## C Arithmetic operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, -</td>
<td>(+) addition, (−) subtraction</td>
</tr>
<tr>
<td>++, --</td>
<td>(++) increment, (−−) decrement</td>
</tr>
<tr>
<td>*, /</td>
<td>(*) multiplication, (/) division</td>
</tr>
<tr>
<td>&gt;&gt;, &lt;&lt;</td>
<td>right shift (&gt;&gt;), left shift (&lt;&lt;)</td>
</tr>
<tr>
<td>&amp;,</td>
<td>, ^</td>
</tr>
<tr>
<td>~</td>
<td>bitwise complement</td>
</tr>
</tbody>
</table>

The above are C operators that we would like to implement in PIC24 assembly language. Multiplication and division will be covered in a later lecture.
Bit-wise Logical operations

Bitwise AND operation

\[
\begin{align*}
\text{AND.}\{B\} & \quad Wb, Ws, Wd  \\
\text{AND.}\{B\} & \quad f  \\
\text{AND.}\{B\} & \quad f, \ WREG  \\
\text{AND.}\{B\} & \quad #\text{lit10}, Wn
\end{align*}
\]
\[
\begin{align*}
(Wb) & \& (Ws) \rightarrow Wd  \\
(f) & \& \text{(WREG)} \rightarrow f  \\
(f) & \& \text{(WREG)} \rightarrow \text{WREG}  \\
\text{lit10} & \& (Wn) \rightarrow Wn
\end{align*}
\]
\[
\begin{align*}
\rightarrow j = k \& i;  \\
\rightarrow j = j \& k;  \\
\rightarrow j = j \& k;  \\
\rightarrow j = j \& \text{literal};
\end{align*}
\]

Bitwise Inclusive OR operation

\[
\begin{align*}
\text{IOR.}\{B\} & \quad Wb, Ws, Wd  \\
\text{IOR.}\{B\} & \quad f  \\
\text{IOR.}\{B\} & \quad f, \ WREG  \\
\text{IOR.}\{B\} & \quad #\text{lit10}, Wn
\end{align*}
\]
\[
\begin{align*}
(Wb) & \mid (Ws) \rightarrow Wd  \\
(f) & \mid \text{(WREG)} \rightarrow f  \\
(f) & \mid \text{(WREG)} \rightarrow \text{WREG}  \\
\text{lit10} & \mid (Wn) \rightarrow Wn
\end{align*}
\]
\[
\begin{align*}
\rightarrow j = k \mid i;  \\
\rightarrow j = j \mid k;  \\
\rightarrow j = j \mid k;  \\
\rightarrow j = j \mid \text{literal};
\end{align*}
\]
Bit-wise Logical operations (cont.)

Bitwise XOR operation

XOR.\{B\} Wb, Ws, Wd \rightarrow (Wb)^\wedge (Ws) \rightarrow Wd \quad j = k^\wedge i;

XOR.\{B\} f \rightarrow (f)^\wedge (WREG) \rightarrow f \quad j = j^\wedge k;

XOR.\{B\} f, WREG \rightarrow (f)^\wedge (WREG) \rightarrow WREG \quad j = j^\wedge k;

XOR.\{B\} \#\text{lit10}, Wn \rightarrow \text{lit10}^\wedge (Wn) \rightarrow Wn \quad j = j^\wedge \text{literal};

Bitwise complement operation

COM.\{B\} Ws, Wd \rightarrow \neg (Ws) \rightarrow Wd \quad j = \neg k;

COM.\{B\} f \rightarrow \neg(f) \rightarrow f \quad j = \neg j;

COM.\{B\} f, WREG \rightarrow \neg(f) \rightarrow WREG \quad j = \neg k;
Bit-wise Logical operations (cont.)

Clear ALL bits:

CLR.\{B\} f 0 \rightarrow f \quad j=0;
CLR.\{B\} WREG 0 \rightarrow WREG \quad j=0;
CLR.\{B\} Wd 0 \rightarrow Wd \quad j=0;

Set ALL Bits:

SETM.\{B\} f 111\ldots1111 \rightarrow f
SETM.\{B\} WREG 111\ldots1111 \rightarrow WREG
SETM.B\} Wd 111\ldots1111 \rightarrow Wd
Clearing a group of bits

Clear upper four bits of i.

In C:

```c
uint8 i;
    i = i & 0x0F;  // The ‘mask’
```

In PIC24 μC assembly

```assembly
mov.b  #0x0F, W0 ; W0 = mask
and.b i ; i = i & 0x0f
```

AND: mask bit = ‘1’, result bit is same as operand.

mask bit = ‘0’, result bit is cleared

Data Memory

<table>
<thead>
<tr>
<th>Location</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i) 0x0800</td>
<td>0x2C</td>
</tr>
<tr>
<td>(j) 0x0801</td>
<td>0xB2</td>
</tr>
<tr>
<td>(k) 0x0802</td>
<td>0x8A</td>
</tr>
</tbody>
</table>

\[ i = \quad 0x2C = \quad 0010 \quad 1100 \]
\[ \text{mask} = \quad 0x0F = \quad 0000 \quad 1111 \]
\[ \text{result} = \quad 0000 \quad 1100 \]
\[ = \quad 0x0C \]
Setting a group of bits

Set bits b3:b1 of j

In C:

```c
uint8 j;
```

```c
j = j | (0x0E);  // The 'mask'
```

In PIC24 μC assembly

```asm
mov.b #0x0E, W0   ; W0 = mask
ior.b j           ; j = j | 0x0E
```

**OR:** mask bit = ‘0’, result bit is same as operand.

mask bit = ‘1’, result bit is set

<table>
<thead>
<tr>
<th>Location</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i) 0x0800</td>
<td>0x2C</td>
</tr>
<tr>
<td>(j) 0x0801</td>
<td>0xB2</td>
</tr>
<tr>
<td>(k) 0x0802</td>
<td>0x8A</td>
</tr>
</tbody>
</table>
Complementing a group of bits

Complement bits b7:b6 of k

In C:
```c
uint8 k;

k = k ^ 0xC0;
```

In PIC24 μC assembly
```assembly
mov.b #0xC0, W0 ; W0 = mask
xor.b k ; k = k ^ 0xC0
```

XOR: mask bit = ‘0’, result bit is same as operand.
mask bit = ‘1’, result bit is complemented
Complementing all bits

Complement all bits of k

In C:

```c
uint8 k;
k = ~k;
```

In PIC24 μC assembly

```assembly
com.b k ; k = ~k
```

<table>
<thead>
<tr>
<th>Location</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i) 0x0800</td>
<td>0x2C</td>
</tr>
<tr>
<td>(j) 0x0801</td>
<td>0xB2</td>
</tr>
<tr>
<td>(k) 0x0802</td>
<td>0x8A</td>
</tr>
</tbody>
</table>

k = 0x8A = 1000 1010

After complement

```assembly
result = 0111 0101
        = 0x75
```
Bit set, Bit Clear, Bit Toggle instructions

Can set/clear/complement one bit of a data memory location by using the AND/OR/XOR operations, but takes multiple instructions as previously seen.

The bit clear (bcf), bit set (bsf), bit toggle (btg) instructions clear/set/complement one bit of data memory or working registers using one instruction.

<table>
<thead>
<tr>
<th>Name</th>
<th>Mnemonic</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit Set</td>
<td>bset{.b} $Ws$, #bit4 $Ws$ indirect modes</td>
<td>$1 \rightarrow Ws&lt;\text{bit}4&gt;$</td>
</tr>
<tr>
<td></td>
<td>bset{.b} $f$, #bit4</td>
<td>$1 \rightarrow f&lt;\text{bit}4&gt;$</td>
</tr>
<tr>
<td>Bit Clear</td>
<td>bclr{.b} $Ws$, #bit4 $Ws$ indirect modes</td>
<td>$0 \rightarrow Ws&lt;\text{bit}4&gt;$</td>
</tr>
<tr>
<td></td>
<td>bclr{.b} $f$, #bit4</td>
<td>$0 \rightarrow f&lt;\text{bit}4&gt;$</td>
</tr>
<tr>
<td>Bit Toggle</td>
<td>btg{.b} $Ws$, #bit4 $Ws$ indirect modes</td>
<td>$\sim Ws&lt;\text{bit}4&gt; \rightarrow Ws&lt;\text{bit}4&gt;$</td>
</tr>
<tr>
<td></td>
<td>btg{.b} $f$, #bit4</td>
<td>$\sim f&lt;\text{bit}4&gt; \rightarrow f&lt;\text{bit}4&gt;$</td>
</tr>
</tbody>
</table>
Bit clear/set/toggle examples

Clear bit 7 of k, Set bit 2 of j, complement bit 5 of i.

In C:
```c
uint8  i, j, k;
k = k & 0x7F;
j = j | 0x04;
i = i ^ 0x20;
```

In PIC24 μC assembly
```asm
bclr.b k,#7
bset.b j,#2
btg.b i,#5
```
**status Register**

The *STATUS* register is a special purpose register (like the Wn registers).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| – | – | – | – | – | – | – | DC | IPL2 | IPL1 | IPL0 | RA | N | OV | Z | C |

---

**Status Register**

<table>
<thead>
<tr>
<th>C</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>Zero</td>
</tr>
<tr>
<td>OV</td>
<td>Overflow</td>
</tr>
<tr>
<td>N</td>
<td>Negative</td>
</tr>
<tr>
<td>RA</td>
<td>Repeat Loop Active</td>
</tr>
<tr>
<td>IPL[2:0]</td>
<td>Interrupt Priority Level</td>
</tr>
<tr>
<td>DC</td>
<td>Decimal Carry</td>
</tr>
<tr>
<td>–</td>
<td>Unimplemented</td>
</tr>
</tbody>
</table>

The C, Z, OV, N, DC flags can be user set/cleared; also are set/cleared as a side effect of instruction execution.

The RA bit is read-only; set when a repeat instruction is active, cleared when repeat is finished.

The IPL[2:0] bits are user set/cleared.

We will **not** discuss the DC flag; it is used in Binary Coded Decimal arithmetic.
Carry, Zero Flags

Bit 0 of the status register is known as the **carry** (C) flag.

Bit 1 of the status register is known as the **zero** (Z) flag.

These flags are set as **side-effects** of particular instructions or can be set/cleared explicitly using the *bset/bclr* instructions.

How do you know if an instruction affects C, Z flags?

Look at Table 19-2 in PIC24HJ32GP202 μC datasheet. – *add* affects all ALU flags, *mov f* only Z, N flags, and *mov f, Wn* no flags.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Syntax.</th>
<th>Desc</th>
<th># of words</th>
<th>Instr Cycles</th>
<th>Status affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Addition: Carry, Zero Flags

Zero flag is set if result is zero and cleared otherwise.

In addition, carry flag is set if there is a carry out of the MSbit and cleared otherwise.

In byte (8-bit) mode, $C=1$ if sum $> 255$ ($0xFF$)
In word (16-bit) mode, $C=1$ if sum $> 65535$ ($0xFFFF$)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0xF0 +0x20</td>
<td>0x00 +0x00</td>
<td>0x01 +0xFF</td>
<td>0x80 +0x7F</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td>0x00</td>
<td>0x00</td>
<td>0xFF</td>
<td></td>
</tr>
<tr>
<td>$Z=0,$</td>
<td>$Z=1,$</td>
<td>$Z=1,$</td>
<td>$Z=0,$</td>
<td></td>
</tr>
<tr>
<td>$C=1$</td>
<td>$C=0$</td>
<td>$C=1$</td>
<td>$C=0$</td>
<td></td>
</tr>
</tbody>
</table>

Byte mode operations are shown.
Subtraction: Carry, Zero Flags

Zero flag is set if result is zero and cleared otherwise.

In subtraction, carry flag is **cleared** if there is a borrow into the MSb (unsigned underflow, result is < 0, larger number subtracted from smaller number). Carry flag is **set** if no borrow occurs.

<table>
<thead>
<tr>
<th>0xF0</th>
<th>0x00</th>
<th>0x01</th>
</tr>
</thead>
<tbody>
<tr>
<td>- 0x20</td>
<td>-0x00</td>
<td>-0xFF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0xD0</td>
<td>0x00</td>
<td>0x02</td>
</tr>
<tr>
<td>Z=0,</td>
<td>Z=1,</td>
<td>Z=0,</td>
</tr>
<tr>
<td>C=1</td>
<td>C=1</td>
<td>C=0</td>
</tr>
</tbody>
</table>

For a subtraction, the combination of Z=1, C=0 will not occur. Byte mode operations are shown.
How do you remember setting of C flag for Subtraction?

Subtraction of \( A - B \) is actually performed in hardware as \( A + (\neg B) + 1 \)

The value \( (\neg B) + 1 \) is called the **two’s complement** of \( B \) (more on this later). The \( C \) flag is affected by the addition of \( A + (\neg B) + 1 \)

\[
\begin{align*}
0xF0 - 0x20 & : \\
\text{---------} & : \\
0xD0 & : Z=0, \ C=1 \\
\text{No borrow, } C=1
\end{align*}
\]

\[
\begin{align*}
0x20 & = 0010 0000 \\
\neg0x20 & = 1101 1111 \\
+0xDF & = 0xDF \\
+0x01 & \\
\text{---------} & : \\
0xD0 & : Z=0, \ C=1 \\
\text{Carry out of MSB, so } C=1
\end{align*}
\]
C Shift Left, Shift Right

*logical* Shift right  \( i \gg 1 \)

all bits shift to right by one, ‘0’ into MSB (8-bit right shift shown)

\[
\begin{array}{cccccccc}
0 & b7 & b6 & b5 & b4 & b3 & b2 & b1 \\
\end{array}
\]

original value

\[
\begin{array}{cccccccc}
0 & b7 & b6 & b5 & b4 & b3 & b2 & b1 \\
\end{array}
\]

\( i \gg 1 \) (right shift by one)

---

Shift left  \( i \ll 1 \)

all bits shift to left by one, ‘0’ into LSB (8-bit left shift shown)

\[
\begin{array}{cccccccc}
b6 & b5 & b4 & b3 & b2 & b1 & b0 & 0 \\
\end{array}
\]

original value

\[
\begin{array}{cccccccc}
b6 & b5 & b4 & b3 & b2 & b1 & b0 & 0 \\
\end{array}
\]

\( i \ll 1 \) (left shift by one)
## PIC24 Family Unsigned Right Shifts

### Logical Shift Right

<table>
<thead>
<tr>
<th>Description</th>
<th>Syntax</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log. Shift Right f</td>
<td>LSR{.B} f</td>
<td>f &gt;&gt; 1 → f</td>
</tr>
<tr>
<td>Log. Shift Right f, WREG</td>
<td>LSR{.B} f, WREG</td>
<td>f &gt;&gt; 1 → WREG</td>
</tr>
<tr>
<td>Log. Shift Right Ws</td>
<td>LSR{.B} Ws, Wd</td>
<td>Ws &gt;&gt; 1 → Wd</td>
</tr>
<tr>
<td>Log. Shift Right by short Literal</td>
<td>LSR Wb, #lit4, Wd</td>
<td>Wb &gt;&gt; lit4 → Wd</td>
</tr>
<tr>
<td>Log. Shift Right by Ws</td>
<td>LSR Wb, Ws, Wd</td>
<td>Wb &gt;&gt; Ws → Wd</td>
</tr>
</tbody>
</table>

The last two logical shifts can shift multiple positions in one instruction cycle (up to 15 positions), but only as word operations. There is an arithmetic right shift that will be covered in a later lecture.
PIC24 Family Left Shifts

Shift left

Cflag

b7 b6 b5 b4 b3 b2 b1 b0

Cflag

b15 b14 . . . . . b1 b0

8-bit
0

16-bit
0

Descr: Syntax Operation

Shift left f
SL\{.B\} f f << 1 → f
SL\{.B\} f,WREG f << 1 → WREG

Shift left Ws
SL\{.B\} Ws,Wd Ws << 1 → Wd
Shift left by short Literal
SL Wb, #lit4, Wd Wb << lit4 → Wd

Shift left by Ws
SL Wb, Ws, Wd Wb << Ws → Wd

The last two shifts can shift multiple positions in one instruction cycle (up to 15 positions), but only as word operations.
PIC24 Rotate Instructions

PIC24 has some rotate left and rotate right instructions as well:

- **rotate right thru carry (rrc)**
- **rotate left thru carry (rlc)**
- **rotate right no carry (rrnc)**
- **rotate left no carry (rlnc)**

The **rrc/rlc** instructions are used in the next chapter for 32-bit shift operations. The **rrnc/rlnc** are not discussed further. The valid addressing modes are the same as for the shift operations that only shift by one position.
C Shift operations

In C

```c
uint16 u16_p, u16_k;
uint8 u8_i;
u8_i = u8_i >> 2;
u16_k = u16_p << 10;
```

In Assembly

```assembly
lsr.b u8_i ;i = i >> 1;
lsr.b u8_i ;i = i >> 1;
mov u16_p, w1 ;w1 = p
sl w1, #10, w1 ;w1 = w1 << 10
mov w1, u16_k ;k = w1
```

It is sometimes more efficient to repeat a single position shift instruction performing a multi-bit shift.
Arithmetic Example

(a) In C

```
uint16 i, n, p;

k = n + (i<<3) - p;
```

(b) Steps:
Copy n, i to working registers
Perform i << 3
Add to n
Subtract p
Write to k

(c) In Assembly

```
mov n, W0 ; W0 = n
mov i, W1 ; W1 = i
sl W1, #3, W1 ; W1 = i << 3;
add W0, W1, W0 ; W0 = n + (i<<3)
mov p, W1 ; W1 = p
sub W0, W1, W0 ; W0 = (n + (i<<3)) - p
mov W0, k ; k = (n + (i<<3)) - p
```

Use working registers for storage of intermediate results.

Copyright Delmar Cengage Learning 2008. All Rights Reserved.
From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”
Mixed 8-bit, 16-bit operations

In C

```c
uint16 u16_p;
uint8 u8_i;

u16_p = u16_p + u8_i;
```

(a) In Assembly (incorrect)

```assembly
mov.b u8_i, WREG ; W0.LSB = u8_i
add u16_p ; u16_p = u16_p + W0
```

The upper 8 bits of W0 are unknown; 16-bit sum is likely incorrect.

(b) In Assembly (correct)

```assembly
mov.b u8_i, WREG ; W0.lsb = u8_i
ze W0, W0 ; Zero extend W0
add u16_p ; u16_p = u16_p + W0
```

The upper 8 bits of W0 are now zero; unsigned 8-bit variables should be zero-extended before use in 16-bit operations.
Conditional Execution using Bit Test

The ‘bit test f, skip if clear’ (btsc) and ‘bit test f, skip if set’ (btss) instructions are used for conditional execution.

\[
\text{btsc \{b\} f, \#bit4 ; skips next instruction is } f<\#bit4> \text{ is clear (‘0’)}
\]

\[
\text{btss \{b\} f, \#bit4 ; skips next instruction is } f<\#bit4> \text{ is set (‘1’)}
\]

Bit test instructions are just the first of many different methods of performing conditional execution in the PIC24 µC.
Number Sequencing Task using btsc

(1) .bss ;uninitialized data section
(2) loc: .space 1 ;byte variable
(3) out: .space 1 ;byte variable
(4) .text ;Start of Code section
(5) __reset: ; first instruction
(6) mov #__SP_init, W15 ;Initialize the Stack Pointer
(7) ;bclr loc, #0 ;uncomment for loc<0>=0
(8) bset loc, #0 ;uncomment for loc<0>=1
(9) loop_top:
(10) btsc.b loc,#0 ;skip next if loc<0> is 0
(11) goto loc_lsb_is_1 ;loc<0> is 0 if reach here
(12) mov.b #3,w0
(13) mov.b wreg,out ;out = 3
(14) mov.b #2,w0
(15) mov.b wreg,out ;out = 2
(16) mov.b #4,w0
(17) mov.b wreg,out ;out = 4
(18) loc_lsb_is_1:
(19) mov.b #8,w0
(20) mov.b wreg,out ;out = 8
(21) mov.b #5,w0
(22) mov.b wreg,out ;out = 5
(23) mov.b #6,w0
(24) mov.b wreg,out ;out = 6
(25) mov.b #1,w0
(26) mov.b wreg,out ;out = 1
(27) goto loop_top ;loop forever

Copyright Delmar Cengage Learning 2008. All Rights Reserved.
From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

V 0.2
## C Conditional Tests

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==, !=&quot;=</td>
<td>equal, not-equal</td>
</tr>
<tr>
<td>&gt;, &gt;=&quot;=&quot;</td>
<td>greater than, greater than or equal</td>
</tr>
<tr>
<td>&lt;, &lt;=&quot;=&quot;</td>
<td>less than, less than or equal</td>
</tr>
<tr>
<td>&quot;&amp;&quot;</td>
<td>logical AND</td>
</tr>
<tr>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;!&quot;</td>
<td>logical negation</td>
</tr>
</tbody>
</table>

If an operator used in a C conditional test, such as an IF statement or WHILE statement, returns nonzero, then the condition test is TRUE.
Logical Negation vs. Bitwise Complement

\[ !i \quad \text{is not the same as} \quad \sim i \]

\[ i = 0xA0 \quad i = 0xA0 \]

\[ !(i) \quad \rightarrow \quad 0 \quad \sim (i) \quad \rightarrow \quad 0x5F \]

Logical operations: !, &&, || always treat their operands as either being zero or non-zero, and the returned result is always either 0 or 1.
Examples of C Equality, Inequality, Logical, Bitwise Logical Tests

```c
uint8 a,b,a_lt_b, a_eq_b, a_gt_b, a_ne_b;

a = 5; b = 10;
a_lt_b = (a < b);  // a_lt_b result is 1
a_eq_b = (a == b); // a_eq_b result is 0
a_gt_b = (a > b);  // a_gt_b result is 0
a_ne_b = (a != b); // a_ne_b result is 1
```

```c
uint8 a_lor_b, a_bor_b, a_lneg_b, a_bcom_b;

(2)     a = 0xF0; b = 0x0F;
(3)     a_land_b = (a && b); //logical and, result is 1
(4)     a_band_b = (a & b);  //bitwise and, result is 0
(5)     a_lor_b = (a || b);  //logical or, result is 1
(6)     a_bor_b = (a | b);   //bitwise or, result is 0xFF
(7)     a_lneg_b = (!b);    //logical negation, result is 0
(8)     a_bcom_b = (~b);    //bitwise negation, result is 0xF0
```
if{} Statement Format in C

if (condition_test) {
    if-body  \(\rightarrow\) Executed when condition_test is non-zero (true)
} else {
    else-body  \(\rightarrow\) Executed when condition_test is zero (false)
}

if-body and else-body can contain multiple statements.

else-body is optional.
C zero/non-zero tests

A C conditional test is true if the result is non-zero; false if the result is zero.

The ! operator is a logical test that returns 1 if the operator is equal to ‘0’, returns ‘0’ if the operator is non-zero.

```c
if (!i) {
    // do this if i zero
    j = i + j;
}
```

```c
if (i) {
    // do this if i non-zero
    j = i + j;
}
```

Could also write:

```c
if (i == 0) {
    // do this if i zero
    j = i + j;
}
```

```c
if (i != 0) {
    // do this if i non-zero
    j = i + j;
}
```
C equality tests

‘==’ is the equality test in C; ‘=’ is the assignment operator.

A common C code mistake is shown below (= vs ==)

```c
if (i = 5) {
    j = i + j;
}//wrong
```

```c
if (i == 5) {
    j = i + j;
}//right
```

Always executes because i=5 returns 5, so conditional test is always non-zero, a true value. The = is the assignment operator.

The test i == 5 returns a 1 only when i is 5. The == is the equality operator.
C Bitwise logical vs. Logical AND

The ‘&’ operator is a bitwise logical AND. The ‘&&’ operator is a logical AND and treats its operands as either zero or non-zero.

\[
\text{if (i && j) \{ } \quad \text{is read as:} \quad \text{If } \ (\text{i is nonzero) AND (j is nonzero) } \) \text{ then do this.}
\]

\[
\text{if (i & j) \{ } \quad \text{is read as:} \quad \text{If } \ (\text{i bitwise AND j) is nonzero) } \) \text{ then do this.}
\]

\[
i = 0xA0, \ j = 0x0B; \quad \text{1}
\]

\[
i = 0xA0, \ j = 0x0B; \quad (i \ & j) \quad 0x0
\]
C Bitwise logical vs. Logical OR

The ‘|’ operator is a bitwise logical OR. The ‘||’ operator is a logical OR and treats its operands as either zero or nonzero.

```c
if (i || j) { /* do this */
    }
```

is read as:

If ( (i is nonzero) OR (j is nonzero) ) { do...

```c
if (i | j) { /* do this */
    }
```

is read as:

If ( (i bitwise OR j) is nonzero) ) { do....

i = 0xA0, j = 0x0B;

(i || j)  →  1

i = 0xA0, j = 0x0B;

(i | j)  →  0xAB
Non-Zero Test labels for SFRs defined in \textit{p24Hxxxx.inc}; use for clarity!!!!

In C

\begin{verbatim}
uint16 k;

if (k) {
  \textit{if-body}
}
\textit{... rest of code}
\end{verbatim}

In Assembly

\begin{verbatim}
  mov k ; k = k, affects N,Z flags
  btsc SR,#1 ; skip if Z = 0 (Z is SR<1>)
  goto end_if ; Z = 1, k is 0
\begin{cases}
  \textit{if-body stmt1} \\
  \ldots \textit{stmtN}
\end{cases}
end_if: \textit{... rest of code}
\end{verbatim}

The \textit{mov i} instruction just moves \textit{i} back onto itself! Does no useful work except to affect the Z, N flags.

Copyright Delmar Cengage Learning 2008. All Rights Reserved.
From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

V 0.2
Conditional Execution using branches

A branch functions as a conditional goto based upon the setting of one more flags

Simple branches test only one flag:

- BRA Z, <label> branch to label if Z=1
- BRA NZ, <label> branch to label if Z=0 (not zero)
- BRA C, <label> branch to label if C=1
- BRA NC, <label> branch to label if C=0 (no carry)
- BRA N, <label> branch to label if N=1
- BRA NN, <label> branch to label if N=0 (not negative)

BRA <label> unconditional branch to <label>

Using branch instructions instead of btsc/btss generally results in fewer instructions, and improves code clarity.
Non-Zero Test

The `bra Z` (branch if Zero, Z=1) replaces the `btfsc/goto` combination.

For a non-zero test `if(!i){}` replace `bra Z` with `bra NZ`
General if-else form with branches

In C

```c
if (condition_test) {
    if-body
} else {
    else-body
}
... rest of code
```

In Assembly

```
test condition ; set status flags
cond == false
branch
skip
if-body
cond == true, execute if-body
if-body stmt1
    ...
stmtN
branch always
else_body:
else-body stmt1
    ...
stmtN
end_if:
... rest of code
```

Choose the branch instruction such that the branch is **TAKEN** when the condition is **FALSE**.
Equality Test (==)

In C

```c
uint16 k, j;
if (k == j) {
    if-body
}
... rest of code
```

In Assembly

```
mov j,W0 ; W0 = j
sub k,WREG ; W0 = k - j
bra NZ,end_if ; skip if-body when Z=0 (k != j)
```

```
if-body stmt1
... stmtN
```

```
end_if:
... rest of code
```

Subtraction operation of k-j performed to check equality;
if k == j then subtraction yields ‘0’, setting the Z flag. Does not matter if k-j or j-k is performed.
$>$, $\geq$, $<$, $\leq$ tests using $Z$, $C$ flags and subtraction

(a) $k - j$

$\begin{align*}
&k \geq j \quad k \leq j \\
&C = 1 \quad C = 0 \text{ or } Z = 1
\end{align*}$

$\begin{align*}
&k > j \\
&C = 1 \\
&Z = 0
\end{align*}$

$\begin{align*}
&k < j \\
&C = 0 \\
&Z = 0
\end{align*}$

Note: $k \leq j$ is $\sim(k > j)$ is $\sim(C \& \sim Z)$ is $\sim(C \mid Z)$ by DeMorgan’s law. Similarly, $k < j$ is $\sim(k \geq j)$ is $\sim(C)$ is $\sim C$.  

(b) $j - k$

$\begin{align*}
&k \geq j \quad k \leq j \\
&C = 0 \text{ or } Z = 1 \quad C = 1
\end{align*}$

$\begin{align*}
&k > j \\
&C = 0 \\
&Z = 0
\end{align*}$

$\begin{align*}
&k < j \\
&C = 1 \\
&Z = 0
\end{align*}$

Note: $k < j$ is $\sim(k \geq j)$ is $\sim(!C \mid Z)$ is $(C \& \sim Z)$ by DeMorgan’s law. Similarly, $k \leq j$ is $\sim(k > j)$ is $\sim(\sim C)$ is $C$.  

V 0.2
The false condition of \( k > j \) is \( k \leq j \), so use \( k \leq j \) to skip around the if-body. For the k-j test, this is accomplished by \( C = 0 \) or \( Z = 1 \), requiring two branches.
k>j test using j-k

**In C**

```c
uint16 k, j;
if (k > j) {
    if-body
}...
```

**In Assembly**

```assembly
mov k, W0 ; W0 = k
sub j, WREG ; W0 = j - k
bra C, end_if ; skip if-body when C = 1 (k <= j)
    if-body stmt1
    ... stmtN
end_if:
    ... rest of code
```

The false condition of k>j is k<=j, so use k<=j to skip around the if-body. For the j-k test, this is accomplished by C=1, requiring one branch.
Comparison, Unsigned Branches

Using subtraction, and simple branches can be confusing, since it can be difficult to remember which preferred subtraction to perform and which branch to use.

Also, the subtraction operation overwrites a register value.

The comparison instruction (CP) performs a subtraction without placing the result in register:

<table>
<thead>
<tr>
<th>Descr:</th>
<th>Syntax</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare f with WREG</td>
<td>CP{.B} f</td>
<td>f – WREG</td>
</tr>
<tr>
<td>Compare Wb with Ws</td>
<td>CP {.B} Wb,Ws</td>
<td>Wb – Ws</td>
</tr>
<tr>
<td>Compare Wb with #lit5</td>
<td>CP{.B} Wb,#lit5</td>
<td>Wb – #lit5</td>
</tr>
<tr>
<td>Compare f with zero</td>
<td>CP0{.B} f</td>
<td>f – 0</td>
</tr>
<tr>
<td>Compare Ws with zero</td>
<td>CP0{.B} Ws</td>
<td>Ws – 0</td>
</tr>
</tbody>
</table>
Comparison, Unsigned Branches (cont)

Unsigned branches are used for unsigned comparisons and tests a combination of the Z, C flags, depending on the comparison.

<table>
<thead>
<tr>
<th>Descr:</th>
<th>Syntax</th>
<th>Branch taken when</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch &gt;, unsigned</td>
<td>BRA GTU, label</td>
<td>C=1 &amp;&amp; Z=0</td>
</tr>
<tr>
<td>Branch &gt;=, unsigned</td>
<td>BRA GEU, label</td>
<td>C=1</td>
</tr>
<tr>
<td>Branch &lt;, unsigned</td>
<td>BRA LTU, label</td>
<td>C=0</td>
</tr>
<tr>
<td>Branch &lt;=, unsigned</td>
<td>BRA LEU, label</td>
<td>C=0</td>
</tr>
</tbody>
</table>

Use a Compare instruction to affect the flags before using an unsigned branch.

Example:

```assembly
CP W0, W1       ; W0 – W1
BRA GTU, place  ; branch taken if W0 > W1
```
Unsigned Comparison (> test)

In C
uint16 k, j;

if (k > j) {
  if-body
}

... rest of code

In Assembly

    mov j, W0       ; W0 = j
    op k            ; k - WREG
    bra LEU, end_if ; skip if-body when k <= j
    if-body stmt1
    ... stmtN
    end_if:
    ... rest of code

For k > j test, use the LEU (less than or equal unsigned) branch to skip IF body if k <= j

Copyright Delmar Cengage Learning 2008. All Rights Reserved.
From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.
If-else Example

In C
uint16 k, j;
if (k <= j) {
  // if-body code
} else {
  //else-body code
}
// ...rest of code...

In Assembly
mov j,W0 ;W0 = j
op k ;k - WREC
bra GTU, else_body ;skip if-body when k > j
  ..if-body stmt1
  ..if-body stmtN
bra end_if ;use unconditional branch
else_body: ;to skip else-body after
  ..else-body stmt1
  ..else-body stmtN
end_if: ;executing if-body
  ..rest of code..

Must use unconditional branch at end of if-body to skip the else-body.

Copyright Delmar Cengage Learning 2008. All Rights Reserved.
From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

V 0.2
## Unsigned literal Comparison

<table>
<thead>
<tr>
<th>(a)</th>
<th>In C</th>
<th>In Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint16 k;</td>
<td>mov k, W0</td>
<td>5-bit literal, unsigned range 0 to 31</td>
</tr>
<tr>
<td>if (k &gt; 10) {</td>
<td>cp $W0, #10</td>
<td>; $W0 = k</td>
</tr>
<tr>
<td>// if-body code</td>
<td>bra LEU, end_if</td>
<td>; k - 10</td>
</tr>
<tr>
<td>}</td>
<td>..if-body stmt1</td>
<td>; skip if-body when k &lt;= 10</td>
</tr>
<tr>
<td>// ...rest of code...</td>
<td>..if-body stmtN</td>
<td>end_if:</td>
</tr>
<tr>
<td></td>
<td>..rest of code...</td>
<td>..rest of code...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(b)</th>
<th>In C</th>
<th>In Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint16 k;</td>
<td>mov #520, W0</td>
<td>16-bit literal, unsigned range 0 to 65535</td>
</tr>
<tr>
<td>if (k &gt; 520) {</td>
<td>cp k</td>
<td>; $W0 = 520</td>
</tr>
<tr>
<td>// if-body code</td>
<td>bra LEU, end_if</td>
<td>; k - WREG</td>
</tr>
<tr>
<td>}</td>
<td>..if-body stmt1</td>
<td>; skip if-body when k &lt;= 520</td>
</tr>
<tr>
<td>// ...rest of code...</td>
<td>..if-body stmtN</td>
<td>end_if:</td>
</tr>
<tr>
<td></td>
<td>..rest of code...</td>
<td>..rest of code...</td>
</tr>
</tbody>
</table>

Copyright Delmar Cengage Learning 2008. All Rights Reserved.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.
**switch Statement in C**

(a) Chained *if-else* structure

```c
uint8 u8_i;
uint16 u16_j, u16_k;

if (u8_i == 1) {
    u16_k++;
} else if (u8_i == 2) {
    u16_j--;
} else if (u8_i == 3) {
    u16_j = u16_j + u16_k;
} else {
    u16_k = u16_k - u16_j;
}
```

(b) *switch* structure

```c
uint8 u8_i;
uint16 u16_j, u16_k;

switch (u8_i) {
    case 1: u16_k++;
        break;
    case 2: u16_j--;
        break;
    case 3: u16_j = u16_j + u16_k;
        break;
    default: u16_k = u16_k - u16_j;
}
```

**A switch statement** is a shorthand version of an *if-else* chain where the same variable is compared for equality against different values.
**switch** Statement in assembly language

In C
```c
uint8 u8_i;
uint16 u16_j, u16_k;

switch (u8_i) {
    case 1: u16_k++; 
        break;
    case 2: u16_j--; 
        break;
    case 3:
        u16_j = u16_j + u16_k;
        break;
    default:
        u16_k = u16_k - u16_j;
}
```

In Assembly
```assembly
mov.b u8_i, WREG
cp.b W0, #1
bra NZ, case_2
inc u16_k
bra end_switch

case_2:
    cp.b W0, #2
    bra NZ, case_3

case_3:
    cp.b W0, #3
    bra NZ, default
    mov u16_k, W0
    add u16_j
    bra end_switch

default:
    mov u16_j, W0
    sub u16_k
    end_switch:
    ..rest of code..
```

Note: The literal size in the CP instruction is 5-bits (unsigned values of 0-31).
## Unsigned, Zero, Equality Comparison Summary

<table>
<thead>
<tr>
<th>Condition</th>
<th>Test</th>
<th>True Branch</th>
<th>False Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>i == 0</td>
<td>i − 0</td>
<td>bra Z</td>
<td>bra NZ</td>
</tr>
<tr>
<td>i != 0</td>
<td>i − 0</td>
<td>bra NZ</td>
<td>bra Z</td>
</tr>
<tr>
<td>i == k</td>
<td>i − k</td>
<td>bra Z</td>
<td>bra NZ</td>
</tr>
<tr>
<td>i != k</td>
<td>i − k</td>
<td>bra NZ</td>
<td>bra Z</td>
</tr>
<tr>
<td>i &gt; k</td>
<td>i − k</td>
<td>bra GTU</td>
<td>bra LEU</td>
</tr>
<tr>
<td>i &gt;= k</td>
<td>i − k</td>
<td>bra GEU</td>
<td>bra LTU</td>
</tr>
<tr>
<td>i &lt; k</td>
<td>i − k</td>
<td>bra LTU</td>
<td>bra GEU</td>
</tr>
<tr>
<td>i &lt;= k</td>
<td>i − k</td>
<td>bra LEU</td>
<td>bra GTU</td>
</tr>
</tbody>
</table>
Other PIC24 Comparison Instructions

The PIC24 has various other comparison instructions

- **CPSEQ Wb, Wn** ; if Wb == Wn, skip next instruction
- **CPSNE Wb, Wn** ; if Wb != Wn, skip next instruction
- **CPSGT Wb, Wn** ; if Wb == Wn, skip next instruction
- **CPSLT Wb, Wn** ; if Wb < Wn, skip next instruction

These are provided as upward compatibility with previous PICmicro families, and may save an instruction or two in certain situations. However, we will not use them since their functionality can be duplicated by previously covered compare/branch instructions.
Complex Conditions (&&)

In C
if (condition_test1 &&
    condition_test2 &&
    ... 
    condition_testN) {
  if-body
} else {
  else-body
}
... rest of code

In Assembly

```
cond1 == false
  test condition1
  branch
cond2 == false
  test condition2
  branch
  ...
condN == false
  test conditionN
  branch
  skip
  if-body
  if-body stmt1
  ...
  stmtN
  branch always
  else_body:
  else-body stmt1
  ...
  stmtN
  end_if:
  ... rest of code
```

Unconditional branch to skip else-body after execution of if-body

The else-body is branched to on the first condition that is false.
The if-body is executed if all conditions are true.
Complex Condition Example (&&)

In C

```c
uint16 i, j, k;
if ((i < k) && (j != 20)) {
    if-body
} else {
    else-body
}
... rest of code
```

In Assembly

```assembly
mov k, W0 ; W0 = k
cp i ; i - WREG
bra GEU, else_body ; skip if-body when i >= k
mov #20, W0 ; W0 = 20
cp j ; j - WREG
bra Z, else_body ; skip if-body when j == 20
```

```
if-body stmt1
... stmtN
bra end_if ; skip else-body
```

```
else_body:
else-body stmt1
... stmtN
end_if:
```

... rest of code
Complex Conditions (||)

**In C**

```c
if (condition_test1 ||
    condition_test2 ||
    ...
    condition_testN ) {
    if-body
} else {
    else-body
}
... rest of code
```

**In Assembly**

```
cond1 == true
  test condition1
  branch

cond2 == true
  test condition2
  branch
  ...

condN-1 == true
  test conditionN-1
  branch
  test conditionN
  branch
  if_body:
    if-body stmt1
    ...
    stmtN
  branch always

skip
else-body:
  else-body stmt1
  ...
  stmtN

condN == false
  This branch taken to else-body if all conditions are false.

end_if:
  ... rest of code
```

The *if-body* is branched to on the first condition that is true.
The *else-body* is executed if all conditions are false.

Careful of last branch!
Different from others!
Complex Conditions (||), alternate method

In C
if (condition_test1 || condition_test2 || ...
   condition_testN ) {
   if-body
} else {
   else-body
}
... rest of code

In Assembly
cond1 == true
   test condition1 branch
   test condition2 branch
   ...
cond2 == true
   test conditionN-1 branch
condN-1 == true
   test conditionN branch
condN == true
   branch always
   if_body:
      if-body stmt1
      ... stmtN
      else_body:
      else-body stmt1
      ... stmtN
end_if:
   ... rest of code

In this solution, all branches are for the true condition. This requires an extra unconditional branch.

This branch taken to else-body if all conditions are false.

The if-body is branched to on the first condition that is true.
The else-body is executed if all conditions are false.
Complex Condition Example (||)

In C

```c
uint16 i, j, k;
uint16 p,q;
if ((i < k) ||
    (j == p) ||
   (q != 0)) {
    if-body
} else {
    else-body
}
... rest of code
```

In Assembly

```assembly
In body:
    mov k, W0
    cp i
    bra LTU, if_body
    mov p, W0
    cp j
    bra Z, if_body
    cp0 q
    bra Z, else_body

else-body:
    else-body stmt1
    ... stmtN

end_if:
    ... rest of code
```

```assembly
; W0 = j
; i - WREG
; execute if-body when i < k
; W0 = p
; j - WREG
; execute if-body when j == p
; q - 0
; skip if-body when q == 0

Can be replaced with:

```assembly
bra NZ, if_body ; true cond!
bra else else_body
```

Copyright Delmar Cengage Learning 2008. All Rights Reserved.
From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.
**while loop Structure**

In C

```c
while (condition_test) {
    while-body
}
...rest of code
```

In Assembly

```
cond == false
    skip
    while-body
cond == true,
    test condition
    branch
    execute while-body
while-body stmt1
    ... stmtN
    branch always
end_while:
    ... rest of code
```

Unconditional branch to return to the top of the while loop.

The *while-body* is not executed if the condition test is initially false.

Observe that at the end of the loop, there is a jump back to *top_while* after the while-body is performed. The body of a *while* loop will not execute if the condition test is initially false.
**while loop Example**

In C

```c
uint16 k, j;

while (k > j) {
    while-body
}
... rest of code
```

In Assembly

```
top_while:
    mov j, W0 ; W0 = j
    cp k ; k - WREG
    bra LEU, end_while
    while-body stmt1 ...
    stmtN
    bra top_while
end_while:
... rest of code
```

skip `while-body` if `k <= j`
**do-while** loop Structure

**In C**
```
do {
    do-while-body
} while (condition_test)
... rest of code
```

**In Assembly**
```
top_do_while:
    do-while-body stmt1 ... stmtN
    test condition cond == true
    branch
    cond == false, exit loop
    ... rest of code
```

On true condition, return to the top of the **do-while** loop.

The **do-while-body** is always executed at least once.
**do-while Example**

**In C**
```c
uint16 k, j;

do {
    while-body
} while (k > j);
... rest of code
```

**In Assembly**
```
top_do_while:
    while-body stmt1
        ... stmtN
    mov j,W0 ;W0 = j
    cp k ;k - WREG
    bra GTU, top_do_while
    ... rest of code
```
return to top of

`do{}while` loop if

k > j
Aside: *for* loops in C

A *for* loop is just another way to write a *while* loop. Typically used to implement a counting loop (a loop that is executed a fixed number of times).

```c
#include <stdio.h>

uint16 i,j;

i = 0;
while (i != 10) {
    k = k + j;
    i++;
}
/* ..do stuff..*/
```

These statements executed 10 times. Both code blocks are equivalent.
The ‘bit test’ instruction: \texttt{btst f, \#bit4} is useful for testing a single bit in an operand and branching on that bit. The complement of the bit is copied to the Z flag (if bit is 0, then \(Z=1\); if bit is 1, then \(Z=0\)).

\begin{center}
\begin{tabular}{llll}
\textbf{In C} & & \textbf{In Assembly} \\
\begin{verbatim}
uint16 k, j;
if (k & 0x0080) {
    if-body
}
... rest of code
\end{verbatim} & \begin{verbatim}
btst k,#7 ; same as k & 0x0080
bra Z, end_if ; skip if-body when bit #7 is 0
if-body stmt1
    ... stmtN
end_if:
    ... rest of code
\end{verbatim} & \\

This is testing bit #7 of \(k\)
\end{tabular}
\end{center}

Other forms of ‘bit test’ are available; they will not be discussed.
What instructions do you use?

You will discover that there are many ways for accomplishing the same thing using different instruction sequences.

Which method do you use?

The method that you understand......(and have not MEMORIZED), since memorization of code fragments will fail if faced with a situation different from what is memorized.

Your grade will not be penalized for ‘inefficient code’ in this course since this is your first look at assembly language programming.

Your grade will always be penalized for incorrect code – “close” does not count.
What do you need to know?

- Bitwise logical operations (and, or, xor, complement)
  - Clearing/setting/complementing groups of bits
- Bit set/clear/toggle instructions
- Shift left (<<), shift right (>>)
- Status register (C, Z flags)
- ==, !=, >, <, >=, <= tests on 8-bit, 16-bit unsigned variables
  - Conditional execution
- Loop structures