Microcomputers

Logical and Control Operations

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

• Bitwise Inclusive OR operation
  \[
  \text{IOR.}(B) \ Wb, Ws, Wd \quad (Wb) \mid (Ws) \rightarrow Wd \quad j = k \mid i;
  \]
  \[
  \text{IOR.}(B) f \quad (f) \mid (WREG) \rightarrow f \quad j = j \mid k;
  \]
  \[
  \text{IOR.}(B) f, WREG \quad (f) \mid (WREG) \rightarrow WREG \quad j = j \mid k;
  \]
  \[
  \text{IOR.}(B) \ #\text{lit10}, Wn \quad \text{lit10} \mid (Wn) \rightarrow Wn \quad j = j \mid \text{literal};
  \]

• Bitwise XOR operation
  \[
  \text{XOR.}(B) \ Wb, Ws, Wd \quad (Wb) \^ (Ws) \rightarrow Wd \quad j = k \^ i;
  \]
  \[
  \text{XOR.}(B) f \quad (f) \^ (WREG) \rightarrow f \quad j = j \^ k;
  \]
  \[
  \text{XOR.}(B) f, WREG \quad (f) \^ (WREG) \rightarrow WREG \quad j = j \^ k;
  \]
  \[
  \text{XOR.}(B) \ #\text{lit10}, Wn \quad \text{lit10} \^ (Wn) \rightarrow Wn \quad j = j \^ \text{literal};
  \]

• Bitwise complement operation
  \[
  \text{COM.}(B) \ Ws, Wd \quad (Ws) \rightarrow Wd \quad j = \neg k;
  \]
  \[
  \text{COM.}(B) f \quad (f) \rightarrow f \quad j = \neg j;
  \]
  \[
  \text{COM.}(B) f, WREG \quad (f) \rightarrow WREG \quad j = \neg k;
  \]
Bit-wise Logical operations (continued)

- Clear ALL bits
  
  CLR.\{B\} f          \[0\rightarrow f\] \[j = 0;\]
  CLR.\{B\} WREG       \[0\rightarrow\text{WREG}\] \[j = 0;\]
  CLR.\{B\} Wd         \[0\rightarrow Wd\] \[j = 0;\]

- Set ALL bits
  
  SETM.\{B\} f         \[111..111\rightarrow f\]
  SETM.\{B\} WREG      \[111..111\rightarrow\text{WREG}\]
  SETM.\{B\} Wd        \[111..111\rightarrow Wd\]

Clearing a group of bits

<table>
<thead>
<tr>
<th>Location</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Memory</td>
<td>Location contents</td>
</tr>
<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>

Clear upper four bits of i.

In C:

```c
uint8 i;
i = i & 0x0F;
```

In PIC24 \(\mu\)C 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
### Setting a group of bits

<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>

Set bits b3:b1 of j

In C:
```c
uint8 j;
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

### Complementing a group of bits

<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>

Complement bits b7:b6 of k

In C:
```c
uint8 k;
k = k ^ 0xC0; // The 'mask'
```

In PIC24 μC assembly
```asm
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
Complement all bits

Complement all bits of k

In C:
```c
uint8 k;
k = ~k ;
```

In PIC24 μC assembly
```assembly
com.b k ; k = ~k
```

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>

After complement
```c
uint8 k;
k = ~k ;
```
```
```assembly
com.b k ; k = ~k
```

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><code>bset(.b) Ws, #bit4</code></td>
<td>1 → <code>Ws&lt;bit4&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>bset(.b) f, #bit4</code></td>
<td>1 → <code>f&lt;bit4&gt;</code></td>
</tr>
<tr>
<td>Bit Clear</td>
<td><code>bclr(.b) Ws, #bit4</code></td>
<td>0 → <code>Ws&lt;bit4&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>bclr(.b) f, #bit4</code></td>
<td>0 → <code>f&lt;bit4&gt;</code></td>
</tr>
<tr>
<td>Bit Toggle</td>
<td><code>btg(.b) Ws, #bit4</code></td>
<td>~<code>Ws&lt;bit4&gt;</code> → <code>Ws&lt;bit4&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>btg(.b) f, #bit4</code></td>
<td>~<code>f&lt;bit4&gt;</code> → <code>f&lt;bit4&gt;</code></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:
```assembly
bclr.b k, #7
j = 0xB2 = 1011 0010
bset.b j, #2
j = 0xB6 = 1011 0110
btg i, #5
i = 0x0C = 0000 1100
```

status Register

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

<table>
<thead>
<tr>
<th>Status Register</th>
<th>DC</th>
<th>IPL2</th>
<th>IPL1</th>
<th>IPL0</th>
<th>RA</th>
<th>N</th>
<th>OV</th>
<th>Z</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Register high byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Status Register low byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **C**: Carry
- **Z**: Zero
- **OV**: Overflow
- **N**: Negative
- **RA**: Repeat Loop Active
- **IPL[2:0]**: Interrupt Priority Level
- **DC**: Decimal Carry
- **-**: Unimplemented

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 26-2 in PIC24HJ128GP502 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</th>
<th>Status affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>C,DC,Z,OV,N</td>
</tr>
<tr>
<td>MOV</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>none</td>
</tr>
<tr>
<td>MOV</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>N,Z</td>
</tr>
</tbody>
</table>

Addition: Carry, Zero Flags

Zero flag is set if result is zero.

In addition, carry flag is set if there is a carry out of the MSbit.
In byte (8-bit) mode, C=1 if sum > 255 (0xFF)
In word (16-bit) mode, C=1 if sum > 65535 (0xFFFF)

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

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

Zero flag is set if result is zero.

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

\[
\begin{array}{ccc}
0xF0 & 0x00 & 0x01 \\
-0x20 & -0x00 & -0xFF \\
\hline
0xD0 & 0x00 & 0x02 \\
\end{array}
\]

Z=0, C=1

Z=1, C=1

Z=0, C=0

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 + (~B) + 1

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

\[
\begin{array}{ccc}
0xF0 & 0x20 & 0x01 \\
-0x20 & \sim0x20 & +0xFF \\
\hline
0xD0 & 0x00 & 0x02 \\
\end{array}
\]

Z=0, C=1

Carry out of MSB, so C=1

No borrow, C=1
**C Shift Left, Shift Right**

*logical* Shift right \(i >> 1\)
all bits shift to right by one, ‘0’ into MSB (8-bit right shift shown)

\[
\begin{array}{cccccccc}
\text{b7} & \text{b6} & \text{b5} & \text{b4} & \text{b3} & \text{b2} & \text{b1} & \text{b0} \\
\end{array}
\]
original value

\[
\begin{array}{cccccccc}
\text{0} & \text{b7} & \text{b6} & \text{b5} & \text{b4} & \text{b3} & \text{b2} & \text{b1} \\
\end{array}
\]
i \(>> 1\) (right shift by one)

Shift left \(i << 1\)
all bits shift to left by one, ‘0’ into LSB (8-bit left shift shown)

\[
\begin{array}{cccccccc}
\text{b7} & \text{b6} & \text{b5} & \text{b4} & \text{b3} & \text{b2} & \text{b1} & \text{b0} \\
\end{array}
\]
original value

\[
\begin{array}{cccccccc}
\text{b6} & \text{b5} & \text{b4} & \text{b3} & \text{b2} & \text{b1} & \text{b0} & \text{0} \\
\end{array}
\]
i \(<< 1\) (left shift by one)

---

**PIC24 Family Unsigned Right Shifts**

<table>
<thead>
<tr>
<th>Logical Shift Right</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 \rightarrow \text{f})</td>
</tr>
<tr>
<td></td>
<td>LSR{.B} f,WREG</td>
<td>(f &gt;&gt; 1 \rightarrow \text{WREG})</td>
</tr>
<tr>
<td>Log. Shift Right Ws</td>
<td>LSR{.B} Ws,Wd</td>
<td>(Ws &gt;&gt; 1 \rightarrow \text{Wd})</td>
</tr>
<tr>
<td>Log. Shift Right by short Literal</td>
<td>LSR Wb, #lit4, Wd</td>
<td>(Wb &gt;&gt; \text{lit4} \rightarrow \text{Wd})</td>
</tr>
<tr>
<td>Log. Shift Right by Ws</td>
<td>LSR Wb, Ws, Wd</td>
<td>(Wb &gt;&gt; Ws \rightarrow \text{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**

<table>
<thead>
<tr>
<th>Cflag</th>
<th>b7 b6 b5 b4 b3 b2 b1 b0</th>
<th>0</th>
<th>8-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cflag</td>
<td>b15 b14 . . . . . b1 b0</td>
<td>0</td>
<td>16-bit</td>
</tr>
</tbody>
</table>

**Descri:**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift left f</td>
<td>f &lt;&lt; 1 (\rightarrow f)</td>
</tr>
<tr>
<td>Shift left Ws</td>
<td>Ws (\rightarrow Wd)</td>
</tr>
<tr>
<td>Shift left by short Literal</td>
<td>Wb (\rightarrow Wd)</td>
</tr>
<tr>
<td>Shift left by Ws</td>
<td>Wb (\rightarrow Wd)</td>
</tr>
</tbody>
</table>

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:

We will not cover these instructions since their need is specialized; the shift instructions are sufficient for C language shift operations.
### C Shift operations

<table>
<thead>
<tr>
<th>In C</th>
<th>In Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>uint16 u16_p, u16_k;</code></td>
<td><code>lsr.b u8_i ;i = i &gt;&gt; 1;</code></td>
</tr>
<tr>
<td><code>uint8 u8_i;</code></td>
<td><code>lsr.b u8_i ;i = i &gt;&gt; 1;</code></td>
</tr>
<tr>
<td><code>u8_i = u8_i &gt;&gt; 2;</code></td>
<td><code>mov u16_p, w1 ;w1 = p</code></td>
</tr>
<tr>
<td><code>u16_k = u16_p &lt;&lt; 10;</code></td>
<td><code>sl w1, #10, w1 ;w1 = w1 &lt;&lt; 10</code></td>
</tr>
<tr>
<td></td>
<td><code>mov w1, u16_k ;k = w1</code></td>
</tr>
</tbody>
</table>

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

---

### Mixed 8-bit, 16-bit operations

#### (a) In Assembly (incorrect)

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

<table>
<thead>
<tr>
<th>u16_p</th>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ W0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>n8_i</td>
<td></td>
</tr>
</tbody>
</table>

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

#### (b) In Assembly (correct)

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

<table>
<thead>
<tr>
<th>u16_p</th>
<th>MSB</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ W0</td>
<td>0</td>
<td>u8_i</td>
</tr>
</tbody>
</table>

The upper 8 bits of W0 are now zero; unsigned 8-bit variables should be zero-extended before use in 16-bit operations.
**Arithmetic Example**

### (a) In C

```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

```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.

---

**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.

```assembly
btsc{.b} f, #bit4 ; skips next instruction if f<#bit4> is clear ('0')
bssc{.b} f, #bit4 ; skips next instruction if f<#bit4> is set ('1')
```

- Bit test instructions are just the first of many different methods of performing conditional execution in the PIC24 µC.
### C Conditional Tests

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==, !=</td>
<td>equal, not-equal</td>
</tr>
<tr>
<td>&gt;, &gt;=</td>
<td>greater than, greater than or equal</td>
</tr>
<tr>
<td>&lt;, &lt;=</td>
<td>less than, less than or equal</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</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` is not the same as `~i`
- `i = 0xA0`
- `! (i) ➞ 0`
- `~ (i) ➞ 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
```

Copyright Delmar Cengage Learning 2008. All Rights Reserved.

if/else Statement Format in C

```c
if (condition_test) {
    if-body  // Executed when condition_test is non-zero (true)
} else {
    else-body  // 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.

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

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

Could also write:

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

```
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. ===)

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

```
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.

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

is read as:

If ((i is nonzero) AND (j is nonzero)) then do this.

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

is read as:

If ((i bitwise AND j) is nonzero) then do this.

---

**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 non-zero.

```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....

---
Non-Zero Test

labels for SFRs defined in p24Hxxxx.inc; use for clarity!!!!

In C

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

In Assembly

```assembly
mov k, k ; k = k, affects N, Z flags
btac SR,#1 ; skip if Z = 0 (Z is SR<1>)
```

The `mov k` instruction just moves `k` back onto itself! Does no useful work except to affect the Z, N flags.

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.

### In C
```c
uint16 i, j;
if (i) {
    // if-body code
} else {
    // else-body code
} ...
```

### In Assembly
```assembly
mov  i ; i = i, affects N, Z flags
bra  Z, end_if ; skip if-body when Z=1 (i is 0)
..if-body stmt1
..if-body stmtN
end_if:
..rest of code...
```

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
} ...
```

### In Assembly
```assembly
cond == false

test condition : set status flags
branch
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
}
```

In Assembly

```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.

---

Inequality tests using Z, C flags and subtraction

(a) \( k - j \)

\[
\begin{align*}
&k \geq j & k < j \\
&C = 1 & C = 0 \text{ or } Z = 1
\end{align*}
\]

(b) \( j - k \)

\[
\begin{align*}
&k \geq j & k < j \\
&C = 0 \text{ or } Z = 1 & C = 1
\end{align*}
\]

Note: \( k \leq j \) is \( \neg(k > j) \) is \( \neg(C \& \neg Z) \) is \( \neg(C | Z) \) by DeMorgan's law. Similarly, \( k < j \) is \( \neg(k \geq j) \) is \( \neg(C) \) is \( \neg(C) \).

Note: \( k < j \) is \( \neg(k \geq j) \) is \( \neg(C \& Z) \) is \( \neg(C | Z) \) by DeMorgan's law. Similarly, \( k \leq j \) is \( \neg(k > j) \) is \( \neg(C) \).
**k>j test using k-j**

In C

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

In Assembly

```assembly
mov j, W0 ;W0 = j
sub k, WREG ;W0 = k - j
bra KC, end_if ;skip if-body when C = 0 (k < j)
bra Z, end_if ;or skip if-body when Z = 1 (k == j)
    if-body stmt1
    ... stmtN
    end_if:
    ... rest of code
```

False condition of k>j is k <= j, so need branches that accomplish this.

The false condition of k>j is k <= j, so use k <= 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 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 test one or more 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.
### Unsigned Comparison (> test)

**In C**
```c
uint16 k, j;
if (k > j) {
    if-body
}
... rest of code
```

**In Assembly**
```assembly
mov j, W0 ; W0 = j
cp 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.

---

### If-else Example

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

**In Assembly**
```assembly
mov j, W0 ; W0 = j
cp k ; k - WREG
bra GTU, else_body ; skip if-body when k > j
    if-body stmt1
    ... stmtN
else_body:
bra end_if ; use unconditional branch to skip else-body after executing if-body
    else-body stmt1
    ... stmtN
end_if:
    ... 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.”
**Unsigned literal Comparison**

(a) In C

```c
uint16 k;

if (k > 10) {
    // if-body code
    // ...rest of code...
}
```

In Assembly

```assembly
mov k, W0
if (k > 10) {
    bra LEU, end_if
    ; skip if-body when k <= 10
    ...if-body stmt1
    ...if-body stmtN
    end_if:
    ...rest of code..
}
```

(b) In C

```c
uint16 k;

if (k > 520) {
    // if-body code
    // ...rest of code...
}
```

In Assembly

```assembly
mov (#520), W0
if (k > 520) {
    bra LEU, end_if
    ; skip if-body when k <= 520
    ...if-body stmt1
    ...if-body stmtN
    end_if:
    ...rest of code..
}
```

---

**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.

---

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

Electrical & Computer Engineering - Microcomputers Lecture 3-45
**switch Statement in assembly language**

In C
```c
unsigned u8_i;
unsigned 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;
} // end switch
```

In Assembly
```assembly
mov.b u8_i, WREG
cp.b W0, #1
bra NZ, case_2
ise u16_k
bra end_switch case_2:
cp.b W0, #2
bra NZ, case_3
dec u16_j
bra end_switch case_3:
cp.b W0, #3
bra NZ, default
mov u16_k, #0
add u16_j
bra end_switch default:
mov u16_j, W0
sub u16_k
end_switch:
```

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>
Complex Conditions (&&)

In C

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

In Assembly

```
test condition1
branch cond1 == false

cond2 == false
branch test condition2
...

condN == false
branch test conditionN
...

if-body
skip

stmt1
stmtN
branch always
else_body:
  else-body stmt1
  ...
  stmtN
end_if:

... rest of code
```

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

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

In 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
} ...
```

In Assembly:

```
cond1 == true
```  
```
cond2 == true
```  
```
...  
```
condN == false
```  
```
This branch taken to else-body if all conditions are false.
```
```
branch always
```  
```
if_body:
```  
```
if-body stmt1
```  
```
... stmtN
```  
```
else_body:
```  
```
else-body stmt1
```  
```
... stmtN
```  
```
end_if;
```  
```
... rest of code
```
```
skip
```
```
branch
```
```
... 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 Condition Example (||)

In C:

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

In Assembly:

```
mov k, W0
```  
```i = j
```  
```i - WREG
```  
```mov p, W0
```  
```j = WREG
```  
```bra LTU, if_body
```  
```
; execute if-body when i < k
```  
```
bra Z, if_body
```  
```
; execute if-body when j == p
```  
```
bra Z, else_body
```  
```
; skip if-body when q == 0
```
```
if_body:
```  
```
if-body stmt1
```  
```
... stmtN
```  
```
else-body:
```  
```
else-body stmt1
```  
```
... stmtN
```  
```
end_if;
```  
```
... rest of code
```
```
Can be replaced with:
```
bra NZ, if_body ; true cond
```  
```
br else_body
```
```
... rest of code
```

Can be replaced with:

```
br else_body
```

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

Electrical & Computer Engineering – Microcomputers

Lecture 3-51

Lecture 3-52
**while loop Structure**

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

In Assembly
```
top_while:
    test condition branch
    cond == false
    skip
    while-body
    cond == true,
    execute while-body while-body stmtN
    ... 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
```
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 LSU, end_while
    while-body stmt1
    ... stmtN
    bra top_while
    end_while:
    ... rest of code
```

Copyright Delmar Cengage Learning 2008. All Rights Reserved.
From: Reese/Brace/Chen, "Microcontrollers: From Assembly to C with the PIC24 Family".
**do-while loop Structure**

**In C**

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

**In Assembly**

```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**

```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
uint16 i,j;

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

```c
unsigned int i,j;
for (i = 0; i!= 10; i++){
    k = k + j;
}
/* do stuff */
```

These statements executed 10 times. Both code blocks are equivalent.

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.
What do you need to know?

- Logical operations (and, or, xor, complement)
- Clearing/setting/complementing groups of bits
- Bit set/clear/test instructions
- Shift left (<<), shift right (>>)
- ==, !=, >, <, >=, <= tests on 8-bit, 16-bit unsigned variables
- Loop structures