COE 115

Lecture 3

Indirect Addressing

Mov with indirect Addressing:

mov{.b} [Wso], [Wdo] ((Wso)) → (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:

(EAs) → EAd

which is:

((Wso)) → (Wdo)

Indirect Addressing

MOV Example

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

	Before	After	
W0	0x1000	0x1000	
W1	0x1002	0x1000	Modified
0x1000	0x8BFA	0x8BFA	
0x1002	0x25AC	0x25AC	

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

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

	Before	After	
W0	0x1000	0x1000	
W1	0x1002	0x1002	
0x1000	0x8BFA	0x8BFA	
0x1002	0x25AC	0x8BFA	Modified

(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

	Before	After	
W0	0x1000	0x1000	
W1	0x1002	0x1002	
0x1000	0x8BFA	0x8BFA	
0x1002	0x25AC	0x1000	Modified

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**
- ***register indirect* the simplest form of indirect addressing, which is as shown on the previous 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) → Wd

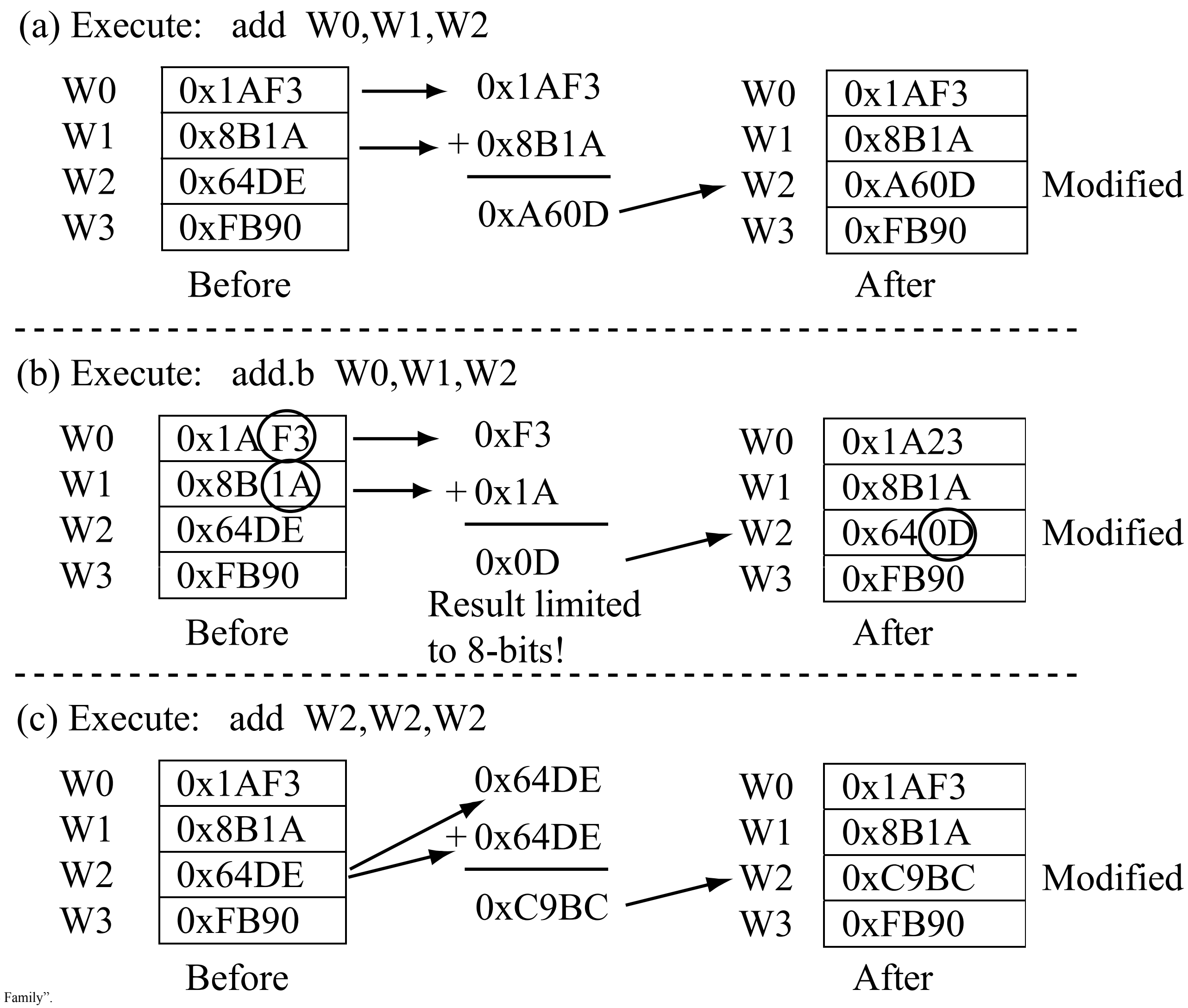
Wb, Ws, Wd are any of the 16 working registers W0-W15

ADD W0, W1, W2 (W0) + (W1) → 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 Instruction Execution



SUB{.B} Wb, Ws, Wd Instruction

Three operand subtraction, register-to-register form:

SUB{.B} *Wb, Ws, Wd* (Wb) – (Ws) → Wd

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

Be careful:

while ADD Wx, Wy, Wz gives the same result as ADD Wy, Wx, Wz

The same is not true for

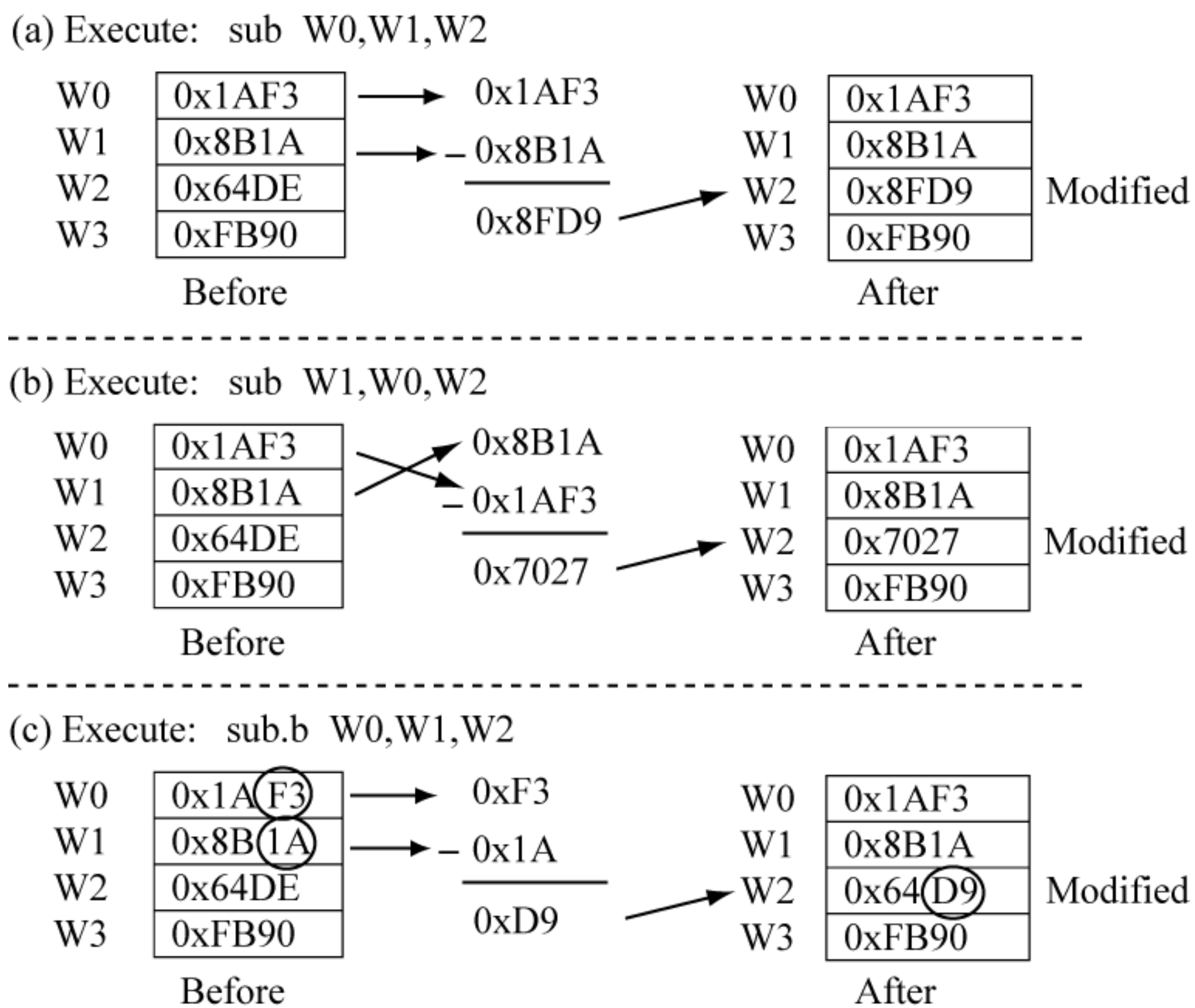
SUB Wx, Wy, Wz versus SUB Wy, Wx, Wz

SUB W0, W1, W2 (W0) – (W1) → W2

SUB W1,W0, W2 (W1) – (W0) → W2

SUB.B W0, W1, W2 Lower 8 bits of W0, W1 are subtracted and placed in the lower 8-bits of W2

SUB{.B} Wb, Ws, Wd Instruction Execution



From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”

Subtraction/Addition with Literals

- **Three operand addition/subtraction with literals:**

ADD{.B} *Wb, #lit5, Wd* (Wb) – #lit5 → Wd

SUB{.B} *Wb, #lit5, Wd* (Wb) – #lit5 → 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

ADD W0,#4,W2	(W0)+4 →W2
SUB.B W1,#8, W3	(W1) – 8 → W3
ADD W0, #60, W1	illegal, 60 is greater than 31!

ADD {.B} f {,WREG} Instruction

Two operand addition form:

ADD{.B} *f*	(f) + (WREG) → f
ADD{.B} *f*, WREG	(f) + (WREG) → WREG

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

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

ADD 0x1000	(0x1000) + (WREG) → 0x1000
ADD 0x1000,WREG	(0x1000) + (WREG) → WREG
ADD.B 0x1001, WREG	(0x1001) + (WREG.lsb) → WREG.lsb

Assembly Language Efficiency

- **The effects of the following instruction:**

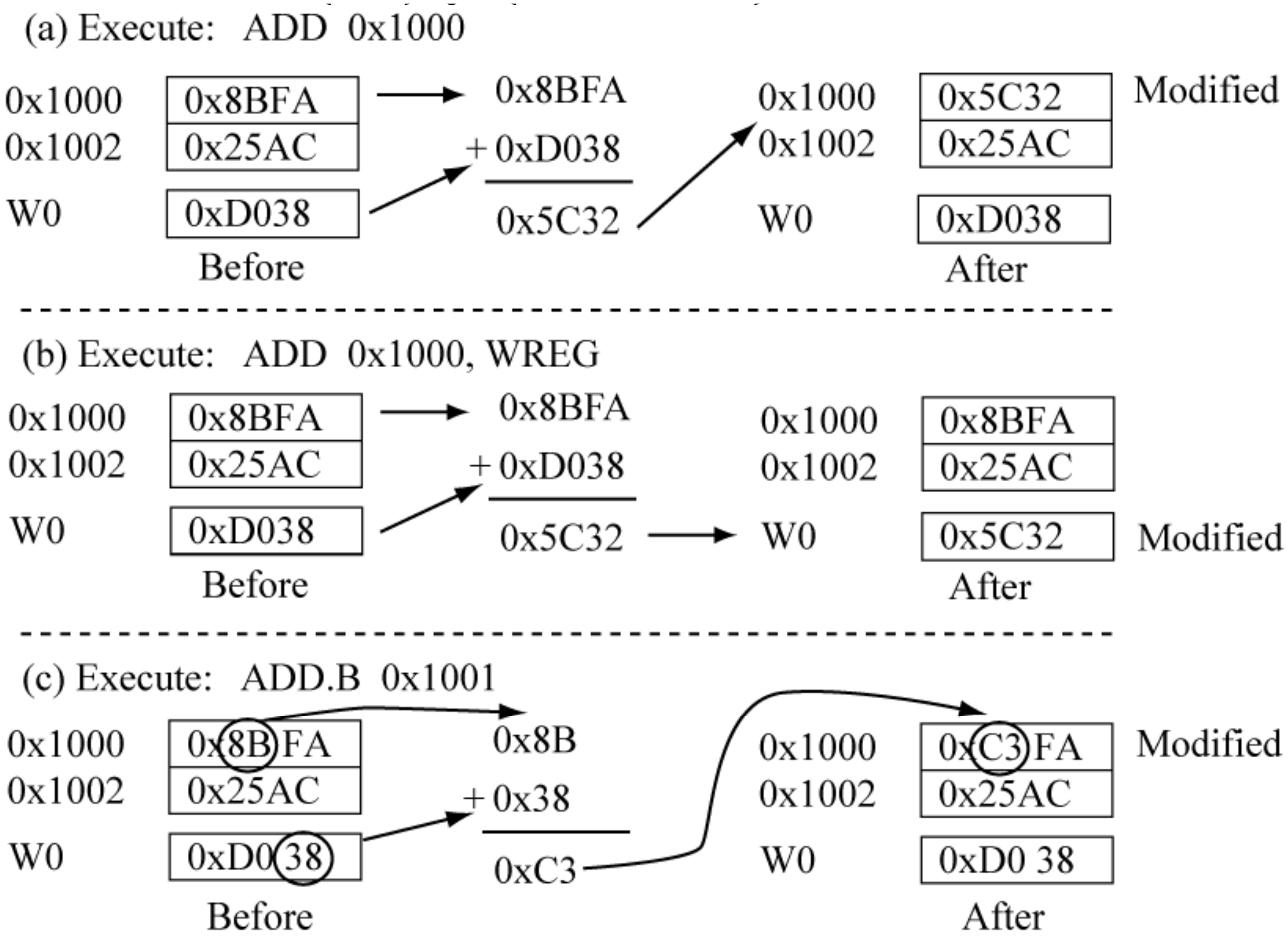
  ADD 0x1000 (0x1000) + (WREG) → 0x1000

  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} Instruction Execution



SUB{.B} f {,WREG} Instruction

- **Two operand subtraction form:**

SUB{.B} *f*	(f) – (WREG) → f
SUB{.B}*f*, WREG	(f) – (WREG) → WREG

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

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

SUB 0x1000	(0x1000) – (WREG) → 0x1000
SUB 0x1000,WREG	(0x1000) – (WREG) → WREG
SUB.B 0x1001, WREG	(0x1001) – (WREG.lsb) → WREG.lsb

Increment

Increment operation, register-to-register form:

INC{.B} *Ws, Wd* (Ws) +1 → Wd

Increment operation, memory to memory/WREG form:

INC{.B} *f* (*f*)+1→*f*

INC{.B} *f*, WREG (*f*) + 1 → WREG

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

Examples:

INC W2, W4	(W2) + 1 → W4
INC.B W3,W3	(W3.lsb)+1→W3.lsb
INC 0x1000	(0x1000) +1 → 0x1000
INC.B 0x1001,WREG	(0x1001)+1 → WREG.lsb

Decrement

Decrement operation, register-to-register form:

DEC{.B} *Ws, Wd* (Ws) – 1 → Wd

Increment operation, memory to memory/WREG form:

DEC{.B} *f* (*f*)–1→*f*

DEC{.B}*f*, WREG (*f*) – 1 → WREG

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

Examples:

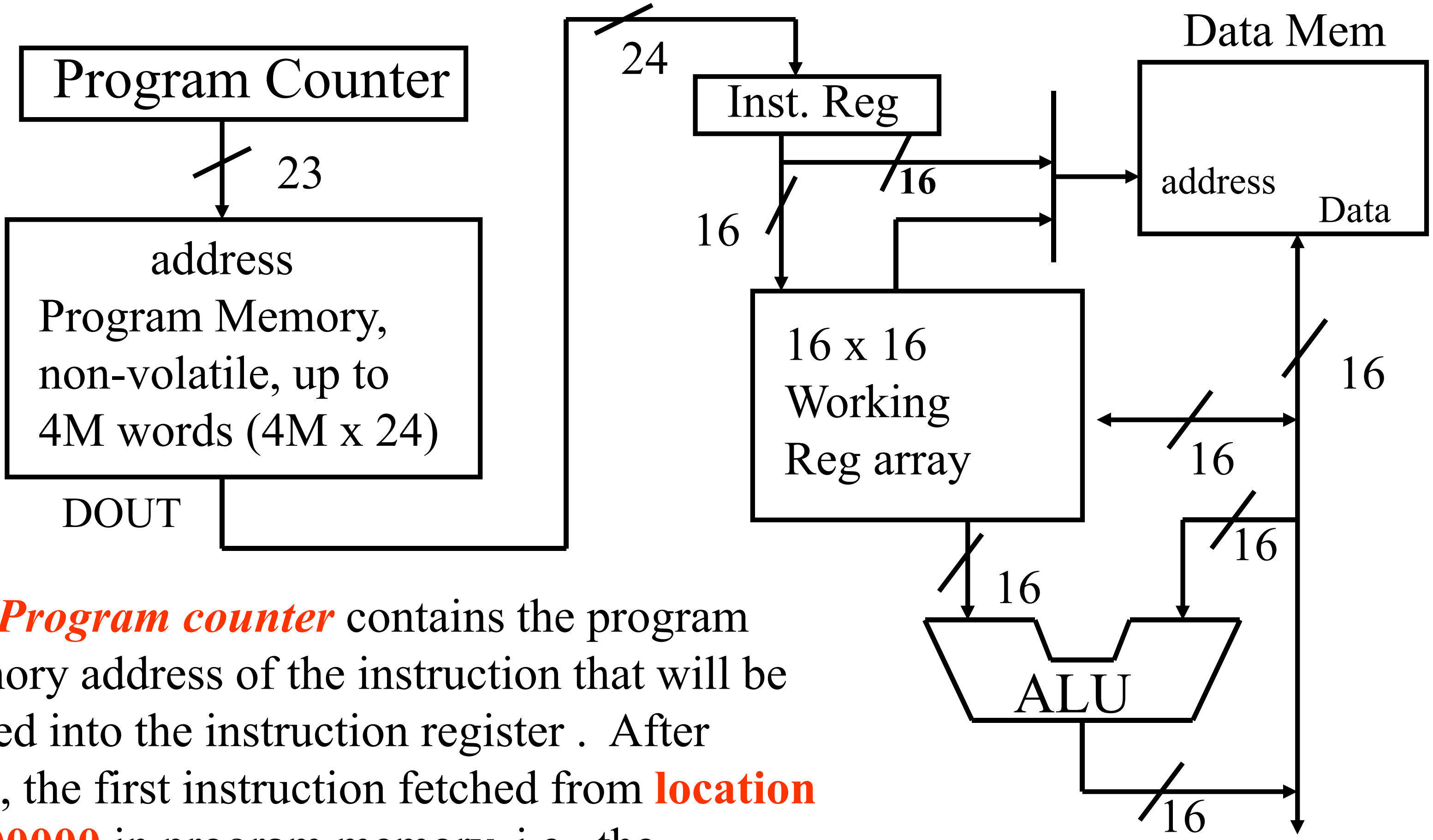
DEC W2, W4

DEC.B W3, W3

DEC 0x1000

DEC.B 0x1001,WREG

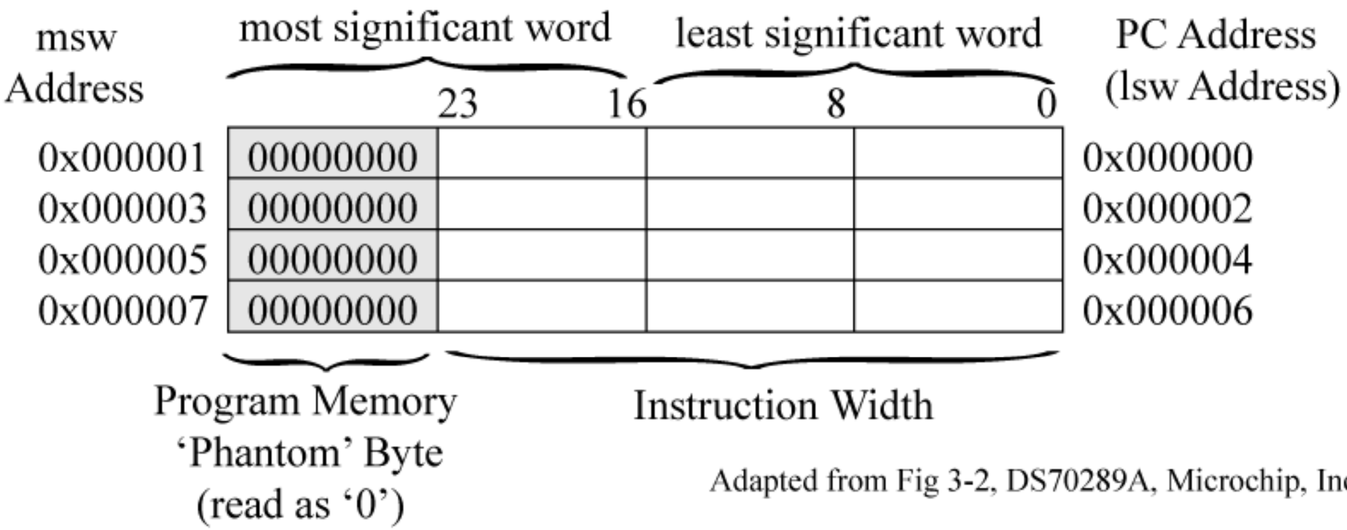
How is the instruction loaded?



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.**

17 x 17 Multiplier not shown

Program Memory Organization



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

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

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).

Goto location (goto)

How can the program counter be changed?

			BBBB	BBBB	BBBB	BBBB	BBBB	BBBB
			2222	1111	1111	1100	0000	0000
			3210	9876	5432	1098	7654	3210
goto	*Expr*	*lit23* → PC	0000	0100	nnnn	nnnn	nnnn	nnn0
			0000	0000	0000	0000	0nnn	nnnn

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).

nn..nn0 = 23-bit value that is loaded into the PC

The GOTO instruction requires two instruction words:

Assembly:		Machine Code:	
goto	0x000800	0x040800	First word
		0x000000	Second word

A GOTO instruction is an unconditional jump.

Valid addressing modes

- **What are valid addressing modes for instructions**

  Complete information can be found in table 19-2 of the PIC24H32GP202 datasheet.

TABLE 19-2: INSTRUCTION SET OVERVIEW (CONTINUED)

Base Instr #	Assembly Mnemonic	Assembly Syntax		Description	# of Words	# of Cycles	Status Flags Affected
40	MOV	MOV	f,Wn	Move f to Wn	1	1	None
		MOV	f	Move f to f	1	1	N,Z
		MOV	f,WREG	Move f to WREG	1	1	N,Z
		MOV	#lit16,Wn	Move 16-bit literal to Wn	1	1	None
		MOV.b	#lit8,Wn	Move 8-bit literal to Wn	1	1	None
		MOV	Wn,f	Move Wn to f	1	1	None
		MOV	Wso,Wdo	Move Ws to Wd	1	1	None
		MOV	WREG,f	Move WREG to f	1	1	N,Z
		MOV.D	Wns,Wd	Move Double from W(ns):W(ns + 1) to Wd	1	2	None
		MOV.D	Ws,Wnd	Move Double from Ws to W(nd + 1):W(nd)	1	2	None

From: Reese/Bruce/Jones, "Microcontrollers: From Assembly to C with the PIC24 Family".

Wso, Wsd, Wn

- **MOV Wso, Wdo**

Symbols used in opcode descriptions

Field	Description
Wnd	One of 16 destination working registers ∈ {W0..W15}
Wns	One of 16 source working registers ∈ {W0..W15}
WREG	W0 (working register used in file register instructions)
Ws	Source W register ∈ { Ws, [Ws], [Ws++], [Ws--], [++Ws], [--Ws] }
Wso	Source W register ∈ { Wns, [Wns], [Wns++], [Wns--], [++Wns], [--Wns], [Wns+Wb] }
Wd	Destination W register ∈ { Wd, [Wd], [Wd++], [Wd--], [++Wd], [--Wd] }
Wdo	Destination W register ∈ { Wnd, [Wnd], [Wnd++], [Wnd--], [++Wnd], [--Wnd],
Wn	One of 16 working registers ∈ {W0..W15}
Wb	Base W register ∈ {W0..W15}

ADD forms

- **ADD Wb, Ws, Wd**

Field	Description
Ws	Source W register ∈ { Ws, [Ws], [Ws++], [Ws--], [++Ws], [--Ws] }
Wd	Destination W register ∈ { Wd, [Wd], [Wd++], [Wd--], [++Wd], [--Wd] }
Wb	Base W register ∈ {W0..W15}

- **Legal:**

  ADD W0, W1, W2

  ADD W0, [W1], [W4]

- **Illegal**

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

Simple Program (example)

- **Sample programs written in C, translated (compiled) to PIC 24uC assembly language**

```
C Program equivalent

  #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

i = 100;	`mov.b #100,W0 ;WREG = 100 = 0x64` `mov.b WREG,0x0800 ;i = WREG`
i = i + 1;	`inc.b 0x0800 ;i = i + 1`
j = i;	`mov.b 0x0800,WREG ;WREG = i` `mov.b WREG,0x0801 ;j = WREG`
j = j - 1;	`dec.b 0x0801 ;j = j - 1`

k = j + i;	`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*.

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

mptst_byte.s

```
 .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
```

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

Declare the __reset label as global – it is 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).

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.

mptst_byte.s (cont.)

```
.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 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.

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   (WREG is W0)
   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

mptst_byte.s (cont.)

```
done:
  goto    done   ;loop forever


  .end
```

A label that is the target of a *goto* instruction. Lables are **case sensitive** (instruction mnemonics and assembler directives are not case sensitive.

A comment

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

An Alternate Solution

C Program equivalent

```
#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
```

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

```
;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


 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)
 ;;write variables to memory
 mov.b W1,W0       ; W0 = i
 mov.b WREG,i      ; 0x800 (i) = W0
 mov.b W2,W0       ; W0 = j
 mov.b WREG,j      ; 0x801 (j) = W0
 mov.b W3,W0       ; W3 = k
 mov.b WREG,k      ; 0x802 (k) = W0
```

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.

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 mpstst_bytes.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.**

	Instruction Cycles
`mov #__SP_init, W15`	1
`mov #__SPLIM_init,W0`	1
`mov W0,SPLIM`	1
`mov.b #avalue, W0`	1
`mov.b WREG,i`	1
`inc.b    i`	1
`mov.b    i,WREG`	1
`mov.b    WREG,j`	1
`dec.b    j`	1
`mov.b    i,WREG`	1
`add.b    j,WREG`	1
`mov.b    WREG,k`	1
V 0.2 Total	12

What if we used 16-bit variable instead of 8-bit variables?

C Program equivalent

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

A uint16 variable is 16 bits (1 byte)

```
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
```

```
 .include "p24Hxxxx.inc"
 .global __reset
 .bss   ;reserve space for variables
i:        .space 2
j:        .space 2
k:        .space 2

.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 = 2048

; i = 2048;
    mov  #avalue, W0        ; W0 = 2048
    mov  WREG,i             ; i = 2048

; i = i + 1;
    inc   i                 ; i = i + 1
; j = i
    mov    i,WREG           ; W0 = i
    mov    WREG,j           ; j = W0
; j = j - 1;
    dec    j               ; j= j - 1
; k = j + i
    mov    i,WREG          ; W0 = i
    add    j,WREG          ; W0 = W0+j   (WREG is W0)
    mov    WREG,k          ; k = W0
done:
    goto    done    ;loop forever
```

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

```
#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


 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)
 ;;write variables to memory
 mov W1,i        ; 0x800 (i) = W1
 mov W2,j        ; 0x802 (j) = W2
 mov W3,k        ; 0x804 (k) = W3
```

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.

	Instruction Cycles
`mov #__SP_init, W15`	1
`mov #__SPLIM_init,W0`	1
`mov W0,SPLIM`	1
`mov #avalue, W0`	1
`mov WREG,i`	1
`inc    i`	1
`mov    i,WREG`	1
`mov    WREG,j`	1
`dec    j`	1
`mov    i,WREG`	1
`add    j,WREG`	1
`mov    WREG,k`	1
Total	12

16 bit operations vs. 8 bit operations

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:

Time	Frequency
milliseconds (ms = 10^{-3} s)	kilohertz (kHz = 10^{3} Hz)
microseconds (µs = 10^{-6} s)	megahertz (MHz = 10^{6} Hz)
nanoseconds (ns = 10^{-9} s)	gigahertz (GHz = 10^{9} Hz)

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 PIC24 is found on the class UVLE site.**

PICmicro Survey

	PIC16F87x	PIC18F242	PIC24H
Instruction width	14 bits	16 bits	24 bits
Program Memory	8K instruction	8K instructions	~10K instructions
Data Memory	386 bytes	1537 bytes	2048 bytes
Clock Speed	Max 20 MHz 4 clks = 1 instruction	Max 40 MHz 4 clks = 1 instruction	Max 80 MHz 2 clks = 1 instruction

The PIC24H can execute about 6x faster than the PIC18F242

Summary

- **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 µC assembly language program in the MPLAB IDE
- **Understand the relationship between instruction cycles and machine cycles**