 PIC24 Instruction Set Features

A large and complex instruction set with a variable format:

- **Different numbers of operands**
  - No operands; Example: RETURN
    meaning return from subroutine

  - Implicit operands; Example: ADD{.B} f
    meaning memory[f] <= memory[f] + WREG

  - One operand; Example: MOV{.B} f
    meaning memory[f] <= memory[f]

  - Two operand: Example MOV Wns, f
    meaning memory[f] <= Wns

  - Three Operand: ADD Wb, Ws, Wd
    meaning Wd <= Wb + Ws*
    * Other addressing modes are possible

- **Different sizes and types of immediate values**
  - A 13 bit immediate address as in: ADD{.B} f
    - Example: ADD 0x1FFF

  - A 15 bit immediate address as in: MOV f, Wnd
    - Example: MOV 0x7FFF, W1

  - A 5 bit literal as in: AND{.B} Wb, #lit5, Wd
    - Example: AND W1, #0x1F, W2

  - A 10 bit literal as in: ADD{.B} #lit10, Wn
    - Example: ADD #0x3FF, W1

  - Other literal sizes are: #4, #8, #14, and #16

- An accumulator-style working register: WREG (another name for W0)
• Over 60 different operators (i.e., distinct opcode mnemonics)
## Address Mode Encoding in PIC24 Instruction Set

<table>
<thead>
<tr>
<th>Register Designation</th>
<th>Addressing Mode</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| Wn, Wns, Wd, Wd | (1) register direct for W0-W15 | MOV Wns, f direct: MOV W3, 0x1002 | memory[f] <= Wns
memory[1002] <= W3 |
| Wd, Ws | Any combination of (1) register direct, and (2) register indirect including pre- and post-incrementing for registers W0-W15 | ADD Wb, Ws, Wd direct: ADD W1, W2, W3 | Wd <= Wb + Ws
W3 <= W1 + W2
| Wso, Wdo | Any combination of (1) register direct, (2) register indirect including pre- and post-incrementing, and (3) register offset indirect for registers W0-W15 | MOV {.B} Wso, Wdo direct: MOV W1, W2 | Wdo <= Wso
W2 <= W1
memory[W1] <= memory[W2] |
| &nbsp; | | indirect: MOV [W1], [W2] | |
| &nbsp; | | indirect with pre-increment: MOV [++W1], [++W2] | |
| &nbsp; | | indirect with pre-decrement: MOV [--W1], [--W2] | |
| &nbsp; | | indirect with post-increment: MOV [W1++], [W2++] | |
| &nbsp; | | indirect with post-decrement: MOV [W1--], [W2--] | |
| &nbsp; | | offset indirect for source: MOV [W1 + W3], W2 | |
| &nbsp; | | offset indirect for destination: MOV W1, [W2 + W3] | |
| &nbsp; | | | |
| &nbsp; | | &nbsp; | |
| &nbsp; | | &nbsp; | |
| &nbsp; | | &nbsp; | |

* one word of memory
PIC 24 Addressing Mode Exercise

For each questions a-g, assume the memory/register contents shown below at the start of instruction execution and give the modified memory location or register and its contents after instruction execution.

<table>
<thead>
<tr>
<th>Location</th>
<th>Value</th>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>0x1006</td>
<td>0x1000</td>
<td>0x382A</td>
</tr>
<tr>
<td>W1</td>
<td>0x0006</td>
<td>0x1002</td>
<td>0xFB80</td>
</tr>
<tr>
<td>W2</td>
<td>0x8345</td>
<td>0x1004</td>
<td>0x4D19</td>
</tr>
<tr>
<td>W3</td>
<td>0x1000</td>
<td>0x1006</td>
<td>0xE7C0</td>
</tr>
<tr>
<td>W4</td>
<td>0x1006</td>
<td>0x1008</td>
<td>0xFF00</td>
</tr>
</tbody>
</table>

a. MOV 0x1002, WREG
b. MOV #0x1002, W0
c. MOV [W3], [W4]
d. MOV [W0++], W4
e. MOV [++W0], W4
f. MOV [W1 + W3], W2
g. AND W0, W1, W0
A Simple Program

In this class, will present programs in C form, then translate (compile) them to PIC24 μC assembly language.

C Program equivalent

```c
#define avalue 100
uint8 i,j,k;

i = avalue;   // i = 100
i = i + 1;    // i++, i = 101
j = i;        // j is 101
j = j - 1;    // j--, j is 100
k = j + i;    // k = 201
```

A uint8 variable is 8-bits (1 byte)
Where are variables stored?

When writing assembly language, can use any free data memory location to store values, it your choice.

A logical place to begin storing data in the first free location in data memory, which is 0x0800 (Recall that 0x0000-0x07FF is reserved for SFRs).

Assign \(i\) to \(0x0800\), \(j\) to \(0x0801\), and \(k\) to \(0x0802\). Other choices could be made.
C to PIC24 Assembly

```
mov.b #100, W0       ; WREG = 100 = 0x64
mov.b WREG, 0x0800   ; i = WREG

inc.b 0x0800         ; i = i + 1

mov.b 0x0800, WREG   ; WREG = i
mov.b WREG, 0x0801   ; j = WREG

dec.b 0x0801         ; j = j - 1

mov.b 0x0800, WREG   ; WREG = i
add.b 0x0801, WREG   ; WREG = j + WREG
mov.b WREG, 0x0802   ; k = WREG
```

*i is location 0x0800, j is location 0x0801, k is location 0x0802*

**Comments:** The assembly language program operation is not very clear. Also, multiple assembly language statements are needed for one C language statement. Assembly language is more *primitive* (operations less powerful) than C.

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

V 0.1
PIC24 Assembly to PIC24 Machine Code

- Could perform this step manually by determining the instruction format for each instruction from the data sheet.
- Much easier to let a program called an assembler do this step automatically
- The MPLAB™ Integrated Design Environment (IDE) is used to assemble PIC24 programs and simulate them
  - Simulate means to execute the program without actually loading it into a PIC24 microcontroller
.include "p24Hxxxx.inc"
.global __reset
.bss ;reserve space for variables
i: .space 1
j: .space 1
k: .space 1
.text ;Start of Code section
__reset: ; first instruction located at __reset label
  mov #__SP_init, W15 ;;initialize stack pointer
  mov #__SPLIM_init,W0
  mov W0,SPLIM ;;initialize Stack limit reg.
  avalue = 100
; i = 100;
  mov.b #avalue, W0 ; W0 = 100
  mov.b WREG,i ; i = 100
; i = i + 1;
  inc.b i ; i = i + 1
; j = i
  mov.b i,WREG ; W0 = i
  mov.b WREG,j ; j = W0
; j = j - 1;
  dec.b j ; j = j - 1
; k = j + i
  mov.b i,WREG ; W0 = i
  add.b j,WREG ; W0 = W0+j (WREG is W0)
  mov.b WREG,k ; k = W0
done:
  goto done ;loop forever

This file can be assembled by the MPLAB™ assembler into PIC24 machine code and simulated. Labels used for memory locations 0x0800 (i), 0x0801(j), 0x0802(k) to increase code clarity.
mptst_byte.s (cont.)

.include "p24Hxxxx.inc"
.global __reset

.bss ;reserve space for variables
i: .space 1
j: .space 1
k: .space 1

An *assembler directive* is not a PIC24 instruction, but an instruction to the assembler program. Assembler directives have a leading ‘.’ period, and are not case sensitive.

Include file that defines various labels for a particular processor. ‘include’ is an assembler directive.

Declare the __reset label as global – it is needed by linker for defining program start.

The .bss assembler directive indicates the following should be placed in data memory. By default, variables are placed beginning at the first free location, 0x800. The .space assembler directive reserves space in bytes for the named variables. i, j, k are labels, and labels are case-sensitive and must be followed by a ‘:’ (colon).
mptst_byte.s (cont.)

```asm
.text
__reset:  mov #__SP_init, W15
          mov #__SPLIM_init, W0
          mov W0, SPLIM
```

`.text` is an assembler directive that says what follows is code. Our first instruction must be labeled as `__reset`.

These move instructions initialize the stack pointer and stack limit registers – this will be discussed in a later chapter.

```asm
avalue = 100
```

The equal sign is an assembler directive that equates a label to a value.
```
mptst_byte.s (cont.)

; i = 100;
mov.b #avalue, W0 ; W0 = 100
mov.b WREG,i ; i = 100

; i = i + 1;
inc.b i ; i = i + 1

; j = i
mov.b i,WREG ; W0 = i
mov.b WREG,j ; j = W0

; j = j – 1;
dec.b j ; j= j – 1

; k = j + i
mov.b i,WREG ; W0 = i
add.b j,WREG ; W0 = W0+j
mov.b WREG,k ; k = W0
```

The use of labels and comments greatly improves the clarity of the program.

It is hard to over-comment an assembly language program if you want to be able to understand it later.

Strive for at least a comment every other line; refer to lines (WREG is W0)
A label that is the target of a `goto` instruction. Lables are *case sensitive* (instruction mnemonics and assembler directives are not case sensitive).

An assembler directive specifying the end of the program in this file.
Video tutorials

A number of videos illustrate important concepts; all are listed on the video page at http://www.reesemicro.com/site/pic24micro/Home/pic24-video-tutorials-1.

Available tutorials, which cover topics on the following pages of these lecture notes:

- **MPLAB IDE introduction** at http://www.ece.msstate.edu/courses/ece3724/main_pic24/videos/mplab_assem/index.htm
- **A simple assembly language program** at http://www.ece.msstate.edu/courses/ece3724/main_pic24/videos/assem_intro/index.htm
- **Simulation of this program** at http://www.ece.msstate.edu/courses/ece3724/main_pic24/videos/assem_intro2/index.htm
- **Converting the program from 8 to 16 bits** at http://www.ece.msstate.edu/courses/ece3724/main_pic24/videos/assem_intro3/index.htm
Now you try it! Here’s the assembly language program that we just discussed:  `mptst_byte.s`

Create an MPLAB project using the project wizard and run the program while viewing program memory, data memory and the special function registers.

Next we will discuss the implementation of flow control using the PIC24 instructions CP (compare) and BRA (branch).
status Register

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

### Status Register

<table>
<thead>
<tr>
<th>C</th>
<th>Z</th>
<th>OV</th>
<th>N</th>
<th>RA</th>
<th>IPL[2:0]</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>Carry</td>
<td>Zero</td>
<td>Overflow</td>
<td>Negative</td>
<td>Repeat Loop Active</td>
<td>Interrupt Priority Level</td>
<td>Decimal Carry</td>
<td>Unimplemented</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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.
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, 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>
Unsigned Comparison (> test)

For \( k > j \) test, use the LEU (less than or equal unsigned) branch to skip IF body if \( k \leq j \)
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
lop k
bra GTU, else_body
..if-body stmt1
..if-body stmtN
bra end_if
else_body:
   ..else-body stmt1
   ..else-body 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”.

V 0.1 44
Unsigned literal Comparison

(a) In C

```c
uint16 k;

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

In Assembly

```assembly
mov k, W0 ; W0 = k
.cp W0, #10 ; k - 10
bra LEU, end_if ; skip if-body when k <= 10
    ..if-body stmt1
    ..if-body stmtN
end_if: ; rest of code...
```

5-bit literal, unsigned range 0 to 31

(b) In C

```c
uint16 k;

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

In Assembly

```assembly
mov #520, W0 ; W0 = 520
.cp k ; k - WREG
bra LEU, end_if ; skip if-body when k <= 520
    ..if-body stmt1
    ..if-body stmtN
end_if: ; rest of code...
```

16-bit literal, unsigned range 0 to 65535
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
    dec u16_j
    bra end_switch

    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..

    ; W0 = u8_i  
    ; u8_i == 1?

    ; u16_k++
    ; break statement

    ; u8_i == 2?

    ; u16_j--
    ; break statement

    ; u8_i == 3?
OK to use W0 for computation after comparison is done.

    ; u16_j = u16_j + u16_k
    ; break statement

    ; u16_k = u16_k - u16_j

Note: The literal size in the CP instruction is 5-bits (unsigned values of 0-31).
```
In-Class Exercise: Flow Control using the PIC24 ISA

Overview

There are 2 main steps required to implement a flow control statement in assembly language:

1. Set the status register bits based on the value of the test expression. This step requires that you evaluate the test expression.
2. Branch to a statement label based on the value of the status register set in step 1.

Here's an example:

while (i < 10) { ... }

A partial translation of this loop structure into assembly language is:

```assembly
MOV #10, W0 ; setup to evaluate the test expression
CP i ; set status register bits by comparing i to 10; specifically calculate (i-10)
BRA GEU, end_loop ; branch if (i-10) is greater than or equal to 0; this means the loop test is not true
```

Note that (1) a compare instruction (CP) can be used to set the status register bits, and (2) there are many forms of the branch instruction (BRA), one for each possible comparison outcome. See the PIC 24 instruction set for variations of the CP and BRA instructions.

Exercise

Implement the following C code segment in PIC24 assembly language. Use 16 bit variables.

```c
unsigned int cnt = 0;
for ( unsigned int i=0; i < 10; i++) {
    cnt = cnt + i;
}
```