \\
Z = 0 \\
OV = 0 \\
N = 0
\end{array}
\end{array}
\]

(d) Determine C, Z, OV and N flags of the SR after 0x00 + 0x00

\[
\begin{array}{c}
0x00 \\
+ \ \\
0x00 \\
\hline \\
0x00 \\
\rightarrow \\
\begin{array}{l}
C = 0 \\
Z = 1 \\
OV = 0 \\
N = 0
\end{array}
\end{array}
\]
Logical shift operations

(a) Logical Shift Right (lsr) operations

\[
\begin{align*}
\text{lsr} & \quad \text{ws, wd} \quad \Rightarrow \quad \text{Reg} [\text{ws}] \gg 1 \text{ bit} \rightarrow \text{Reg} [\text{wd}] \quad (16 \text{ bits}) \\
\text{lsr.b} & \quad \text{ws, wd} \quad \Rightarrow \quad \text{Reg} [\text{ws}] \gg 1 \text{ bit} \rightarrow \text{Reg} [\text{wd}] \quad (\text{LS} \ 8 \text{ bits}) \\
\text{lsr} & \quad f \quad \Rightarrow \quad \text{mem} (f) \gg 1 \text{ bit} \rightarrow \text{mem} (f) \quad (16 \text{ bits}) \\
\text{lsr.b} & \quad f \quad \Rightarrow \quad \text{mem} (f) \gg 1 \text{ bit} \rightarrow \text{mem} (f) \quad (\text{LS} \ 8 \text{ bits}) \\
\text{lsr} & \quad f, \text{wreg} \quad \Rightarrow \quad \text{mem} (f) \gg 1 \text{ bit} \rightarrow \text{Reg} [\text{w0}] \quad (16 \text{ bits}) \\
\text{lsr.b} & \quad f, \text{wreg} \quad \Rightarrow \quad \text{mem} (f) \gg 1 \text{ bit} \rightarrow \text{Reg} [\text{w0}] \quad (\text{LS} \ 8 \text{ bits}) \\
\text{lsr} & \quad \text{wb, ws, wd} \quad \Rightarrow \quad \text{Reg} [\text{wb}] \gg \text{Reg} [\text{ws} (4:0)] \rightarrow \text{Reg} [\text{wd}] \quad (\text{max} \ 15 \text{ bits}) \\
\text{lsr} & \quad \text{wb, #lit4, wd} \quad \Rightarrow \quad \text{Reg} [\text{wb}] \gg \#\text{lit4} \rightarrow \text{Reg} [\text{wd}] \quad (\text{max} \ 15 \text{ bits})
\end{align*}
\]

Logical Shift Right operations shift the contents of a register or memory by 1 bits (the first 3 instruction sets) or multiple bits (the last 2 instructions) to the right. When right shift occurs, 0 is fed to the register from its MSB side as shown below.

![](image)

The bit comes out of the LSB position becomes the carry-out bit and may set the C-flag of the SR.
(b) Logical Shift Left (sl) operations

\[
\begin{align*}
\text{sl} & \quad \text{ws, wd} \Rightarrow \text{Reg [ws] } \ll 1 \text{ bit } \rightarrow \text{Reg [wd]} \quad \text{(for all 16 bits)} \\
\text{sl.b} & \quad \text{ws, wd} \Rightarrow \text{Reg [ws] } \ll 1 \text{ bit } \rightarrow \text{Reg [wd]} \quad \text{(for only LS 8 bits)} \\
\text{sl} & \quad f \Rightarrow \text{mem (f) } \ll 1 \text{ bit } \rightarrow \text{mem (f)} \quad \text{(for all 16 bits)} \\
\text{sl.b} & \quad f \Rightarrow \text{mem (f) } \ll 1 \text{ bit } \rightarrow \text{mem (f)} \quad \text{(for only LS 8 bits)} \\
\text{sl} & \quad f, \text{wreg} \Rightarrow \text{mem (f) } \ll 1 \text{ bit } \rightarrow \text{Reg [w0]} \quad \text{(for all 16 bits)} \\
\text{sl.b} & \quad f, \text{wreg} \Rightarrow \text{mem (f) } \ll 1 \text{ bit } \rightarrow \text{Reg [w0]} \quad \text{(for only LS 8 bits)} \\
\text{sl} & \quad \text{wb, ws, wd} \Rightarrow \text{Reg [wb] } \ll \text{Reg [ws (4:0)]} \rightarrow \text{Reg [wd]} \quad \text{(up to 15 bits of shift)} \\
\text{sl} & \quad \text{wb, #lit4, wd} \Rightarrow \text{Reg [wb] } \ll \#\text{lit4} \rightarrow \text{Reg [wd]} \quad \text{(up to 15 bits of shift)}
\end{align*}
\]

Logical Shift Left operations shift the contents of a register or memory by 1 bits (the first 3 instruction sets) or multiple bits (the last 2 instructions) to the left. When left shift occurs, 0 is fed to the register from its LSB side as shown below.

```
C-\hspace{1cm}15\hspace{1cm}..\hspace{1cm}0
\begin{array}{c}
\text{MSB}
\end{array}
```

The bit comes out of the MSB position becomes the carry-out bit and may set the C-flag of the SR.
**Example:** Execute `lsr.b 0x1234` where memory location 0x1234 contains 0xFFFF

Thus:

<table>
<thead>
<tr>
<th>15</th>
<th>1111</th>
<th>1111</th>
<th>1111</th>
<th>1111</th>
<th>0</th>
<th>before</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1234</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>15</th>
<th>1111</th>
<th>1111</th>
<th>0111</th>
<th>1111</th>
<th>0</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1234</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After the operation, `mem (0x1234) = 0xFF7F` and the C-flag = 1

**Example:** Execute `sl w1, #15, w2` where `w1 0xFFFF`

Here, `Reg [w1] << 15` bits → `Reg [w2]`

<table>
<thead>
<tr>
<th>15</th>
<th>1111</th>
<th>1111</th>
<th>1111</th>
<th>1111</th>
<th>0</th>
<th>before</th>
</tr>
</thead>
<tbody>
<tr>
<td>w1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>15</th>
<th>1000</th>
<th>0000</th>
<th>0000</th>
<th>0000</th>
<th>0</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td>w1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After the operation, `Reg [w2] = 0x8000` and the C-flag = 0 since multiple shifts do NOT affect the C-flag of the SR.
Rotate operations

(a) Rotate Right with Carry (rrc) operations

rrc ws, wd  ⇒  Reg [ws] Rotate Right Once with Carry → Reg [wd]  (for all 16 bits)
rrc.b ws, wd  ⇒  Reg [ws] Rotate Right Once with Carry → Reg [wd]  (only LSB)

rrc f  ⇒  mem (f) Rotate Right Once with Carry → mem (f)  (for all 16 bits)
rrc.b f  ⇒  mem (f) Rotate Right Once with Carry → mem (f)  (only LSB)

rrc f, wreg  ⇒  mem (f) Rotate Right Once with Carry → Reg (w0]  (for all 16 bits)
rrc.b f, wreg  ⇒  mem (f) Rotate Right Once with Carry → Reg [w0]  (only LSB)
(b) Rotate Right with No Carry (rrnc) operations

\[ \text{rrnc ws, wd} \Rightarrow \text{Reg [ws] Rotate Right Once w/ No Carry} \rightarrow \text{Reg [wd]} \text{ (for all 16 bits)} \]
\[ \text{rrnc.b ws, wd} \Rightarrow \text{Reg [ws] Rotate Right Once w/ No Carry} \rightarrow \text{Reg [wd]} \text{ (only LSB)} \]

\[ \text{rrnc f} \Rightarrow \text{mem (f) Rotate Right Once w/ No Carry} \rightarrow \text{mem (f)} \text{ (for all 16 bits)} \]
\[ \text{rrnc.b f} \Rightarrow \text{mem (f) Rotate Right Once w/ No Carry} \rightarrow \text{mem (f)} \text{ (only LSB)} \]

\[ \text{rrnc f, wreg} \Rightarrow \text{mem (f) Rotate Right Once w/ No Carry} \rightarrow \text{Reg (w0)} \text{ (for all 16 bits)} \]
\[ \text{rrnc.b f, wreg} \Rightarrow \text{mem (f) Rotate Right Once w/ No Carry} \rightarrow \text{Reg [w0]} \text{ (only LSB)} \]
(c) Rotate Left with Carry (rlc) operations

\[
\text{rlc ws, wd } \Rightarrow \text{ Reg [ws] Rotate Left Once with Carry } \rightarrow \text{ Reg [wd] (for all 16 bits)}
\]
\[
\text{rlc.b ws, wd } \Rightarrow \text{ Reg [ws] Rotate Left Once with Carry } \rightarrow \text{ Reg [wd] (only LSB)}
\]
\[
\text{rlc f } \Rightarrow \text{ mem (f) Rotate Left Once with Carry } \rightarrow \text{ mem (f) (for all 16 bits)}
\]
\[
\text{rlc.b f } \Rightarrow \text{ mem (f) Rotate Left Once with Carry } \rightarrow \text{ mem (f) (only LSB)}
\]
\[
\text{rlc f, wreg } \Rightarrow \text{ mem (f) Rotate Left Once with Carry } \rightarrow \text{ Reg [w0] (for all 16 bits)}
\]
\[
\text{rlc.b f, wreg } \Rightarrow \text{ mem (f) Rotate Left Once with Carry } \rightarrow \text{ Reg [w0] (only LSB)}
\]
(d) Rotate Left with No Carry (rlnc) operations

\[
\text{rlnc} \ ws, \ wd \rightarrow \ \text{Reg} [ws] \ \text{Rotate Left Once w/ No Carry} \rightarrow \ \text{Reg} [wd] \quad \text{(for all 16 bits)}
\]

\[
\text{rlnc.b} \ ws, \ wd \rightarrow \ \text{Reg} [ws] \ \text{Rotate Left Once w/ No Carry} \rightarrow \ \text{Reg} [wd] \quad \text{(only LSB)}
\]

\[
\text{rlnc} \ f \rightarrow \ \text{mem} (f) \ \text{Rotate Left Once w/ No Carry} \rightarrow \ \text{mem} (f) \quad \text{(for all 16 bits)}
\]

\[
\text{rlnc.b} \ f \rightarrow \ \text{mem} (f) \ \text{Rotate Left Once w/ No Carry} \rightarrow \ \text{mem} (f) \quad \text{(only LSB)}
\]

\[
\text{rlnc} \ f, \ w\text{reg} \rightarrow \ \text{mem} (f) \ \text{Rotate Left Once w/ No Carry} \rightarrow \ \text{Reg} (w0) \quad \text{(for all 16 bits)}
\]

\[
\text{rlnc.b} \ f, \ w\text{reg} \rightarrow \ \text{mem} (f) \ \text{Rotate Left Once w/ No Carry} \rightarrow \ \text{Reg} [w0] \quad \text{(only LSB)}
\]
**Example:** Assume the following:

Reg [w1] = 0x0FFF
C-flag of SR = 1

Execute

```
rrc  w1, w1  ⇒  Reg [w1] Rotate Right Once with Carry → Reg [w1]
```

Thus, Reg [w1] = 0x87FF after the operation.
Example: Assume the following:

\[
\text{mem (0x1A2A] = 0x0001}
\]

Execute

\[
\text{rlnc.b 0x1A2A} \Rightarrow \text{mem (0x1A2A) Rotate Left Once w/ No Carry} \Rightarrow \text{mem (0x1A2A)}
\]
Example: Convert the following C program into PIC24 Assembly

Assume the following program in C:
unsigned int i, n, p, k;
k = n + (i << 3) – p

In Assembly, the same program is written as:
i: .space 2
n: .space 2
p: .space 2
k: .space 2

    mov n, w0 ; Reg [w0] = mem (n)
    mov i, w1 ; Reg [w1] = mem (i)
    sl  w1, #3, w1 ; Reg [w1] << 3 bits → Reg [w1]
    add w0, w1, w0 ; n + (i << 3) → Reg [w0]
    mov p, w1 ; Reg [w1] = mem (p)
    sub w0, w1, w0 ; n + (i<<3) – p → Reg [w0]
    mov w0, k ; mem [k] = n + (i<<3) – p
Branch Bit Test Instructions

bra Z, <label>  ⇒  Branch to <label> location in the program if Z-flag = 1
bra NZ, <label> ⇒  Branch to <label> location in the program if Z-flag = 0
bra C, <label>  ⇒  Branch to <label> location in the program if C-flag = 1
bra NC, <label> ⇒  Branch to <label> location in the program if C-flag = 0
bra N, <label>  ⇒  Branch to <label> location in the program if N-flag = 1
bra NN, <label> ⇒  Branch to <label> location in the program if N-flag = 0
bra OV, <label> ⇒  Branch to <label> location in the program if OV-flag = 1
bra NOV, <label>⇒  Branch to <label> location in the program if OV-flag = 0
bra <label>  ⇒  Branch to <label> location in the program unconditionally
**Example:** Assume the following PIC24 Assembly program:

```
mov.b  #3, w1 ; #3 → Reg [w1] and Z-flag = 0
bra   NZ, loc1 ; if Z-flag = 0 after the previous operation then go to loc1
bra   loc2 ; unconditionally branch to loc2
.
.
loc1:
mov.b  w1, 0x1234 ; Reg [w1] → mem (0x1234)
.
.
loc2:
mov.b  w1, 0xABCD ; Reg [w1] → mem (0xABCD)
```
**Example:** Assume the following PIC24 Assembly program:

```assembly
mov 0x1234, w1 ; mem (0x1234) → Reg [w1]
mov 0xABCD, w2 ; mem (0xABCD) → Reg [w2]
sub w1, w2, w3 ; Reg [w1] – Reg [w2] → Reg [w3]
bra NZ, NotEqual ; if Z-flag = 0 after the previous instruction then go to Not-Equal
```

```
Not-Equal:
```

```assembly
```
```
Compare Instructions

- `cp wb, ws`  \(\Rightarrow\)  compare Reg [wb] with Reg [ws]
- `cp.b wb, ws`  \(\Rightarrow\)  compare the LSB of Reg [wb] with the LSB of Reg [ws]

- `cp wb, #lit5`  \(\Rightarrow\)  compare Reg [wb] with #lit5
- `cp.b wb, #lit5`  \(\Rightarrow\)  compare the LSB of Reg [wb] with #lit5

- `cp f`  \(\Rightarrow\)  compare mem (f) with Reg [w0]
- `cp.b f`  \(\Rightarrow\)  compare the LSB of mem (f) with Reg [w0]

- `cp0 ws`  \(\Rightarrow\)  compare Reg [ws] against 0
- `cp0.b ws`  \(\Rightarrow\)  compare the LSB of Reg [ws] against 0

- `cp0 f`  \(\Rightarrow\)  compare mem (f) against 0
- `cp0.b f`  \(\Rightarrow\)  compare the LSB of mem (f) against 0
**Unsigned Branch Instructions**

When a compare instruction is executed, the C and Z-flags of the SR are affected. A branch instruction can be placed right after the compare instruction to direct the program to the desired location.

- `bra GTU, <label>` ; if the result of the compare is greater than 0 then go to `<label>`
- `bra GEU, <label>` ; if the result of the compare is greater than or equal 0 then go to `<label>`
- `bra LTU, <label>` ; if the result of the compare is less than 0 then go to `<label>`
- `bra LEU, <label>` ; if the result of the compare is less than or equal 0 then go to `<label>`
**Example:** Write a PIC24 assembly code according to the following flow chart below:

- **k, j = 16 bit unsigned variables**
  - If $k > j$, go to if-body statements
  - Else, go to else-body statements

The assembly program is as follows:

```assembly
mov k, w1 ; mem (k) → Reg [w1]
mov j, w2 ; mem (j) → Reg [w2]
cp w1, w2 ; compare Reg [w1] with Reg [w2] using (Reg [w1] – Reg [2])
br a LEU, else-body ; if compare yields Reg [w1] less than or equal Reg [w2], then go to else-body

<if-body statements>

<else-body statements>
```

```
Example: Write a PIC24 assembly code according to the following flow chart below:

```
i = 8-bit variable
j, k = 16-bit variables

i = 3
  j = j + 1  case 1

i = 6
  j = j - 1  case 2

i = 100
  j = j + k  case 3

else
  j = j - k  case 4
```
mov.b #3, w1 ; #3 → Reg [w1] (only LSB)
mov.b #6, w2 ; #6 → Reg [w1] (only LSB)
mov.b #100, w3 ; #100 → Reg [w1] (only LSB)
mov.b i, wreg ; mem (i) → Reg [w0] (only LSB)
cp w0, w1 ; compare mem (i) with Reg [w1] = 3
bra NZ, case2 ; if Z-flag = 0, mem (i) ≠ 3, after compare then go to case2
; this is case1
inc j ; mem (j) = mem (j) + 1
bra end ; go to end

case 2:
cp.b w0, w2 ; compare mem (i) with Reg [w2] = 6
bra NZ, case3 ; if Z-flag = 0, mem (i) ≠ 6, after compare then go to case3
dec j ; mem (j) = mem (j) - 1
bra end ; go to end

case3:
cp.b w0, w3 ; compare mem (i) with Reg [w3] = 100
bra NZ, case4 ; if Z-flag = 0, mem (i) ≠ 100, after compare then go to case4
mov j, w0 ; mem (j) → Reg [w0]
add k, wreg ; mem (k) + Reg [w0] → Reg [w0]
mov w0, j ; mem (j) + mem (k) → mem (j)
bra end ; go to end

case4:
mov k, w0 ; mem (k) → Reg [w0]
sub j ; mem (j) – Reg [w0] → mem (j)
end:
Complex Conditional Structures

(a) The condition of ANDs
Assume the following flow chart composed of AND statements

```
cond1 && cond2 ... && condN
```

This flow chart can be simplified as:

```
cond1
|   true
|   false
|   \-- cond2
|       |   true
|       |   false
|       |   \-- cond3
|       |       |   true
|       |       |   false
|       |       |   \-- cond4
|       |       |       |   true
|       |       |       |   false
|       |       |       |   \-- if-body statements
|       |       |       |   \-- else-body statements
```
**Example:** Simplify the following flow chart where i, j, k are 16-bit variables and write a PIC24 assembly program.

The simplified flow chart is as follows:
The PIC24 assembly problem is as follows:

```assembly
mov i, w1 ; mem(i) → Reg [w1]
mov k, w2 ; mem(k) → Reg [w2]
cp w1, w2 ; compare mem (i) with mem (j)
bra GEU, else-body ; if mem (i) ≥ mem (j) then go to else-body
mov j, w0 ; mem (j) → Reg [w0]
cp w0, #20 ; compare mem (j) with 20
bra Z, else-body ; if Z-flag = 1, mem (j) = 20, then go to else-body
.
<if-body statements>
.
bra rest-of-the-program
.
else-body:
.
<else-body statements>
.
rest-of-the-program:
.
< rest-of-the-program>
```
(b) The condition of ORs

Assume the following flow chart composed of OR statements

This flow chart can be simplified as:
**Example:** Simplify the following flow chart where i, j, k, p, q are 16-bit variables and write a PIC24 assembly program.

\[(i < k) || (j == p) || (q! = 0)\]

This can be simplified as:

- \(i < k\) false
- \(j == p\) false
- \(q != 0\) false

if-body statements

else-body statements
The PIC24 assembly problem is as follows:

```
mov  i, w1 ; mem (i) → Reg [w1]
mov  k, w2 ; mem (k) → Reg [w2]
cp   w1, w2 ; compare mem (i) with mem (k)
bra  LTU, if-body ; if mem (i) > mem (k) then go to if-body
mov  j, w3 ; mem (j) → Reg [w3]
mov  p, w4 ; mem (p) → Reg [w4]
cp   w3, w4 ; compare mem (j) with mem (p)
bra  Z, if-body ; if mem (j) = mem (p) then go to if-body
mov  q, w0 ; mem (q) → Reg [w0]
cp0  w0 ; compare mem (q) with 0
bra  NZ, if-body ; if mem (q) ≠ 0 then go to if-body
.
<else-body statements>
.
if-body:
.
<if-body statements>
```
Example: Simplify the following flow chart where i, j, k, p, q are 16-bit variables and write a PIC24 assembly program.

\[(i < k) || (j == p) && (q! = 0)\]

This can be simplified as:
The PIC24 assembly problem is as follows:

```
mov i, w1          ; mem (i) → Reg [w1]
mov k, w2          ; mem (k) → Reg [w2]
cp  w1, w2         ; compare mem (i) with mem (k)
bra  LTU, if-body   ; if mem (i) > mem (k) then go to if-body
mov j, w3          ; mem (j) → Reg [w3]
mov p, w4          ; mem (p) → Reg [w4]
cp  w3, w4         ; compare mem (j) with mem (p)
bra  NZ, else-body  ; if mem (j) ≠ mem (p) then go to else-body
mov q, w0          ; mem (q) → Reg [w0]
cp0  w0            ; compare mem (q) with 0
bra  Z, else-body   ; if mem (q) = 0 then go to else-body
if-body:
  .
  <if-body statements>
  .
bra  rest-of-the-program
else-body:
  .
  <else-body statements>
  .
<rest-of-the-program statements>
```
Loop instructions

(a) While-loop

In a C-program, while-loop is written as follows:

while (condition)
  .
  <while-body>
  .
  <the rest of the code>

This program is implemented in a flow chart as follows:
Example: Convert the following C-program into PIC24 Assembly code.

while (k > j)
 .
<while-body>
 .
<the rest of the code>

The resultant Assembly program is as follows:

top:
 mov  k, w1 ; mem (k) → Reg [w1]
 mov  j, w2 ; mem (j) → Reg [w2]
 cp   w1, w2 ; compare mem (k) with mem (j)
 bra  LEU, rest-of-the-program ; if mem (k) ≤ mem (j) then go to rest-of-the-program
 .
< while-body for k > j >
 .
 bra top
rest-of-the-program:
(b) Do-loop

In a C-program, do-loop is written as follows:

do
  .
  <do-body>
while (condition)
  .
  <the rest of the code>

Important note:

In do-loop, do-body statements are executed first and then tested with a while-statement afterwards.
This program is implemented in a flow chart as follows:

Example: Convert the following C-program into PIC24 Assembly code.

do
  .
  <do-body>
  .
  while (k > j)
  .
  <the rest of the code>
The resultant Assembly program is as follows:

top:
.
<do-body>
.
mov k, w1 ; mem (k) → Reg [w1]
mov j, w2 ; mem (j) → Reg [w2]
cp w1, w2 ; compare mem (k) with mem (j)
bra GTU, top ; if mem (k) > mem (j) then go to top
.
rest-of-the-program:
.

(c) For-loop

In a C-program, for-loop is written as follows:

for (i = start; i <= finish; i++)
  .
  <for-body>
  .
This can be turned into a while statement:
i = start;
while (i <= finish)
  .
  <for-body>
  .
i = i + 1;
Example: Convert the following C-program into PIC24 Assembly code.

```c
for (i = 0; i != 10; i++)
    k = k + j;
```

First, convert this for-statement into a while-loop:

```c
i = 0;
while (i !=10)
{
    k = k + j;
    i = i +1;
}
```
Now, convert the while-loop into Assembly code.

```
mov  #0, w1       ; 0 → Reg [w1]
mov  w1, i        ; 0 → mem (i)
mov  #10, w0      ; 10 → Reg [w0]
mov  k, w2        ; mem (k) → Reg [w2]
mov  j, w3        ; mem (j) → Reg [w3]
top:
cp   i            ; compare mem (i) = 0 with Reg [w0] = 10
bra  Z, rest-program ; if mem (i) = 10 then go to rest-program
add  w2, w3, w2   ; mem (k) + mem (j) → mem (k)
in  i             ; mem (i) + 1 → mem (i)
bra  top
rest-program:
.
<rest-of-the-program>
```