Microcontroller (μC) vs. Microprocessor (μP)

- μC intended as a single chip solution, μP requires external support chips (memory, interface)
- μC has on-chip non-volatile memory for program storage, μP does not.
- μC has more interface functions on-chip (serial interfaces, analog-to-digital conversion, timers, etc.) than μP
- μC does not have virtual memory support (i.e., could not run Linux), while μP does.
- General purpose μPs are typically higher performance (clock speed, data width, instruction set, cache) than μCs
- Division between μPs and μCs becoming increasingly blurred
# Microchip PIC24 Family µC

<table>
<thead>
<tr>
<th>Features</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction width</td>
<td>24 bits</td>
</tr>
<tr>
<td>On-chip program memory (non-volatile, electrically erasable)</td>
<td>PIC24HJ32GP202 has 32Ki bytes/11264 instructions, architecture supports 24Mbytes/4Mi instructions</td>
</tr>
<tr>
<td>On-chip Random Access Memory (RAM), volatile</td>
<td>PIC24HJ32GP202 has 2048 bytes, architecture supports up 65536 bytes</td>
</tr>
<tr>
<td>Clock speed</td>
<td>DC to 80 MHz</td>
</tr>
<tr>
<td>16-bit Architecture</td>
<td>General purpose registers, 71 instructions not including addressing mode variants</td>
</tr>
<tr>
<td>On-chip modules</td>
<td>Async serial IO, I2C, SPI, A/D, three 16-bit timers, one 8-bit timer, comparator</td>
</tr>
</tbody>
</table>
The instruction register contains the machine code of the instruction currently being executed.

ALU (Arithmetic Logic Unit) is 16 bits wide, can accept as operands working registers or data memory.
Memory Organization

Memory on the PIC24 μC family is split into two types: **Program Memory** and **Data Memory**.

PIC24 instructions are stored in **program memory**, which is **non-volatile** (contents are retained when power is lost).

A PIC24 instruction is 24 bits wide (3 bytes). PIC24HJ32GP202 program memory supports 11264 instructions; the PIC24 architecture can support up to 4M instructions.

PIC24 data is stored in **data memory**, also known as the file registers, and is a maximum size of 65536 x 8. Data memory is **volatile** (contents are lost when power is lost).
Program Memory

<table>
<thead>
<tr>
<th>MSW Address</th>
<th>most significant word</th>
<th>least significant word</th>
<th>PC Address (LSW Address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x000001</td>
<td>00000000</td>
<td></td>
<td>0x000000</td>
</tr>
<tr>
<td>0x000003</td>
<td>00000000</td>
<td></td>
<td>0x000002</td>
</tr>
<tr>
<td>0x000005</td>
<td>00000000</td>
<td></td>
<td>0x000004</td>
</tr>
<tr>
<td>0x000007</td>
<td>00000000</td>
<td></td>
<td>0x000006</td>
</tr>
</tbody>
</table>

Program Memory

‘Phantom’ Byte (read as ‘0’)  Instruction Width

Figure redrawn by author from Fig 3-2 found in the PIC24HJ32GP202/204 datasheet (DS70289A), Microchip Technology Inc.

PC is 23 bits wide, but instructions start on even word boundaries (the PC least significant bit is always 0), so the PC can address 4 Mi instructions.

Locations 0x000000- 0x0001FF reserved, User program starts at location 0x000200.
Data Memory Organization

Data memory for PIC24HJ32GP202

2048 byte SFR space

2048 byte SRAM space

0x0001

0x07FF

0x0801

0x0FFF

0x10FF

0x1FFF

Unimplemented on PIC24HJ32GP202

MSB = Most Significant Byte

LSB = Least Significant Byte

Figure redrawn by author from Fig 3-3 found in the PIC24HJ32GP202/204 datasheet (DS70289A), Microchip Technology Inc.

Copyright Delmar Cengage Learning 2008. All Rights Reserved.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.
Special Function Registers (SFRs)

**Special Function Registers** (SFR) are addressed like normal data memory locations but have specified functionality tied to hardware subsystems in the processor. We typically refer to SFRs by name (W0, T3CON, STATUS, etc) instead of by address.

There are many SFRs in the PIC24 μC – they are used as control registers and data registers for processor subsystems (like the serial interface, or the analog-to-digital converter). We will cover their use and names as we need to.

SFRs live in the address range 0x0000 to 0x07FE in data memory. See the datasheet for a complete list of SFRs.

Other locations in data memory that are not SFRs can be used for storage of temporary data; they are not used by the processor subsystems. These are sometimes referred to as GPRs (general purpose registers). MPLAB refers to these locations as file registers.
8-bit, 16-bit, 32-bit Data

We will deal with data that is 8 bits, 16 bits (2 bytes), and 32 bits (4 bytes) in size. Initially we will use only 8 bit and 16 bit examples.

<table>
<thead>
<tr>
<th>Size</th>
<th>Unsigned Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bits</td>
<td>0 to $2^8$-1 (0 to 255, 0 to 0xFF)</td>
</tr>
<tr>
<td>16-bit</td>
<td>0 to $2^{16}$-1 (0 to 65536, 0 to 0xFFFF)</td>
</tr>
<tr>
<td>32-bit</td>
<td>0 to $2^{32}$-1 (0 to 4,294,967,295, 0 to 0xFFFFFFFF)</td>
</tr>
</tbody>
</table>

The lower 8 bits of a 16-bit value or of a 32-bit value is known as the Least Significant Byte (LSB).

The upper 8 bits of a 16-bit value or of a 32-bit value is known as the Most Significant Byte (MSB).
Storing Multi-byte Values in Memory

16-bit and 32-bit values are stored in memory from least significant byte to most significant byte, in increasing memory locations (little endian order).

Assume the 16-bit value 0x8B1A stored at location 0x1000
Assume the 32-bit value 0xF19025AC stored at location 0x1002

<table>
<thead>
<tr>
<th>Location</th>
<th>Contents</th>
<th>LSB</th>
<th>Location</th>
<th>Contents</th>
<th>LSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0x1A</td>
<td></td>
<td>0x1000</td>
<td>0x8B1A</td>
<td></td>
</tr>
<tr>
<td>0x1001</td>
<td>0x8B</td>
<td></td>
<td>0x1002</td>
<td>0x25</td>
<td></td>
</tr>
<tr>
<td>0x1002</td>
<td>0xAC</td>
<td></td>
<td>0x1004</td>
<td>0xF190</td>
<td></td>
</tr>
<tr>
<td>0x1003</td>
<td>0x25</td>
<td></td>
<td>0x1006</td>
<td>?????</td>
<td></td>
</tr>
<tr>
<td>0x1004</td>
<td>0x90</td>
<td></td>
<td>0x1008</td>
<td>?????</td>
<td></td>
</tr>
<tr>
<td>0x1005</td>
<td>0xF1</td>
<td></td>
<td>0x1006</td>
<td>?????</td>
<td></td>
</tr>
</tbody>
</table>

Memory shown as 8 bits wide          Memory shown as 16 bits wide

The LSB of a 16-bit or 32-bit value must begin at an even address (be word aligned).
Data Transfer Instruction

Copies data from Source (src) location to Destination (dst) Location

(src) → dst ‘( )’ read as ‘contents of’

This operation uses *two operands*.

The method by which an operand ADDRESS is specified is called the *addressing mode*.

There are many different addressing modes for the PIC24.

We will use a very limited number of addressing modes in our initial examples.
## Data Transfer Instruction Summary

<table>
<thead>
<tr>
<th>Dest Source</th>
<th>Memory</th>
<th>Register direct</th>
<th>Register indirect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literal</td>
<td>X</td>
<td>MOV{.B} #lit8/16, Wnd&lt;br&gt;(lit \rightarrow Wnd)</td>
<td>X</td>
</tr>
<tr>
<td>Memory</td>
<td>X</td>
<td>MOV (f_{\text{ALL}}), Wnd&lt;br&gt;MV{.B} (f, {\text{WREG}})&lt;br&gt;((f_{\text{ALL}})) (\rightarrow Wnd/\text{WREG})</td>
<td>X</td>
</tr>
<tr>
<td>Register direct</td>
<td>MOV (Wns, f_{\text{ALL}})&lt;br&gt;MV{.B} (W\text{REG}, f)&lt;br&gt;((Wns/W\text{REG})) (\rightarrow f_{\text{ALL}})</td>
<td>MOV{.B} W(so), (\text{Wdo})&lt;br&gt;((Wso)) (\rightarrow W\text{do})</td>
<td>MOV{.B} W(so), [(\text{Wdo})]&lt;br&gt;((Wso)) (\rightarrow (\text{Wdo}))</td>
</tr>
<tr>
<td>Register indirect</td>
<td>X</td>
<td>MOV{.B} [(Wso)], (\text{Wdo})&lt;br&gt;((Wso)) (\rightarrow W\text{do})</td>
<td>MOV{.B} [(Wso)], [(\text{Wdo})]&lt;br&gt;((Wso)) (\rightarrow (\text{Wdo}))</td>
</tr>
</tbody>
</table>

**Key:**
- MOV\{.B\} #lit8/16, Wnd<br>\(lit \rightarrow Wnd\)
- PIC24 assembly
- Data transfer

Yellow shows varying forms of the same instruction

\(f\): near memory (0…8095)  \(f_{\text{ALL}}\): all of memory (0…65534)
**MOV{.B} Wso, Wdo** Instruction

“Copy contents of Wso register to Wdo register”. General form:

\[
\text{mov\{.b\} Wso, Wdo} \quad (\text{Wso}) \rightarrow \text{Wdo}
\]

Wso is one of the 16 working registers W0 through W15 (‘s’ indicates Wso is an operand **source** register for the operation).

Wdo is one of the 16 working registers W0 through W15 (‘d’ indicates Wdo is an operand **destination** register for the operation).

\[
\begin{align*}
\text{mov} & \quad \text{W3, W5} \quad (\text{W3}) \rightarrow \text{W5} & \quad \text{(word operation)} \\
\text{mov.b} & \quad \text{W3, W5} \quad (\text{W3.LSB}) \rightarrow \text{W5.LSB} & \quad \text{(byte operation)}
\end{align*}
\]

Contents of working register W3 copied to working register W5.

This can either be a word or byte operation. The term ‘copy’ is used here instead of ‘move’ to emphasize that Wso is left unaffected by the operation.

The addressing mode used for both the source and destination operands is called **register direct**. The **mov** instruction supports other addressing modes which are not shown.
MOV W<sub>so</sub>, W<sub>do</sub>  Instruction Execution

(a) Execute: mov W<sub>2</sub>,W<sub>1</sub>  (word mode operation)

| W<sub>0</sub> | 0x1AF3 |
| W<sub>1</sub> | 0x8B1A |
| W<sub>2</sub> | 0x64DE |
| W<sub>3</sub> | 0xFB90 |

Before

| W<sub>0</sub> | 0x1AF3 |
| W<sub>1</sub> | 0x64DE |
| W<sub>2</sub> | 0xFB90 |

Modified

After

(b) Execute: mov.b W<sub>2</sub>,W<sub>1</sub>  (byte mode operation)

| W<sub>0</sub> | 0x1AF3 |
| W<sub>1</sub> | 0x8B1A |
| W<sub>2</sub> | 0x64DE |
| W<sub>3</sub> | 0xFB90 |

Before

| W<sub>0</sub> | 0x1AF3 |
| W<sub>1</sub> | 0x8BDE |
| W<sub>2</sub> | 0x64DE |
| W<sub>3</sub> | 0xFB90 |

Modified

After
**MOV Wso, Wdo** Instruction Format

(a) \(\text{mov}\{.b\} Wso,Wdo\)

\[(Wso) \rightarrow Wdo \text{ (reg. direct)} \]

(indirect addressing modes not shown)

\[
\begin{array}{c}
\text{BBBB BBBBB BBBBB BBBBB BBBBB BBBBB}
\text{2222 1111 1111 1100 0000 0000}
\text{3210 9876 5432 1098 7654 3210}
\end{array}
\]

\[0111 1www wBhh hddd dggg sssss\]

\(www = \text{base register (Wb) for indirect offset}\)

\(B = 0 \text{ for word, 1 for byte}\)

\(hhh = Wdo \text{ addressing mode (Register direct = 000)}\)

\(ddd = Wdo \text{ register number (0 to 15)}\)

\(ggg = Wso \text{ addressing mode (Register direct = 000)}\)

\(ssss = Wso \text{ register number (0 to 15)}\)

(b) Assembly:

\(\text{mov W3,W5}\)

Machine Code:

\[0x780283\]

Machine Code = 0111 1000 000 [010 000] [001] = 0x780283

\(B = \text{word mode} = 0\)

\(ssss = 0011 \text{ (register number is 3)}\)

\(ddd = 0101 \text{ (register number is 5)}\)

\(ggg, hhh, www \text{ fields are all 0 because indirect addressing is not used}\)

(c) \(\text{mov.b W3,W5}\)

\[0x784283\]

Byte mode, only difference is \(B = 1\)

Copyright Delmar Cengage Learning 2008. All Rights Reserved.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.
**MOV Wns, f Instruction**

“Copy contents of Wns register to data memory location f.”

General form:

\[
\text{MOV } Wns, f \quad \text{(Wns) → f}
\]

\(f\) is a memory location in data memory, Wns is one of the 16 working registers W0 through W15 (‘s’ indicates Wns is an operand source register for the operation)

\[
\text{MOV } W3, 0x1000 \quad \text{(W3) → 0x1000}
\]

Contents of register W3 copied to data memory location 0x1000. This instruction form only supports WORD operations.

The source operand uses *register direct* addressing, while the destination operand uses *file register* addressing.

*File registers* is how Microchip refers to data memory.
\textit{MOV Wns, f} Instruction Execution

Execute: \texttt{mov W3, 0x1002}

\begin{tabular}{|c|c|}
\hline
Location & Contents \\
\hline
0x1000 & 0x8B1A \\
0x1002 & 0x25AC \\
0x1004 & 0xFB90 \\
0x1006 & 0x9ED7 \\
\hline
\end{tabular}

\texttt{W3 = 0x64DE} (unaffected)

\begin{tabular}{|c|c|}
\hline
Location & Contents \\
\hline
0x1000 & 0x8B1A \\
0x1002 & 0x64DE \\
0x1004 & 0xFB90 \\
0x1006 & 0x9ED7 \\
\hline
\end{tabular}

\texttt{copied}

Before

\texttt{W3 = 0x64DE}

After

\texttt{modified}
MOV \textit{Wns}, f \quad \text{Instruction Format}

(a)  
\begin{align*}
\text{mov } Wns, f \\
\text{2222} & \quad \text{BBBB} \\
\text{1111} & \quad \text{BBBB} \\
\text{1111} & \quad \text{BBBB} \\
\text{1100} & \quad \text{BBBB} \\
\text{0000} & \quad \text{BBBB} \\
\text{0000} & \quad \text{BBBB} \\
\text{3210} & \quad \text{9876} \\
\text{5432} & \quad \text{1098} \\
\text{7654} & \quad \text{3210} \\
\end{align*}

(Wns) \rightarrow f

\begin{align*}
\text{1000} & \quad \text{ffff} \\
\text{ffff} & \quad \text{ffff} \\
\text{ffff} & \quad \text{ffff} \\
\text{ssss} & \quad \text{ssss} \\
\end{align*}

- f ... f = \text{upper 15 bits of 16-bit address (lower bit assumed = 0)}
- ssss = Wns \text{ register number (0 to 15)}

(b) \text{Assembly:}
mov W3,0x1002

\begin{align*}
\text{Machine Code:} \\
\text{0x888013} \\
\end{align*}

\begin{align*}
\text{Machine Code} & = 1000 \quad \text{100010000000001} \quad \text{0011} \\
\text{0x888013} & = 0001 \quad \text{00000000001} \quad \text{0} \\
\text{f} \ldots \text{f} & = \underline{0001} \quad \text{0000} \quad \text{0000} \quad \text{0010} \\
\text{(upper 15-bits of 0x1002)} & \quad \text{ssss} = \text{0011} \quad \text{(register number is 3)}
\end{align*}
**MOV $f$, $Wnd$ Instruction**

“Copy contents of data memory location $f$ to register $Wnd$”. General form:

\[
\text{MOV } f, Wnd \quad (f) \rightarrow Wnd
\]

$f$ is a memory location in data memory, $Wnd$ is one of the 16 working registers W0 through W15 (‘d’ indicates $Wnd$ is an operand destination register for the operation).

\[
\text{MOV } 0x1000, W3 \quad (0x1000) \rightarrow W3
\]

Contents of data memory location 0x1000 copied to W3.

() is read as “Contents of”.

This instruction form only supports a word operation.
**MOV f, Wnd** Instruction Execution

Execute: `mov 0x1002, W3`

Before:

<table>
<thead>
<tr>
<th>Location</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0x8B1A</td>
</tr>
<tr>
<td>0x1002</td>
<td>0x25AC</td>
</tr>
<tr>
<td>0x1004</td>
<td>0xFB90</td>
</tr>
<tr>
<td>0x1006</td>
<td>0x9ED7</td>
</tr>
</tbody>
</table>

W3 = 0x64DE

After:

<table>
<thead>
<tr>
<th>Location</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0x8B1A</td>
</tr>
<tr>
<td>0x1002</td>
<td>0x25AC</td>
</tr>
<tr>
<td>0x1004</td>
<td>0xFB90</td>
</tr>
<tr>
<td>0x1006</td>
<td>0x9ED7</td>
</tr>
</tbody>
</table>

W3 = 0x25AC (modified)

Copyright Delmar Cengage Learning 2008. All Rights Reserved.

From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.
A Note on Instruction Formats

- The instruction formats (machine code) of some instructions will be presented for informational purposes
  - However, studying the machine code formats of the instructions is not a priority; understanding instruction functionality will be emphasized.
  - All instruction formats can be found in the dsPIC30F/dsPIC33F Programmers Reference manual from Microchip.
  - The PIC24 family is a subset of the dsPIC30F/dsPIC33FF instruction set – the PIC24 family does not implement the DSP instructions.
**MOV {.B} WREG, f** Instruction

“Copy content of WREG (default working register) to data memory location \( f \)”. General form:

\[
\text{MOV} \;\{.B\} \; WREG, f \quad \text{(WREG)} \rightarrow f
\]

This instruction provides upward compatibility with earlier PIC \( \mu \)C. WREG is register W0, and \( f \) is a location within the first 8192 bytes of data memory (*near* data memory)

\[
\text{MOV} \quad WREG, 0x1000 \quad \text{(W0)} \rightarrow 0x1000
\]

Contents of register W0 copied to data memory location 0x1000.

Can be used for either WORD or BYTE operations:

MOV WREG, 0x1000 word operation

MOV.B WREG, 0x1001 lower 8-bits of W0 copied to 0x1001

Word copy must be to even (word-aligned) location.

Note: The previously covered **MOV Wns, f** instruction cannot be used for byte operations!
MOV.B WREG, f  Instruction Execution

A byte copy operation is shown.
\[ \text{MOV\{.B\} WREG, \, f} \]

**Instruction Format**

\[
\begin{array}{cccccccc}
\text{BBBB} & \text{BBBB} & \text{BBBB} & \text{BBBB} & \text{BBBB} \\
\text{2222} & \text{1111} & \text{1111} & \text{1100} & \text{0000} & \text{0000} \\
\text{3210} & \text{9876} & \text{5432} & \text{1098} & \text{7654} & \text{3210} \\
\text{1011} & \text{0111} & \text{1B1f} & \text{ffff} & \text{ffff} & \text{ffff} \\
\end{array}
\]

*\( f \) ... *\( f \) = 13-bit address (first 8192 bytes of data memory)

*\( b \) = 0 for word, 1 for byte

**Assembly:**

```
mov WREG, 0x1000
mov.b WREG, 0x1000
mov.b WREG, 0x1001
```

**Machine Code:**

```
0xB7B000 \quad (b \text{ bit } = 0 \text{ since word operation})
0xB7F000 \quad (b \text{ bit } = 1 \text{ since byte operation})
0xB7F001 \quad (\text{bytes can be written to odd addresses})
```
**MOV{.B} \ f \{,WREG\} Instruction**

“Copy contents of data memory location \( f \) to WREG (default working register). General form:

\[
\text{MOV}\{.B\} \ f, \text{WREG} \quad (f) \rightarrow \text{WREG}
\]

\[
\text{MOV}\{.B\} \ f \quad (f) \rightarrow f
\]

This instruction provides upward compatibility with earlier PIC \( \mu \)C. WREG is register W0, and \( f \) is a location within the first 8192 bytes of data memory (near data memory)

Can be used for either WORD or BYTE operations:

MOV 0x1000, WREG        \hspace{1cm} \text{word operation}

MOV.B 0x1001, WREG     \hspace{1cm} \text{only lower 8-bits of W0 are affected.}

MOV 0x1000              \hspace{1cm} \text{Copies contents of 0x1000 back to itself, will see usefulness of this later}

Word copy must be from even (word-aligned) data memory location.

Note: The \( \text{MOV} f, Wnd \) instruction cannot be used for byte operations!
**MOV{.B} f {WREG} Format**

**Assembly:**

- mov 0x1000, WREG
- mov.b 0x1000

**Machine Code:**

- 0xBF9000 (B bit = 0 since word operation, D bit = 0 since WREG destination)
- 0xBFF000 (B bit = 1 since byte operation, D bit = 1 since f destination)

---

5.2

*From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.*
\textbf{MOV.\{B\} f, WREG} Instruction Execution

Execute: \texttt{mov.b 0x1001,WREG}

\[
W0 = 0x64DE
\]

\[
W0 = 0x648B\text{ (modified)}
\]

\begin{tabular}{|c|c|}
\hline
Location & Contents \\
\hline
0x1000 & 0x8B1A \\
0x1002 & 0x25AC \\
0x1004 & 0xFB90 \\
0x1006 & 0x9ED7 \\
\hline
\end{tabular}

\begin{tabular}{|c|c|}
\hline
Location & Contents \\
\hline
0x1000 & 0x8B1A \\
0x1002 & 0x25AC \\
0x1004 & 0xFB90 \\
0x1006 & 0x9ED7 \\
\hline
\end{tabular}

Before

\textbf{copied}

After

\textbf{unaffected}
Move a literal into a Working Register

Moves a *literal* into a working register. The ‘#’ indicates the numeric value is a literal, and NOT a memory address.

General form:

- **MOV #lit16, Wnd**  
  \[ \text{lit16} \rightarrow \text{Wnd} \] (word operation)

- **MOV.B #lit8, Wnd**  
  \[ \text{lit8} \rightarrow \text{Wnd.lsb} \] (byte operation)

The source operand in these examples use the *immediate* addressing mode.

Examples:

- **MOV #0x1000, W2**  
  \[ 0x1000 \rightarrow W2 \]

- **MOV.B #0xAB, W3**  
  \[ 0xAB \rightarrow W3.lsb \]
More on Literals

Observe that the following two instructions are very different!

MOV #0x1000, W2 after execution, W2=0x1000

MOV 0x1000,W2 after execution, W2 = (0x1000), the contents of memory location 0x1000
MOV Literal Execution

(a) Execute: mov #0x1000, W2  (immediate addressing)

```
0x1000: 0xB1A
W2:      0xD038
```

Before

```
0x1000: 0xB1A
W2:      0xD038
```

After

```
0x1000: 0xB1A
W2:      0x1000
```

Modified

(b) Execute: mov 0x1000, W2  (file register addressing)

```
0x1000: 0xB1A
W2:      0xD038
```

Before

```
0x1000: 0xB1A
W2:      0xD038
```

After

```
0x1000: 0xB1A
W2:      0xB1A
```

Modified

(c) Execute: mov.b #0xF2, W2

```
0x1000: 0xB1A
W2:      0xD038
```

Before

```
0x1000: 0xB1A
W2:      0xD038
```

After

```
0x1000: 0xB1A
W2:      0xD0F2
```

Modified
MOV Literal Instruction Formats

\[
\begin{align*}
\text{mov} & \ #\text{lit16}, \ W_n \quad \#\text{lit16} \to W_n \\
\text{mov.b} & \ #\text{lit8}, \ W_n \quad \#\text{lit8} \to W_n
\end{align*}
\]

\#\text{lit16}: 16-bit literal

\#\text{lit8}: 8-bit literal

Assembly: \hspace{1cm} \text{Machine Code:}

\[
\begin{align*}
\text{mov} \quad & \ #0x1000, \ W_2 \quad 0x21002 \\
\text{mov.b} \quad & \ #0xF2, \ W_7 \quad 0xB3CF27
\end{align*}
\]

Observe that the literal is encoded directly in the instruction machine code.
Indirect Addressing

Mov with indirect Addressing:

\[ \text{mov}\.b\ \ [\text{Wso}],\ [\text{Wdo}] \quad ((\text{Wso})) \rightarrow (\text{Wdo}) \]

[] (brackets) indicate indirect addressing.
Source Effective Address (EAs) is the content of Wso, or (Wso).
Destination Effective Address (EAd) is the content of Wdo, or (Wdo).

The MOV instruction copies the content of the Source Effective Address to the Destination Effect Address, or:

\[ (\text{EAs}) \rightarrow \text{EAd} \]

which is:

\[ ((\text{Wso})) \rightarrow (\text{Wdo}) \]
Indirect Addressing

MOV Example

(a) Execute: `mov W0, W1` source, destination use register direct

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>0x1000</td>
</tr>
<tr>
<td>W1</td>
<td>0x1002</td>
</tr>
<tr>
<td>0x1000</td>
<td>0x8BFA</td>
</tr>
<tr>
<td>0x1002</td>
<td>0x25AC</td>
</tr>
</tbody>
</table>

W0 modified to 0x1000

(b) Execute: `mov [W0], [W1]` source, destination use register indirect

Source Effective Address = (W0) = 0x1000
Destination Effective Address = (W1) = 0x1002
Operation is (0x1000) → 0x1002

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>0x1000</td>
</tr>
<tr>
<td>W1</td>
<td>0x1002</td>
</tr>
<tr>
<td>0x1000</td>
<td>0x8BFA</td>
</tr>
<tr>
<td>0x1002</td>
<td>0x25AC</td>
</tr>
</tbody>
</table>

W0 modified to 0x1000

(c) Execute: `mov W0, [W1]` source uses register direct, destination uses register indirect

Source Effective Address = W0
Destination Effective Address = (W1) = 0x1002
Operation is (W0) → 0x1002

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0</td>
<td>0x1000</td>
</tr>
<tr>
<td>W1</td>
<td>0x1002</td>
</tr>
<tr>
<td>0x1000</td>
<td>0x8BFA</td>
</tr>
<tr>
<td>0x1002</td>
<td>0x25AC</td>
</tr>
</tbody>
</table>

W0 modified to 0x1000

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

The instruction:

```
  mov  [W0], [W1]
```

Allows us to do a memory-memory copy with one instruction!

The following is illegal:

```
  mov  0x1000,  0x1002
```

Instead, would have to do:

```
  mov 0x1000, W0
  mov  W0, 0x1002
```
Indirect Addressing Coverage

• There are six forms of indirect addressing
• The need for indirect addressing makes the most sense when covered in the context of C pointers
  – This is done in Chapter 5
• At this time, you will only need to understand the simplest form of indirect addressing, which is *register indirect* as shown on the previous two slides.
• Most instructions that support register direct for an operand, also support indirect addressing as well for the same operand
  – However, must check PIC24 datasheet and book to confirm.
ADD{.B} Wb, Ws, Wd Instruction

Three operand addition, register-to-register form:

ADD{.B} Wb, Ws, Wd \( (Wb) + (Ws) \to Wd \)

Wb, Ws, Wd are any of the 16 working registers W0-W15

ADD W0, W1, W2 \( (W0) + (W1) \to W2 \)
ADD W2, W2, W2 \( W2 = W2 + W2 = W2*2 \)

ADD.B W0, W1, W2 Lower 8 bits of W0, W1 are added and placed in the lower 8 bits of W2
### $ADD\{.B\} Wb, Ws, Wd$ Execution

(a) Execute: $\text{add} \ W0,W1,W2$

<table>
<thead>
<tr>
<th>Before</th>
<th></th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W0$</td>
<td>0x1AF3</td>
<td>$W0$ 0x1AF3</td>
</tr>
<tr>
<td>$W1$</td>
<td>0x8B1A</td>
<td>$W1$ 0x8B1A</td>
</tr>
<tr>
<td>$W2$</td>
<td>0x64DE</td>
<td>$W2$ 0x6A0D</td>
</tr>
<tr>
<td>$W3$</td>
<td>0xFB90</td>
<td>$W3$ 0xFB90</td>
</tr>
</tbody>
</table>

(b) Execute: $\text{add.b} \ W0,W1,W2$

<table>
<thead>
<tr>
<th>Before</th>
<th></th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W0$</td>
<td>0x1AF3</td>
<td>$W0$ 0x1A23</td>
</tr>
<tr>
<td>$W1$</td>
<td>0x8B1A</td>
<td>$W1$ 0x8B1A</td>
</tr>
<tr>
<td>$W2$</td>
<td>0x64DE</td>
<td>$W2$ 0x64DE</td>
</tr>
<tr>
<td>$W3$</td>
<td>0xFB90</td>
<td>$W3$ 0xFB90</td>
</tr>
</tbody>
</table>

(c) Execute: $\text{add} \ W2,W2,W2$

<table>
<thead>
<tr>
<th>Before</th>
<th></th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W0$</td>
<td>0x1AF3</td>
<td>$W0$ 0x1AF3</td>
</tr>
<tr>
<td>$W1$</td>
<td>0x8B1A</td>
<td>$W1$ 0x8B1A</td>
</tr>
<tr>
<td>$W2$</td>
<td>0x64DE</td>
<td>$W2$ 0xC9BC</td>
</tr>
<tr>
<td>$W3$</td>
<td>0xFB90</td>
<td>$W3$ 0xFB90</td>
</tr>
</tbody>
</table>

Result limited to 8-bits!
**SUB{.B} Wb, Ws, Wd** Instruction

Three operand subtraction, register-to-register form:

\[
\text{SUB}\{.B\} \quad Wb, Ws, Wd \quad (Wb) - (Ws) \rightarrow Wd
\]

Wb, Ws, Wd are any of the 16 working registers W0-W15.

Be careful:

while \( \text{ADD} \ Wx, Wy, Wz \) gives the same result as \( \text{ADD} \ Wy, Wx, Wz \)

The same is not true for

\( \text{SUB} \ Wx, Wy, Wz \) versus \( \text{SUB} \ Wy, Wx, Wz \)

\[
\begin{align*}
\text{SUB} & \quad W0, W1, W2 \quad (W0) - (W1) \rightarrow W2 \\
\text{SUB} & \quad W1, W0, W2 \quad (W1) - (W0) \rightarrow W2 \\
\text{SUB}.B & \quad W0, W1, W2 \quad \text{Lower 8 bits of } W0, W1 \text{ are subtracted and placed in the lower 8-bits of } W2
\end{align*}
\]
SUB\{B\} Wb, Ws, Wd Execution

(a) Execute: sub W0,W1,W2

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 0x1AF3</td>
<td>W0 0x1AF3</td>
</tr>
<tr>
<td>W1 0x8B1A</td>
<td>W1 0x8B1A</td>
</tr>
<tr>
<td>W2 0x64DE</td>
<td>W2 0x8FD9</td>
</tr>
<tr>
<td>W3 0xFB90</td>
<td>W3 0xFB90</td>
</tr>
</tbody>
</table>

(b) Execute: sub W1,W0,W2

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 0x1AF3</td>
<td>W0 0x1AF3</td>
</tr>
<tr>
<td>W1 0x8B1A</td>
<td>W1 0x8B1A</td>
</tr>
<tr>
<td>W2 0x64DE</td>
<td>W2 0x7027</td>
</tr>
<tr>
<td>W3 0xFB90</td>
<td>W3 0xFB90</td>
</tr>
</tbody>
</table>

(c) Execute: sub.b W0,W1,W2

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>W0 0x1AF3</td>
<td>W0 0x1AF3</td>
</tr>
<tr>
<td>W1 0x8B1A</td>
<td>W1 0x8B1A</td>
</tr>
<tr>
<td>W2 0x64DE</td>
<td>W2 0x64F9</td>
</tr>
<tr>
<td>W3 0xFB90</td>
<td>W3 0xFB90</td>
</tr>
</tbody>
</table>
Subtraction/Addition with Literals

Three operand addition/subtraction with literals:

\[
\text{ADD}\{.B\} \quad Wb, \ #\text{lit}5, \ Wd \quad \quad (Wb) – \ #\text{lit}5 \rightarrow Wd
\]

\[
\text{SUB}\{.B\} \quad Wb, \ #\text{lit}5, \ Wd \quad \quad (Wb) – \ #\text{lit}5 \rightarrow Wd
\]

#lit5 is a 5-bit unsigned literal; the range 0-31. Provides a convenient method of adding/subtracting a small constant using a single instruction

Examples

\[
\text{ADD} \quad W0, \ #4, \ W2 \quad \quad (W0) + 4 \rightarrow W2
\]

\[
\text{SUB.B} \quad W1,\#8, \ W3 \quad \quad (W1) – 8 \rightarrow W3
\]

\[
\text{ADD} \quad W0, \ #60, \ W1 \quad \text{illegal, 60 is greater than 31!}
\]
**ADD{.B} f {,WREG}** Instruction

Two operand addition form:

\[
\text{ADD{.B}} \ f \quad (f) + (\text{WREG}) \rightarrow f
\]

\[
\text{ADD{.B}} \ f, \text{WREG} \quad (f) + (\text{WREG}) \rightarrow \text{WREG}
\]

WREG is W0, f is limited to first 8192 bytes of memory.

One of the operands, either f or WREG is always destroyed!

\[
\text{ADD} \ 0x1000 \quad (0x1000) + (\text{WREG}) \rightarrow 0x1000
\]

\[
\text{ADD} \ 0x1000, \text{WREG} \quad (0x1000) + (\text{WREG}) \rightarrow \text{WREG}
\]

\[
\text{ADD.B} \ 0x1001, \text{WREG} \quad (0x1001) + (\text{WREG}.\text{lsb}) \rightarrow \text{WREG}.\text{lsb}
\]
Assembly Language Efficiency

The effects of the following instruction:

ADD 0x1000 (0x1000) + (WREG) → 0x1000

Can also be accomplished by:

MOV 0x1000 , W1 (0x1000) → W1
ADD W0, W1, W1 (W0) + (W1) → W1
MOV W1, 0x1000 (W1) → 0x1000

This takes three instructions and an extra register. However, in this class we are only concerned with the correctness of your assembly language, and not the efficiency. Use whatever approach you best understand!!!!!
**ADD{.B} f {,WREG} Execution**

(a) Execute: ADD 0x1000

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Address</th>
<th>Value</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0x8BFA</td>
<td>0x1000</td>
<td>0x5C32</td>
<td>0x1002</td>
<td>0x25AC</td>
</tr>
<tr>
<td>0x1002</td>
<td>0x25AC</td>
<td>+0xD038</td>
<td>0x1002</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W0</td>
<td>0xD038</td>
<td>0x5C32</td>
<td>W0</td>
<td>0xD038</td>
<td></td>
</tr>
</tbody>
</table>

Before

Modified

(b) Execute: ADD 0x1000, WREG

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Address</th>
<th>Value</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0x8BFA</td>
<td>0x1000</td>
<td>0x8BFA</td>
<td>0x1002</td>
<td>0x25AC</td>
</tr>
<tr>
<td>0x1002</td>
<td>0x25AC</td>
<td>+0xD038</td>
<td>0x1002</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W0</td>
<td>0xD038</td>
<td>0x5C32</td>
<td>W0</td>
<td>0x5C32</td>
<td></td>
</tr>
</tbody>
</table>

Before

Modified

(c) Execute: ADD.B 0x1001

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Address</th>
<th>Value</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1000</td>
<td>0x8BFA</td>
<td>0x1000</td>
<td>0xC3FA</td>
<td>0x1002</td>
<td>0x25AC</td>
</tr>
<tr>
<td>0x1002</td>
<td>0x25AC</td>
<td>+0x38</td>
<td>0x1002</td>
<td></td>
<td></td>
</tr>
<tr>
<td>W0</td>
<td>0xD038</td>
<td>0xC3</td>
<td>W0</td>
<td>0xD038</td>
<td></td>
</tr>
</tbody>
</table>

Before

Modified

V 0.2
**SUB{.B} f {,WREG}** Instruction

Two operand subtraction form:

\[ \text{SUB\{.B\} } f \quad (f) - (\text{WREG}) \rightarrow f \]

\[ \text{SUB\{.B\} } f, \text{WREG} \quad (f) - (\text{WREG}) \rightarrow \text{WREG} \]

WREG is W0, f is limited to first 8192 bytes of memory.

One of the operands, either f or WREG is always destroyed!

\[ \text{SUB \ 0x1000} \quad (0x1000) - (\text{WREG}) \rightarrow 0x1000 \]

\[ \text{SUB \ 0x1000, WREG} \quad (0x1000) - (\text{WREG}) \rightarrow \text{WREG} \]

\[ \text{SUB.B \ 0x1001, WREG} \quad (0x1001) - (\text{WREG}.\text{lsb}) \rightarrow \text{WREG}.\text{lsb} \]
Increment

Increment operation, register-to-register form:

\[
\text{INC}.B \ W_s, W_d \quad (W_s) + 1 \rightarrow W_d
\]

Increment operation, memory to memory/WREG form:

\[
\text{INC}.B \ f \quad (f) + 1 \rightarrow f
\]

\[
\text{INC}.B \ f, \ \text{WREG} \quad (f) + 1 \rightarrow \text{WREG}
\]

(f must be in first 8192 locations of data memory)

Examples:

\[
\text{INC} \ W2, W4 \quad (W2) + 1 \rightarrow W4
\]

\[
\text{INC}.B \ W3, W3 \quad (W3.\text{lsb}) + 1 \rightarrow W3.\text{lsb}
\]

\[
\text{INC} \ 0x1000 \quad (0x1000) + 1 \rightarrow 0x1000
\]

\[
\text{INC}.B \ 0x1001, \text{WREG} \quad (0x1001)+1 \rightarrow \text{WREG.\text{lsb}}
\]
Decrement

Decrement operation, register-to-register form:

\[ \text{DEC}.B \ W_s, W_d \quad (W_s) - 1 \rightarrow W_d \]

Increment operation, memory to memory/WREG form:

\[ \text{DEC}.B \ f \quad (f) - 1 \rightarrow f \]
\[ \text{DEC}.B \ f, \ WREG \quad (f) - 1 \rightarrow \text{WREG} \]

(f must be in first 8192 locations of data memory)

Examples:

\[ \text{DEC} \ W2, W4 \quad (W2) - 1 \rightarrow W4 \]
\[ \text{DEC}.B \ W3, W3 \quad (W3.lsb) - 1 \rightarrow W3.lsb \]
\[ \text{DEC} \ 0x1000 \quad (0x1000) - 1 \rightarrow 0x1000 \]
\[ \text{DEC}.B \ 0x1001, \text{WREG} \quad (0x1001) - 1 \rightarrow \text{WREG}.lsb \]
How is the instruction register loaded?

The Program counter contains the program memory address of the instruction that will be loaded into the instruction register. After reset, the first instruction fetched from location 0x000000 in program memory, i.e., the program counter is reset to 0x000000.
Program Memory Organization

An instruction is 24 bits (3 bytes). Program memory should be viewed as words (16-bit addressable), with the upper byte of the upper word of an instruction always reading as ‘0’.

Instructions must start on even-word boundaries. Instructions are addressed by the Program counter (PC).

PC is 23-bits wide, but instructions start on even word boundaries, so the PC can address 4M instructions (M = 2^20).

Adapted from Fig 3-2, DS70289A, Microchip, Inc.

Figure adapted with permission of the copyright owner, Microchip Technology, Incorporated. All rights reserved.
Goto location (*goto*)

How can the program counter be changed?

![Goto Instruction](image)

- **goto** *Expr* *lit23* → PC

*Expr* is a label or expression that is resolved by the linker to a 23-bit program memory address known as the *target address* (this must be an even address).

The GOTO instruction requires two instruction words:

**Assembly:**

```
goto 0x [000800]
```

**Machine Code:**

```
0x04 0800 0x0000 00
```

The first word is 0x04 0800, and the second word is 0x0000 00.

- **First word**
- **Second word**

A GOTO instruction is an unconditional jump.
Valid addressing modes.

What are valid addressing modes for instructions?

The definitive answer can be found in Table 19-2 of the PIC24H32GP202 datasheet.

<table>
<thead>
<tr>
<th>Base Instr #</th>
<th>Assembly Mnemonic</th>
<th>Assembly Syntax</th>
<th>Description</th>
<th># of Words</th>
<th># of Cycles</th>
<th>Status Flags Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>MOV</td>
<td>f, Wn</td>
<td>Move f to Wn</td>
<td>1</td>
<td>1</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>MOV</td>
<td>f</td>
<td>Move f to f</td>
<td>1</td>
<td>1</td>
<td>N.Z</td>
</tr>
<tr>
<td></td>
<td>MOV</td>
<td>f, WREG</td>
<td>Move f to WREG</td>
<td>1</td>
<td>1</td>
<td>N.Z</td>
</tr>
<tr>
<td></td>
<td>MOV</td>
<td>#lit16,Wn</td>
<td>Move 16-bit literal to Wn</td>
<td>1</td>
<td>1</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>MOV.b</td>
<td>#lit8,Wn</td>
<td>Move 8-bit literal to Wn</td>
<td>1</td>
<td>1</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>MOV</td>
<td>Wn,f</td>
<td>Move Wn to f</td>
<td>1</td>
<td>1</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>MOV</td>
<td>Wso,Wdo</td>
<td>Move Ws to Wd</td>
<td>1</td>
<td>1</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>MOV</td>
<td>WREG,f</td>
<td>Move WREG to f</td>
<td>1</td>
<td>1</td>
<td>N.Z</td>
</tr>
<tr>
<td></td>
<td>MOV.D</td>
<td>Wns,Wd</td>
<td>Move Double from W(ns):W(ns+1) to Wd</td>
<td>1</td>
<td>2</td>
<td>None</td>
</tr>
<tr>
<td></td>
<td>MOV.D</td>
<td>Ws,Wnd</td>
<td>Move Double from Ws to W(nd+1):W(nd)</td>
<td>1</td>
<td>2</td>
<td>None</td>
</tr>
</tbody>
</table>
What does ‘Wso’, ‘Wsd’, ‘Wn’ etc. mean?

MOV Wso, Wdo

Table 19-1: Symbols used in opcode descriptions (partial list)

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wnd</td>
<td>One of 16 destination working registers ∈ {W0..W15}</td>
</tr>
<tr>
<td>Wns</td>
<td>One of 16 source working registers ∈ {W0..W15}</td>
</tr>
<tr>
<td>WREG</td>
<td>W0 (working register used in file register instructions)</td>
</tr>
<tr>
<td>Ws</td>
<td>Source W register ∈ {Ws, [Ws], [Ws++], [Ws--], [++Ws], [--Ws] }</td>
</tr>
<tr>
<td>Wso</td>
<td>Source W register ∈ {Wns, [Wns], [Wns++], [Wns--], [++Wns], [--Wns], [Wns+Wb]}</td>
</tr>
<tr>
<td>Wd</td>
<td>Destination W register ∈ {Wd, [Wd], [Wd++], [Wd--], [++Wd], [--Wd]}</td>
</tr>
<tr>
<td>Wdo</td>
<td>Destination W register ∈ {Wnd, [Wnd], [Wnd++], [Wnd--], [++Wnd], [--Wnd]}</td>
</tr>
<tr>
<td>Wn</td>
<td>One of 16 working registers ∈ {W0..W15}</td>
</tr>
<tr>
<td>Wb</td>
<td>Base W register ∈ {W0..W15}</td>
</tr>
</tbody>
</table>
ADD forms

ADD Wb, Ws, Wd

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ws</td>
<td>Source W register ∈ {Ws, [Ws], [Ws++], [Ws--], [++Ws], [--Ws] }</td>
</tr>
<tr>
<td>Wd</td>
<td>Destination W register ∈ {Wd, [Wd], [Wd++], [Wd--], [++Wd], [--Wd] }</td>
</tr>
<tr>
<td>Wb</td>
<td>Base W register ∈ {W0..W15}</td>
</tr>
</tbody>
</table>

Legal:

ADD W0, W1, W2
ADD W0, [W1], [W4]

Illegal:

ADD [W0], W1, W2 ;first operand illegal!
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

• **Converting the program from 8 to 16 bits** at
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

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.
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.
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).
.text

.reset: mov #__SP_init, W15
      mov #__SPLIM_init, W0
      mov W0, SPLIM

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

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

avalue = 100

The equal sign is an assembler directive that equates a label to a value.
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.
General MPLAB IDE Comments

• See Experiment #2 for detailed instructions on installing the MPLAB IDE on your PC and assembling/simulating programs.

• The assembly language file must have the .s extension and must be a TEXT file
  – Microsoft .doc files are NOT text files
  – The MPLAB IDE has a built-in text editor. If you use an external text editor, use one that displays line numbers (e.g. don’t use notepad – does not display line numbers)

• You should use your portable PC for experiments 1-5 in this class; all of the required software is freely available.
An Alternate Solution

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

mov #100,W1    ; W1 (i) = 100
inc.b W1,W1    ; W1 (i) = W1 (i) + 1
mov.b W1,W2    ; W2 (j) = W1 (i)
dec.b W2,W2    ; W2 (j) = W2 (j) -1
add.b W1,W2,W3 ; W3 (k) = W1 (i) + W2 (j)
```

Previous approach took 9 instructions, this one took 11 instructions. Use whatever approach that you best understand.
Clock Cycles vs. Instruction Cycles

The clock signal used by a PIC24 µC to control instruction execution can be generated by an off-chip oscillator or crystal/capacitor network, or by using the internal RC oscillator within the PIC24 µC.

For the PIC24H family, the maximum clock frequency is 80 MHz.

An instruction cycle (FCY) is two clock (FOSC) cycles. Important!!!!!!!

A PIC24 instruction takes 1 or 2 instruction (FCY) cycles, depending on the instruction (see Table 19-2, PIC24HJ32GP202 data sheet). If an instruction causes the program counter to change (i.e, GOTO), that instruction takes 2 instruction cycles.

An add instruction takes 1 instruction cycle. How much time is this if the clock frequency (Fosc) is 80 MHz (1 MHz = 1.0e6 = 1,000,000 Hz)?

1/frequency = period, 1/80 MHz = 12.5 ns (1 ns = 1.0e-9 s)

1 Add instruction @ 80 MHz takes 2 clocks * 12.5 ns = 25 ns (or 0.025 us).

By comparison, an Intel Pentium add instruction @ 3 GHz takes 0.33 ns (330 ps). An Intel Pentium could emulate a PIC24HJ32GP202 faster than a PIC24HJ32GP202 can execute! But you can’t put a Pentium in a toaster, or buy one from Digi-key for $5.00.
How long does mptst_byte.s take to execute?
Beginning at the __reset label, and ignoring the goto at the end, takes 12 instruction cycles, which is 24 clock cycles.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov __SP_init, W15</td>
<td>1</td>
</tr>
<tr>
<td>mov __SPLIM_init,W0</td>
<td>1</td>
</tr>
<tr>
<td>mov W0,SPLIM</td>
<td>1</td>
</tr>
<tr>
<td>mov.b #avalue, W0</td>
<td>1</td>
</tr>
<tr>
<td>mov.b WREG,i</td>
<td>1</td>
</tr>
<tr>
<td>inc.b i</td>
<td>1</td>
</tr>
<tr>
<td>mov.b i,WREG</td>
<td>1</td>
</tr>
<tr>
<td>mov.b WREG,j</td>
<td>1</td>
</tr>
<tr>
<td>dec.b j</td>
<td>1</td>
</tr>
<tr>
<td>mov.b i,WREG</td>
<td>1</td>
</tr>
<tr>
<td>add.b j,WREG</td>
<td>1</td>
</tr>
<tr>
<td>mov.b WREG,k</td>
<td>1</td>
</tr>
</tbody>
</table>

v 0.2 Total 12
What if we used 16-bit variables instead of 8-bit variables?

C Program equivalent

```c
#define avalue 2047
uint16 i, j, k;

i = avalue;    // i = 2047
i = i + 1;     // i++, i = 2048
j = i;         // j is 2048
j = j - 1;     // j--, j is 2047
k = j + i;     // k = 4095
```

A uint16 variable is 16 bits (1 byte)
Reserve 2 bytes for each variable. Variables are now stored at 0x0800, 0x0802, 0x0804.

Instructions now perform WORD (16-bit) operations (the .b qualifier is removed).
An Alternate Solution (16-bit variables)

C Program equivalent

```c
#define avalue 2047
uint16 i,j,k;

i = avalue;    // i = 2047
i = i + 1;     // i++, i = 2048
j = i;        // j is 2048
j = j - 1;    // j--, j is 2047
k = j + i;    // k = 4095
```

; Assign variables to registers
; Move variables into registers.
; use register-to-register operations for computations;
; write variables back to memory
; assign i to W1, j to W2, k to W3

```assembly
mov #2047,W1 ; W1 (i) = 2047
inc W1,W1    ; W1 (i) = W1 (i) + 1
mov W1,W2    ; W2 (j) = W1 (i)
dec W2,W2    ; W2 (j) = W2 (j) -1
add W1,W2,W3 ; W3 (k) = W1 (i) + W2 (j)
```

Previous approach took 9 instructions, this one took 8 instructions. In this case, this approach is more efficient!
How long does mptst_word.s take to execute?

Ignoring the goto at the end, takes 12 instruction cycles, which is 24 clock cycles.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov #__SP_init, W15</td>
<td>1</td>
</tr>
<tr>
<td>mov #__SPLIM_init,W0</td>
<td>1</td>
</tr>
<tr>
<td>mov W0,SPLIM</td>
<td>1</td>
</tr>
<tr>
<td>mov #avalue, W0</td>
<td>1</td>
</tr>
<tr>
<td>mov WREG,i</td>
<td>1</td>
</tr>
<tr>
<td>inc i</td>
<td>1</td>
</tr>
<tr>
<td>mov i,WREG</td>
<td>1</td>
</tr>
<tr>
<td>mov WREG,j</td>
<td>1</td>
</tr>
<tr>
<td>dec j</td>
<td>1</td>
</tr>
<tr>
<td>mov i,WREG</td>
<td>1</td>
</tr>
<tr>
<td>add j,WREG</td>
<td>1</td>
</tr>
<tr>
<td>mov WREG,k</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>12</td>
</tr>
</tbody>
</table>
16-bit operations versus 8-bit

The 16-bit version of the mptst program requires the same number of instruction bytes and the same number of instruction cycles as the 8-bit version.

This is because the PIC24 family is a 16-bit microcontroller; its natural operation size is 16 bits, so 16-bit operations are handled as efficiently as 8-bits operations.

On an 8-bit processor, like the PIC18 family, the 16-bit version would take roughly double the number of instructions and clock cycles as the 8-bit version.

On the PIC24, a 32-bit version of the mptst program will take approximately twice the number of instructions and clock cycles as the 16-bit version. We will look at 32-bit operations later in the semester.
Review: Units

In this class, units are always used for physical quantity:

<table>
<thead>
<tr>
<th>Time</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>milliseconds (ms = 10^{-3} s)</td>
<td>kilohertz (kHz = 10^{3} Hz)</td>
</tr>
<tr>
<td>microseconds (μs = 10^{-6} s)</td>
<td>megahertz (MHz = 10^{6} Hz)</td>
</tr>
<tr>
<td>nanoseconds (ns = 10^{-9} s)</td>
<td>gigahertz (GHz = 10^{9} Hz)</td>
</tr>
</tbody>
</table>

When a time/frequency/voltage/current quantity is asked for, I will always ask for it in some units. Values for these quantities in datasheets are ALWAYS given in units.

For a frequency of 1.25 kHz, what is the period in μs?

period = 1/f = 1/(1.25 e3) = 8.0 e –4 seconds

Unit conversion = 8.0e-4 (s) * (1e6 μs)/1.0 (s) = 8.0e2 μs = 800 μs
PIC24H Family

- Microchip has an extensive line of PICmicro® microcontrollers, with the PIC24 family introduced in 2005.
- The PIC16 and PIC18 are older versions of the PICmicro® family, have been several previous generations.
- Do not assume that because something is done one way in the PIC24, that it is the most efficient method for accomplishing that action.
- The datasheet for the PIC24HJ32GP202 is found on the class web site.
Some PICMicros that we have used

<table>
<thead>
<tr>
<th>Features</th>
<th>16F87x (Fall 2003)</th>
<th>PIC18F242 (Summer 2004)</th>
<th>PIC24H (Summer 2008)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction width</td>
<td>14 bits</td>
<td>16 bits</td>
<td>24 bits</td>
</tr>
<tr>
<td>Program memory</td>
<td>8K instr.</td>
<td>8K instructions</td>
<td>~10K instructions</td>
</tr>
<tr>
<td>Data Memory</td>
<td>368 bytes</td>
<td>1536 bytes</td>
<td>2048 bytes</td>
</tr>
<tr>
<td>Clock speed</td>
<td>Max 20 MHz, 4 clks=1 instr</td>
<td>Max 40 MHz 4 clks=1 instr</td>
<td>Max 80 MHz 2 clks=1 instr</td>
</tr>
<tr>
<td>Architecture</td>
<td>Accumulator, 8-bit architecture</td>
<td>Accumulator, 8-bit architecture</td>
<td>General purpose register, 16-bit architecture</td>
</tr>
</tbody>
</table>

The PIC24H can execute about 6x faster than the PIC18F242 previously used in this class.
What do you need to know?

• Understand the PIC24 basic architecture (program and data memory organization)

• Understand the operation of mov, add, sub, inc, dec, goto instructions and their various addressing mode forms

• Be able to convert simple C instruction sequences to PIC24 assembly language
  – Be able to assemble/simulate a PIC24 \( \mu \)C assembly language program in the MPLAB IDE

• Understand the relationship between instruction cycles and machine cycles