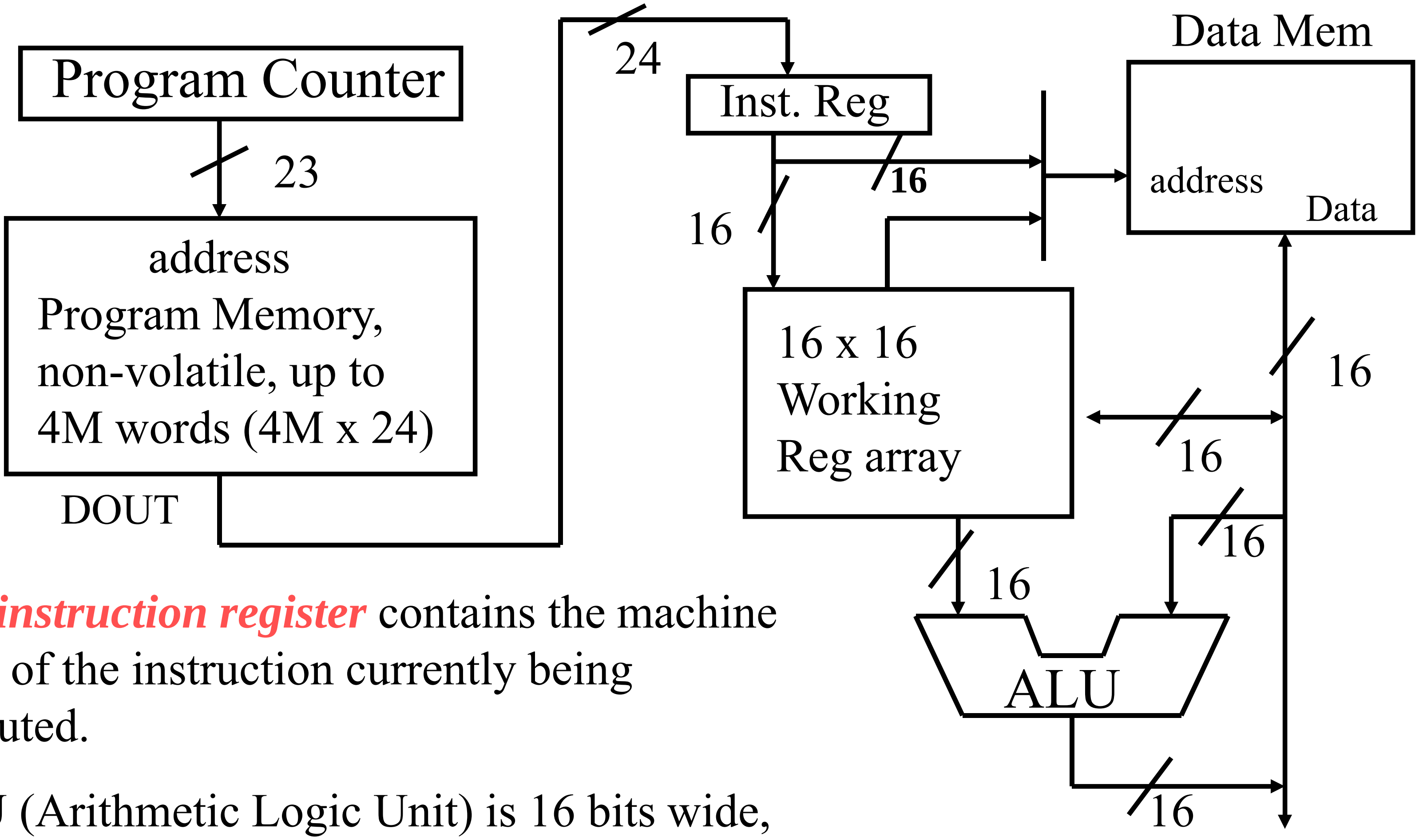
# Microcontroller (μC) vs. Microprocessor (μP)

- 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

| Features | Comments |
|---|---|
| Instruction width | 24 bits |
| On-chip program memory (non-volatile, electrically erasable) | PIC24HJ32GP202 has 32Ki bytes/11264 instructions, architecture supports 24Mibytes/4Mi instructions) |
| On-chip Random Access Memory (RAM) , volatile | PIC24HJ32GP202 has 2048 bytes, architecture supports up 65536 bytes |
| Clock speed | DC to 80 MHz |
| 16-bit Architecture | General purpose registers, 71 instructions not including addressing mode variants |
| On-chip modules | Async serial IO, I2C, SPI, A/D, three 16-bit timers, one 8-bit timer, comparator |

# PIC24 Core (Simplified Block Diagram)



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.

17 x 17 Multiplier
not shown

# Memory Organization

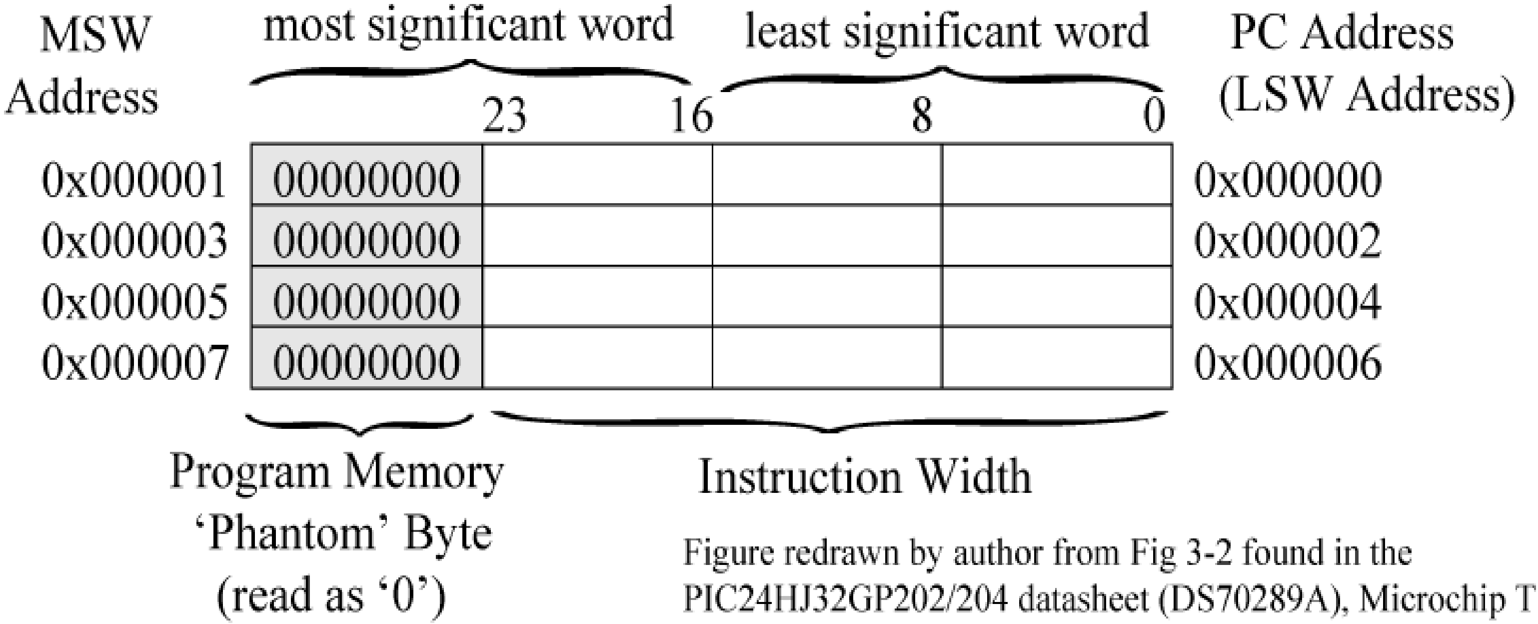
Memory on the PIC24 µC family is split into two types: **Program Memory** and **Data Memory**.

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



| MSW Address | | | | | PC Address (LSW Address) |
|---|---|---|---|---|---|
| 0x000001 | 00000000 | | | | 0x000000 |
| 0x000003 | 00000000 | | | | 0x000002 |
| 0x000005 | 00000000 | | | | 0x000004 |
| 0x000007 | 00000000 | | | | 0x000006 |

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

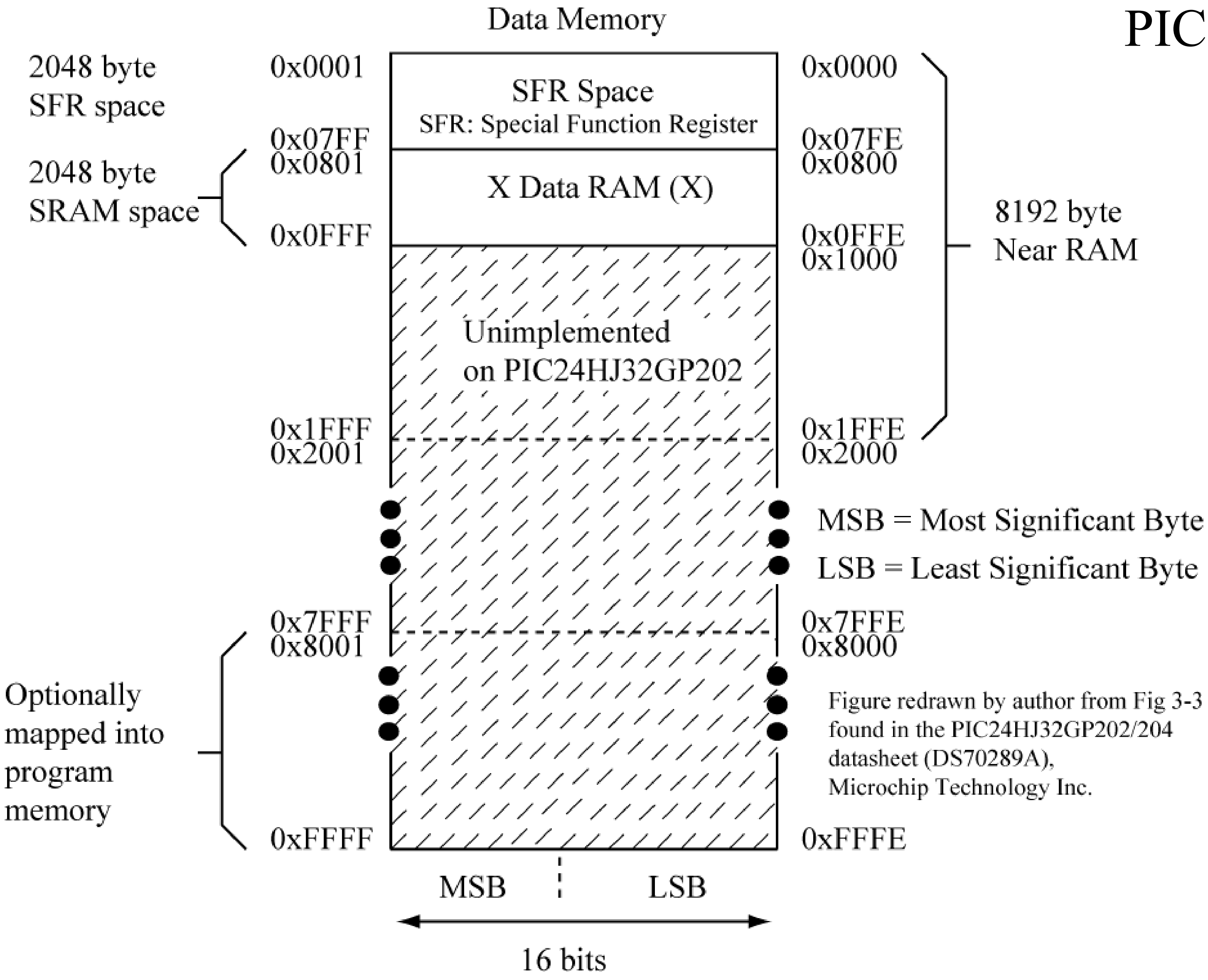
PC is 23 bits wide, but instructions start on even word boundaries (the PC least significant bit is always 0), so the PC can address 4 Mi instructions.

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

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

# Data Memory Organization

## Data memory for PIC24HJ32GP202



Data Memory

2048 byte SFR space

2048 byte SRAM space

Optionally mapped into program memory

0x0001
0x07FF
0x0801
0x0FFF
0x1FFF
0x2001
0x7FFF
0x8001
0xFFFF

SFR Space
SFR: Special Function Register

X Data RAM (X)

Unimplemented on PIC24HJ32GP202

0x0000
0x07FE
0x0800
0x0FFE
0x1000
0x1FFE
0x2000
0x7FFE
0x8000
0xFFFE

8192 byte Near RAM

MSB = Most Significant Byte

LSB = Least Significant Byte

MSB LSB

16 bits

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

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.

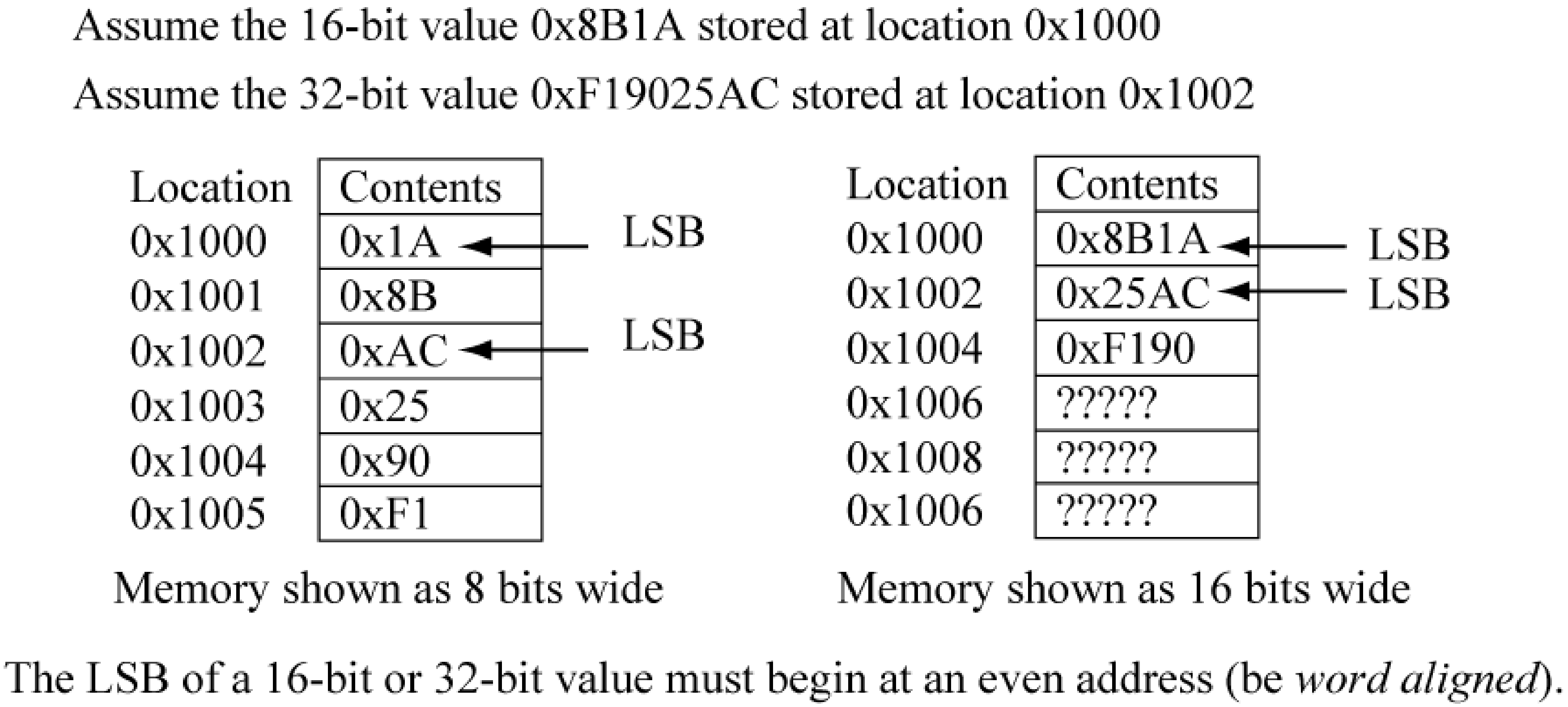
| Size | Unsigned Range |
|---|---|
| 8-bits | 0 to $2^8$-1 (0 to 255, 0 to 0xFF) |
| 16-bit | 0 to $2^{16}$-1 (0 to 65536, 0 to 0xFFFF) |
| 32-bit | 0 to $2^{32}$-1 (0 to 4,294,967,295), 0 to 0xFFFFFFFF) |

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

| Location | Contents | |
|---|---|---|
| 0x1000 | 0x1A | ← LSB |
| 0x1001 | 0x8B | |
| 0x1002 | 0xAC | ← LSB |
| 0x1003 | 0x25 | |
| 0x1004 | 0x90 | |
| 0x1005 | 0xF1 | |

Memory shown as 8 bits wide

| Location | Contents | |
|---|---|---|
| 0x1000 | 0x8B1A | ← LSB |
| 0x1002 | 0x25AC | ← LSB |
| 0x1004 | 0xF190 | |
| 0x1006 | ????? | |
| 0x1008 | ????? | |
| 0x1006 | ????? | |

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*).

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

# 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

| Dest<br>Source | Memory | Register direct | Register indirect |
|---|---|---|---|
| Literal | X | MOV{.B} #lit8/16, Wnd<br>*lit → Wnd* | X |
| Memory | X | MOV $f_{ALL}$, Wnd<br>MOV{.B} f, {WREG}<br>$(f_{\{ALL\}}) \rightarrow Wnd/WREG$ | X |
| Register direct | MOV Wns, $f_{ALL}$<br>MOV{.B} WREG, f<br>$(Wns/WREG) \rightarrow f_{\{ALL\}}$ | MOV{.B} Wso, Wdo<br>*(Wso) → Wdo* | MOV{.B} Wso, [Wdo]<br>*(Wso) → (Wdo)* |
| Register indirect | X | MOV{.B} [Wso], Wdo<br>*((Wso)) → Wdo* | MOV{.B} [Wso], [Wdo]<br>*((Wso)) → (Wdo)* |

Yellow shows varying forms of the same instruction

Key:

| | |
|---|---|
| MOV{.B} #lit8/16, Wnd | PIC24 assembly |
| *lit → Wnd* | Data transfer |

f: near memory (0…8095)

$f_{ALL}$: all of memory (0…65534)

# *MOV{.B} Wso, Wdo* Instruction

"Copy contents of Wso register to Wdo register". General form:

mov{.b}  Wso, Wdo    (Wso) → 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).

| | | | |
|---|---|---|---|
| mov | W3, W5 | (W3) → W5 | (word operation) |
| mov.b | W3, W5 | (W3.LSB) → W5.LSB | (byte operation) |

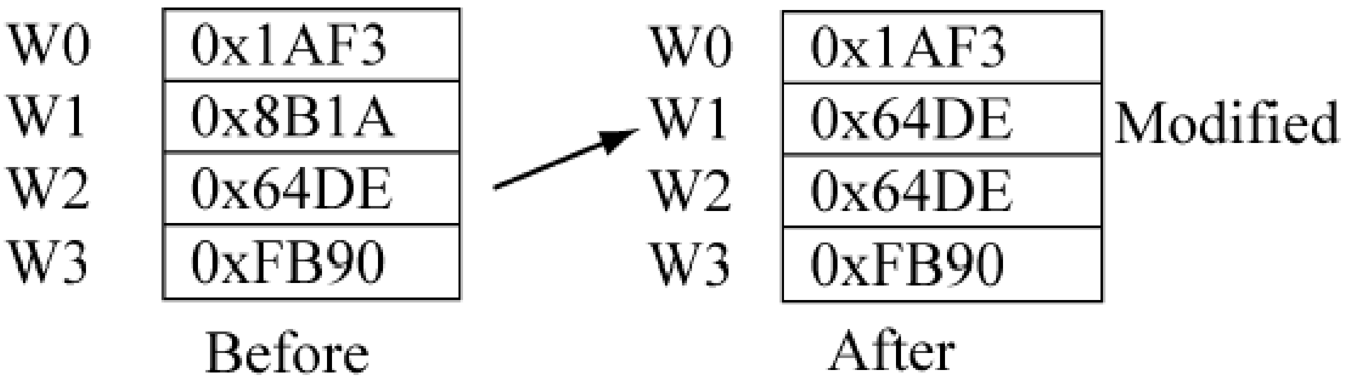
Contents of working register W3 copied to working register W5.

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 Wso, Wdo* Instruction Execution

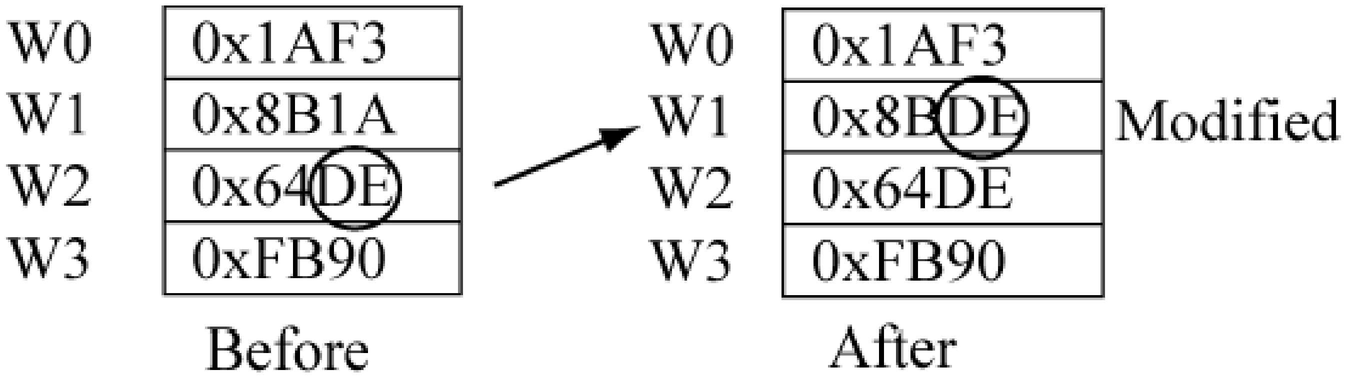
(a) Execute: mov W2,W1 (word mode operation)



| Register | Before | After |
|---|---|---|
| W0 | 0x1AF3 | 0x1AF3 |
| W1 | 0x8B1A | 0x64DE Modified |
| W2 | 0x64DE | 0x64DE |
| W3 | 0xFB90 | 0xFB90 |

---

(b) Execute: mov.b W2,W1 (byte mode operation)



| Register | Before | After |
|---|---|---|
| W0 | 0x1AF3 | 0x1AF3 |
| W1 | 0x8B1A | 0x8BDE Modified |
| W2 | 0x64DE | 0x64DE |
| W3 | 0xFB90 | 0xFB90 |

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

# *MOV Wso, Wdo* Instruction Format

(a)

```
BBBB BBBB BBBB BBBB BBBB BBBB
2222 1111 1111 1100 0000 0000
3210 9876 5432 1098 7654 3210
```

mov{.b} *Wso,Wdo*

```
0111 1www wBhh hddd dggg ssss
```

(*Wso*) → *Wdo* (reg. direct)
(indirect addressing modes not shown)

`wwww` = base register (*Wb)* for indirect offset addressing mode [*Wso*/*Wdo* + *Wb*]; otherwise 0

`B` = 0 for word, 1 for byte

`hhh` = *Wdo* addressing mode (Register direct = 000)

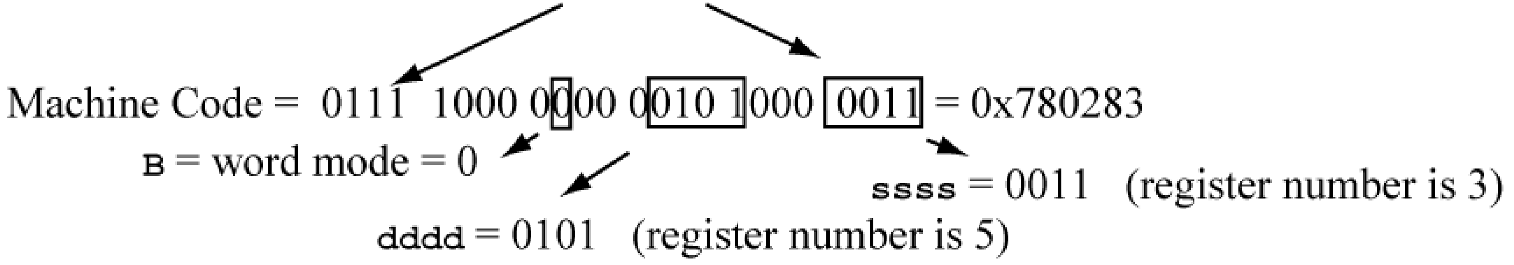
`dddd` = *Wdo* register number (0 to 15)

`ggg` = *Wso* addressing mode (Register direct = 000)

`ssss` = *Wso* register number (0 to 15)

(b) Assembly:
mov W3,W5

Machine Code:
0x780283



Machine Code = 0111 1000 0000 0010 1000 0011 = 0x780283

`B` = word mode = 0

`ssss` = 0011 (register number is 3)

`dddd` = 0101 (register number is 5)

`ggg`, `hhh`, `wwww` fields are all 0 because indirect addressing is not used

(c) mov.b W3,W5 0x784283 Byte mode, only difference is `B` = 1

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:

MOV Wns, *f* (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)

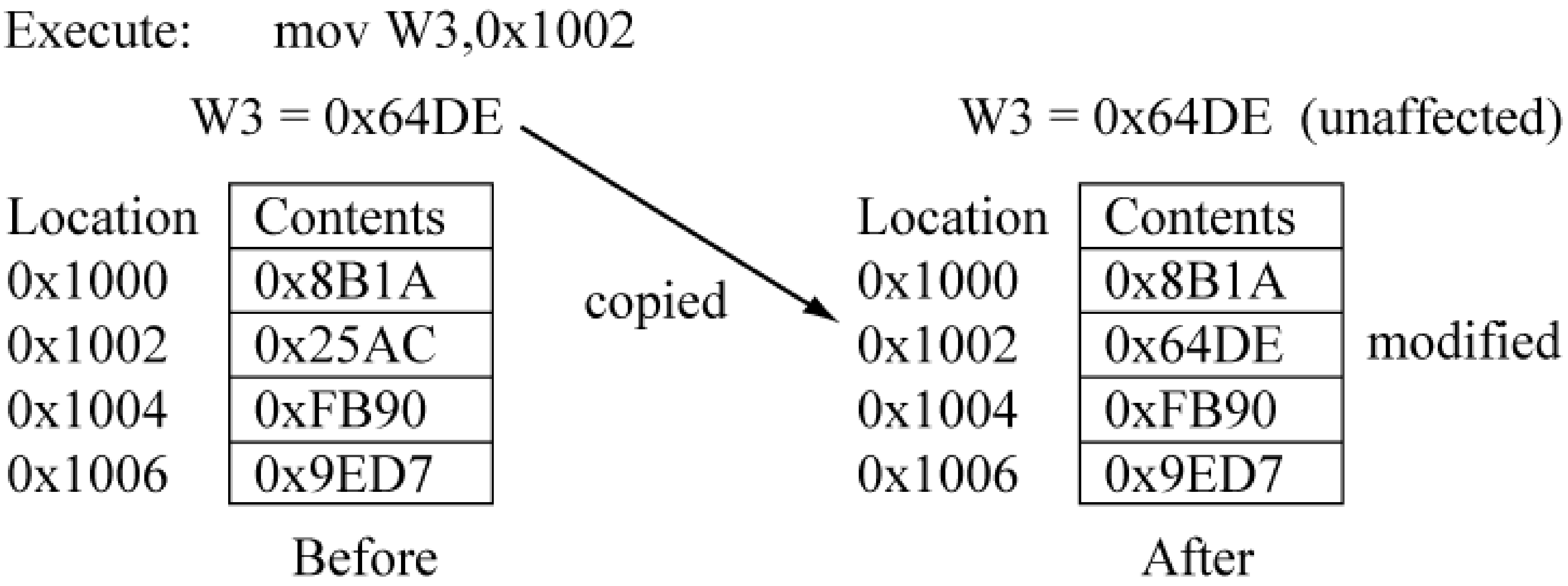
MOV W3, 0x1000 (W3) → 0x1000

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.

# *MOV Wns, f* Instruction Execution



Execute: mov W3,0x1002

W3 = 0x64DE

| Location | Contents |
|---|---|
| 0x1000 | 0x8B1A |
| 0x1002 | 0x25AC |
| 0x1004 | 0xFB90 |
| 0x1006 | 0x9ED7 |

Before

copied

W3 = 0x64DE (unaffected)

| Location | Contents |
|---|---|
| 0x1000 | 0x8B1A |
| 0x1002 | 0x64DE |
| 0x1004 | 0xFB90 |
| 0x1006 | 0x9ED7 |

modified

After

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

# *MOV Wns, f* Instruction Format

(a)

```
                BBBB BBBB BBBB BBBB BBBB BBBB
                2222 1111 1111 1100 0000 0000
mov Wns,f       3210 9876 5432 1098 7654 3210

(Wns) -> f      1000 1fff ffff ffff ffff ssss
```

`f ... f` = upper 15 bits of 16-bit address (lower bit assumed = 0)

`ssss` = *Wns* register number (0 to 15)

---

(b) Assembly: mov W3,0x1002

Machine Code: 0x888013

Machine Code = 1000 1 $\boxed{000\ 1000\ 0000\ 0001}$ $\boxed{0011}$ = 0x888013

`f ... f` = $\boxed{0001\ 0000\ 0000\ 001}$0 (upper 15-bits of 0x1002)

`ssss` = 0011 (register number is 3)

From: Reese/Bruce/Jones, "Microcontrollers: From Assembly to C with the PIC24 Family". V 0.2 17

# *MOV f, Wnd* Instruction

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

MOV *f*, Wnd (f) → 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).

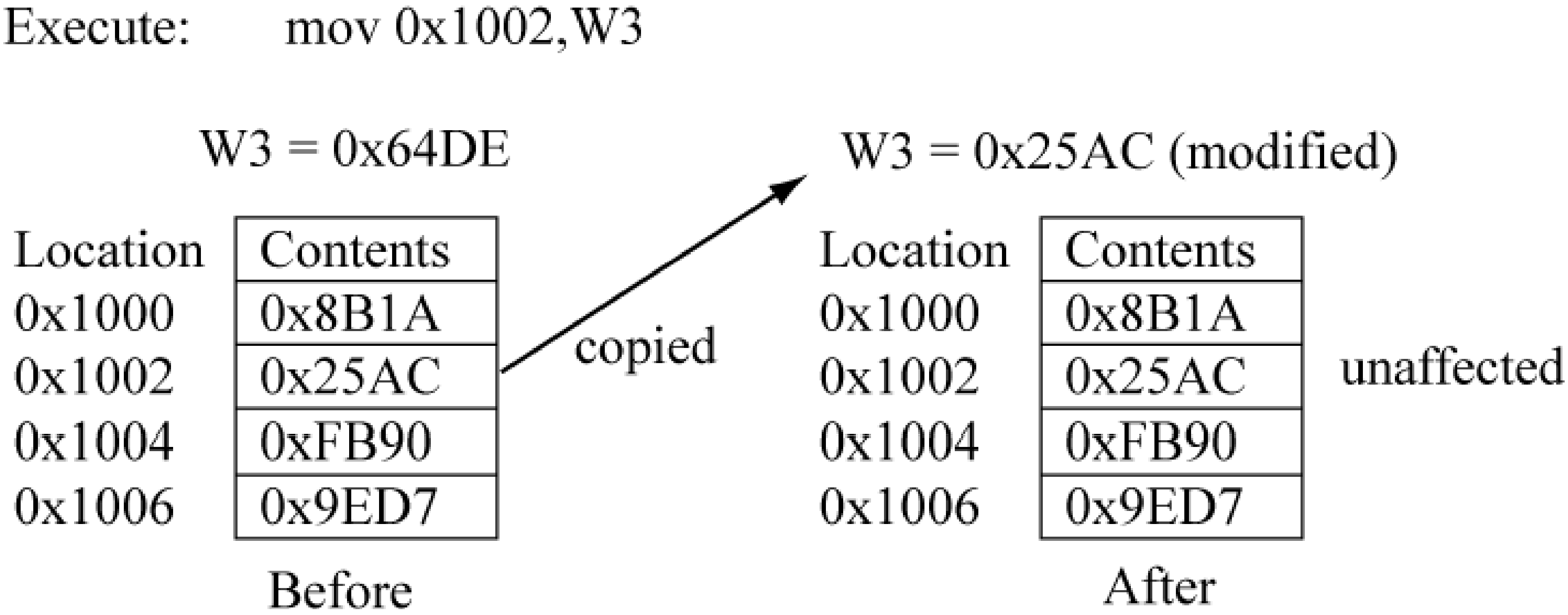
MOV 0x1000, W3 (0x1000) → W3

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

W3 = 0x64DE

| Location | Contents |
|---|---|
| 0x1000 | 0x8B1A |
| 0x1002 | 0x25AC |
| 0x1004 | 0xFB90 |
| 0x1006 | 0x9ED7 |

Before

W3 = 0x25AC (modified)

| Location | Contents |
|---|---|
| 0x1000 | 0x8B1A |
| 0x1002 | 0x25AC |
| 0x1004 | 0xFB90 |
| 0x1006 | 0x9ED7 |

After

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:

MOV{.B} WREG, *f* (WREG) → f

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

MOV WREG, 0x1000 (W0) → 0x1000

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

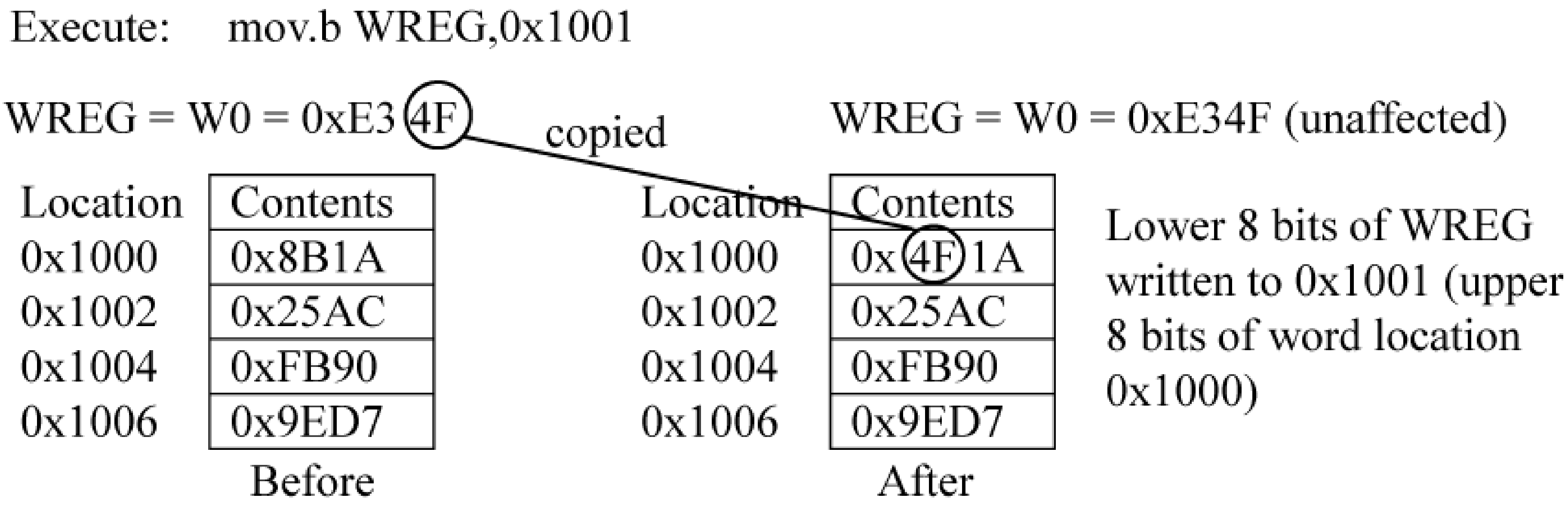
Can be used for either WORD or BYTE operations:

| | |
|---|---|
| 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



Execute: mov.b WREG,0x1001

WREG = W0 = 0xE34F

copied

WREG = W0 = 0xE34F (unaffected)

| Location | Contents |
|---|---|
| 0x1000 | 0x8B1A |
| 0x1002 | 0x25AC |
| 0x1004 | 0xFB90 |
| 0x1006 | 0x9ED7 |

Before

| Location | Contents |
|---|---|
| 0x1000 | 0x4F1A |
| 0x1002 | 0x25AC |
| 0x1004 | 0xFB90 |
| 0x1006 | 0x9ED7 |

After

Lower 8 bits of WREG written to 0x1001 (upper 8 bits of word location 0x1000)

A byte copy operation is shown.

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

# *MOV{.B} WREG, f* Instruction Format

| | | | | | | |
|---|---|---|---|---|---|---|
| | BBBB | BBBB | BBBB | BBBB | BBBB | BBBB |
| mov{.b} WREG, *f* | 2222<br>3210 | 1111<br>9876 | 1111<br>5432 | 1100<br>1098 | 0000<br>7654 | 0000<br>3210 |
| (WREG) → *f* | 1011 | 0111 | 1B1f | ffff | ffff | ffff |

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

`B` = 0 for word, 1 for byte

| Assembly: | Machine Code: | |
|---|---|---|
| mov WREG, 0x1000 | 0xB7B000 | (`B` bit = 0 since word operation) |
| mov.b WREG, 0x1000 | 0xB7F000 | (`B` bit = 1 since byte operation) |
| mov.b WREG, 0x1001 | 0xB7F001 | (bytes can be written to odd addresses) |

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

# *MOV{.B} f {,WREG}* Instruction

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

| | |
|---|---|
| MOV{.B} *f*, WREG | (*f*)→ WREG |
| MOV{.B} *f* | (*f*)→ *f* |

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

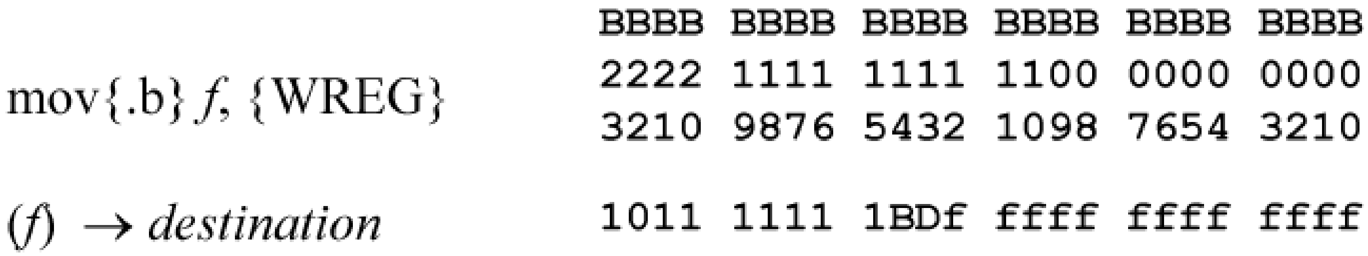
Can be used for either WORD or BYTE operations:

| | |
|---|---|
| MOV 0x1000, WREG | word operation |
| MOV.B 0x1001, WREG | only lower 8-bits of W0 are affected. |
| MOV 0x1000 | 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 *MOV f,Wnd* instruction cannot be used for byte operations!

# *MOV{.B} ƒ {,WREG}* Format



mov{.b} *f*, {WREG}

(*f*) → *destination*

```
BBBB BBBB BBBB BBBB BBBB BBBB
2222 1111 1111 1100 0000 0000
3210 9876 5432 1098 7654 3210

1011 1111 1BDf ffff ffff ffff
```

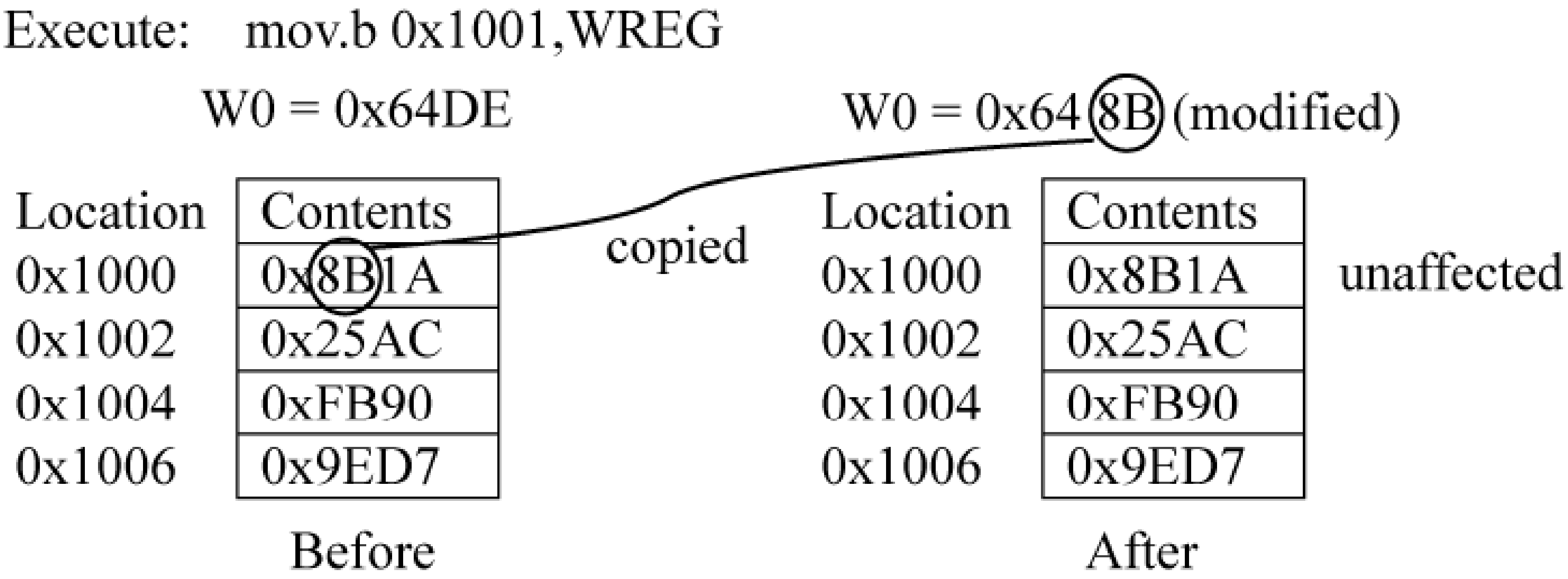
Destination is either *f* or WREG.

f..f = 13-bit address (first 8192 bytes of data memory)
B = ‘0’ for word, ‘1’ for byte
D = destination = ‘0’ for WREG, ‘1’ for *f*

| Assembly: | Machine Code: | |
|---|---|---|
| mov 0x1000,WREG | 0xBF9000 | (B bit = 0 since word operation,<br>D bit = 0 since WREG destination ) |
| mov.b 0x1000 | 0xBFF000 | (B bit = 1 since byte operation,<br>D bit = 1 since *f* destination) |

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

# *MOV.{B} f, WREG* Instruction Execution



Execute: mov.b 0x1001,WREG

W0 = 0x64DE

| Location | Contents |
|---|---|
| 0x1000 | 0x8B1A |
| 0x1002 | 0x25AC |
| 0x1004 | 0xFB90 |
| 0x1006 | 0x9ED7 |

Before

copied

W0 = 0x648B (modified)

| Location | Contents |
|---|---|
| 0x1000 | 0x8B1A |
| 0x1002 | 0x25AC |
| 0x1004 | 0xFB90 |
| 0x1006 | 0x9ED7 |

unaffected

After

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

# 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    lit16 → Wnd (word operation)

MOV.B #lit8, Wnd    lit8 → Wnd.lsb (byte operation)

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

Examples:

MOV #0x1000, W2    0x1000 → W2

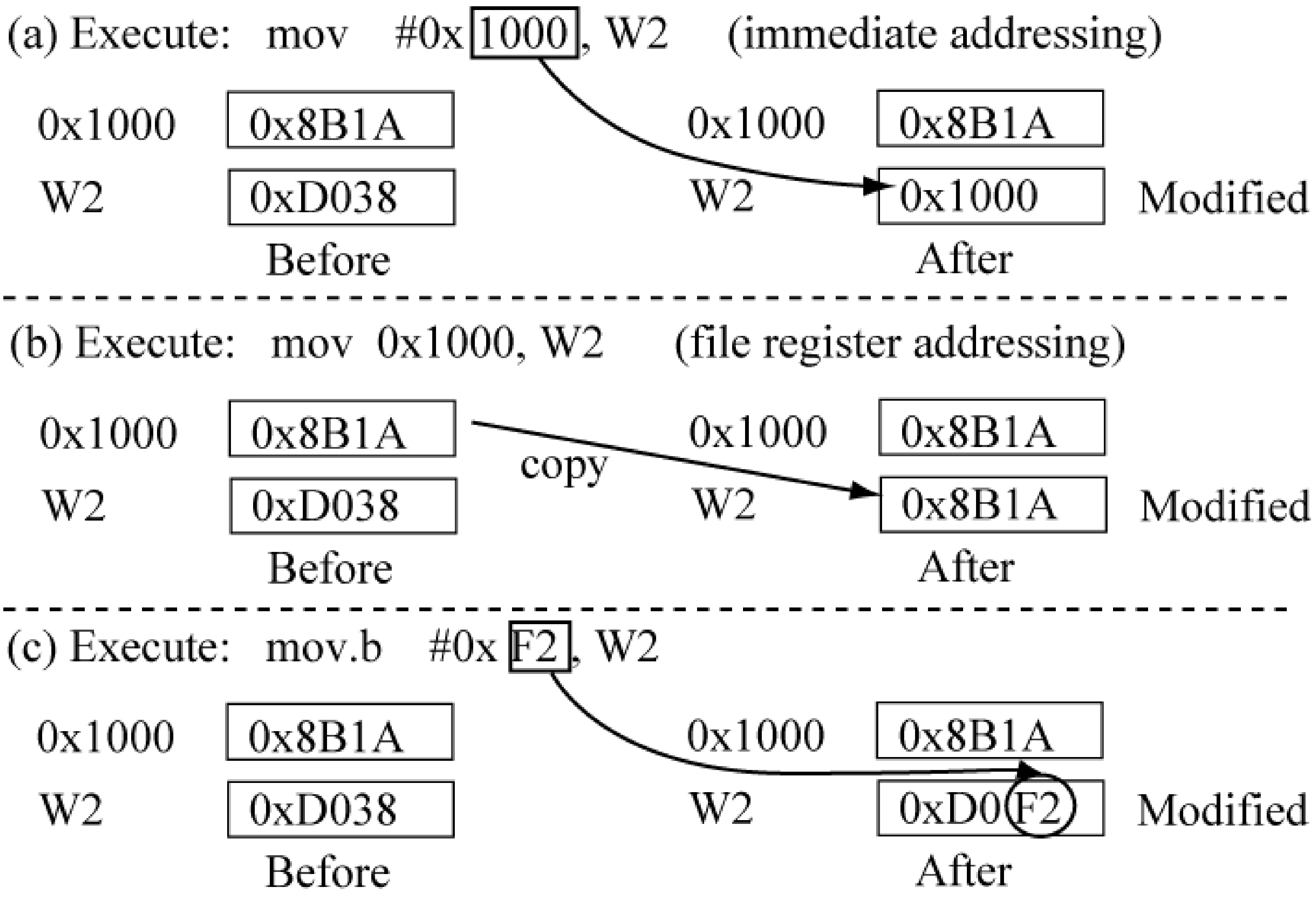
MOV.B #0xAB, W3    0xAB → W3.lsb

# 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)

Before:
0x1000 0x8B1A
W2 0xD038

After:
0x1000 0x8B1A
W2 0x1000 Modified

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

Before:
0x1000 0x8B1A
W2 0xD038

copy

After:
0x1000 0x8B1A
W2 0x8B1A Modified

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

Before:
0x1000 0x8B1A
W2 0xD038

After:
0x1000 0x8B1A
W2 0xD0F2 Modified

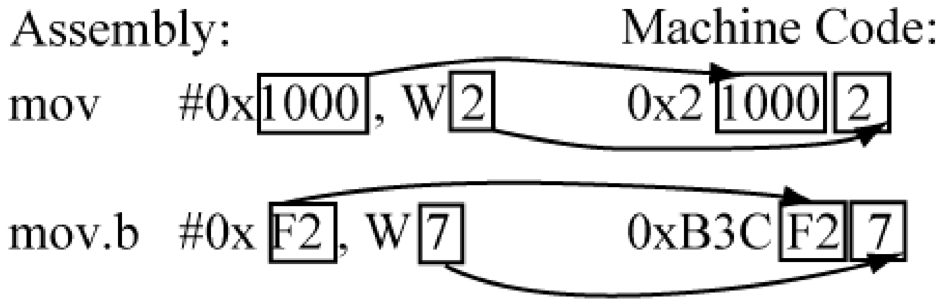
From: Reese/Bruce/Jones, “Microcontrollers: From Assembly to C with the PIC24 Family”.

# MOV Literal Instruction Formats

| | | BBBB | BBBB | BBBB | BBBB | BBBB | BBBB |
|---|---|---|---|---|---|---|---|
| | | 2222 | 1111 | 1111 | 1100 | 0000 | 0000 |
| | | 3210 | 9876 | 5432 | 1098 | 7654 | 3210 |
| mov *#lit16*, *Wn* | *#lit16* → *Wn* | 0010 | kkkk | kkkk | kkkk | kkkk | dddd |
| mov.b *#lit8*, *Wn* | *#lit8* → *Wn* | 1011 | 0011 | 1100 | kkkk | kkkk | dddd |

*#lit16*: 16-bit literal
*#lit8*: 8-bit literal

`k ... k` = literal
`dddd` = *Wn* register number (0 to 15)


Assembly: Machine Code:
mov #0x1000, W2 0x210002
mov.b #0xF2, W7 0xB3CF27

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

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

# 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).

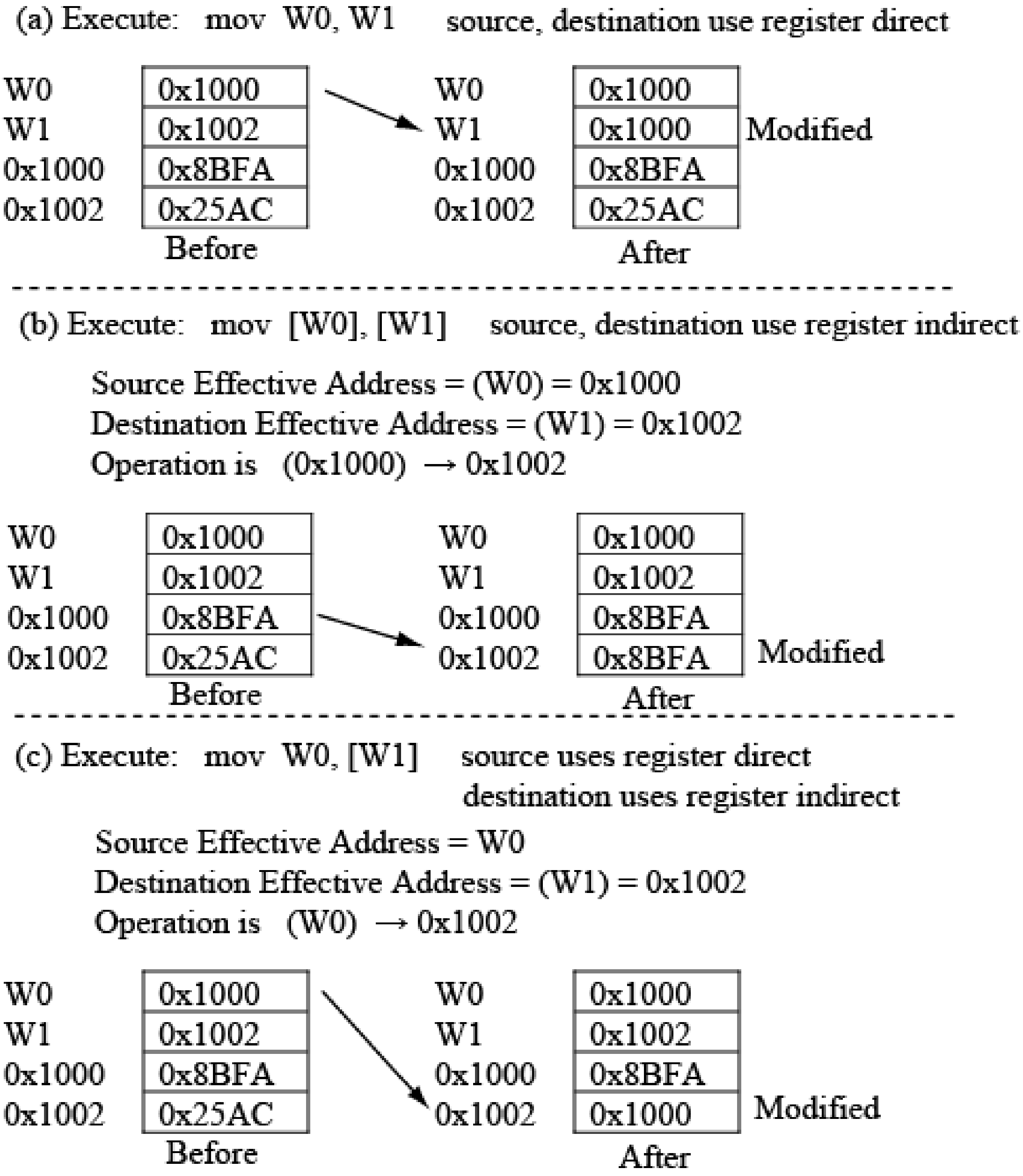
The MOV instruction copies the content of the Source Effective Address to the Destination Effect Address, or:

(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 indirect

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 |

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) → Wd

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

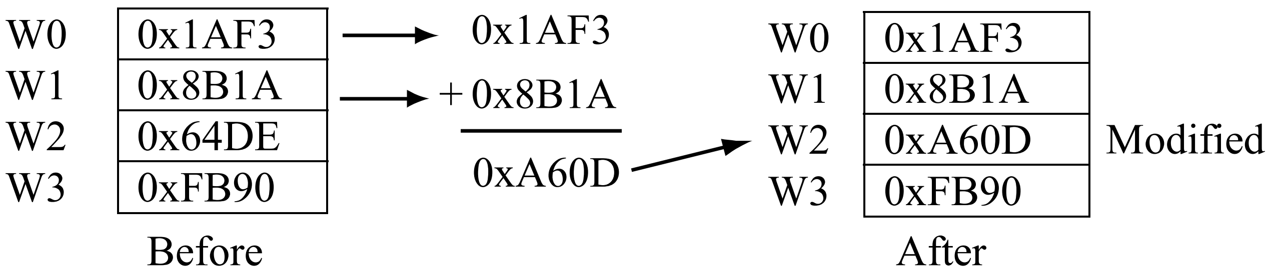
ADD W0, W1, W2 (W0) + (W1) → W2

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* Execution

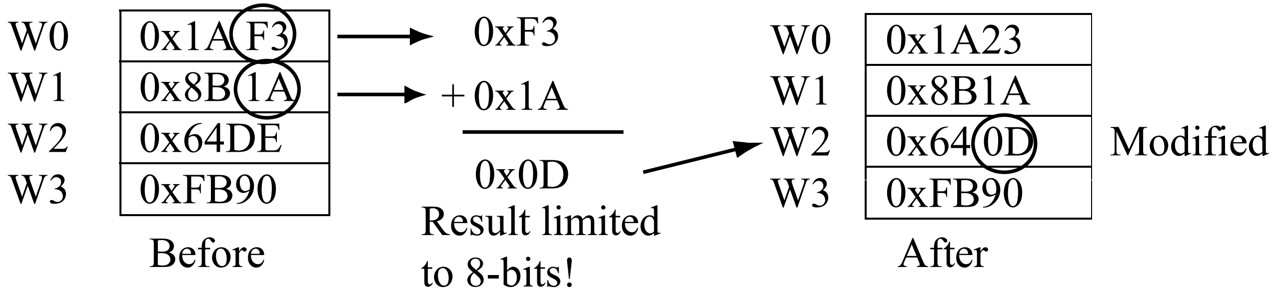
(a) Execute: add W0,W1,W2



| | Before | After | |
|---|---|---|---|
| W0 | 0x1AF3 | 0x1AF3 | |
| W1 | 0x8B1A | 0x8B1A | |
| W2 | 0x64DE | 0xA60D | Modified |
| W3 | 0xFB90 | 0xFB90 | |

$$\begin{array}{r} \texttt{0x1AF3} \\ +\,\texttt{0x8B1A} \\ \hline \texttt{0xA60D} \end{array}$$

(b) Execute: add.b W0,W1,W2

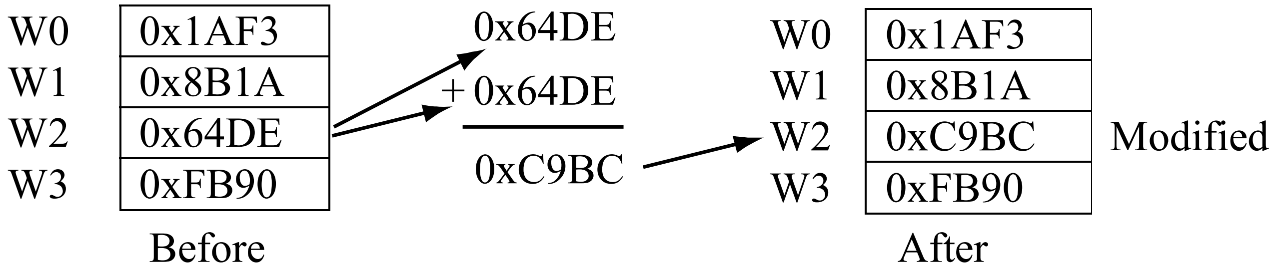


| | Before | After | |
|---|---|---|---|
| W0 | 0x1AF3 | 0x1A23 | |
| W1 | 0x8B1A | 0x8B1A | |
| W2 | 0x64DE | 0x640D | Modified |
| W3 | 0xFB90 | 0xFB90 | |

$$\begin{array}{r} \texttt{0xF3} \\ +\,\texttt{0x1A} \\ \hline \texttt{0x0D} \end{array}$$

Result limited to 8-bits!

(c) Execute: add W2,W2,W2



| | Before | After | |
|---|---|---|---|
| W0 | 0x1AF3 | 0x1AF3 | |
| W1 | 0x8B1A | 0x8B1A | |
| W2 | 0x64DE | 0xC9BC | Modified |
| W3 | 0xFB90 | 0xFB90 | |

$$\begin{array}{r} \texttt{0x64DE} \\ +\,\texttt{0x64DE} \\ \hline \texttt{0xC9BC} \end{array}$$

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

# *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

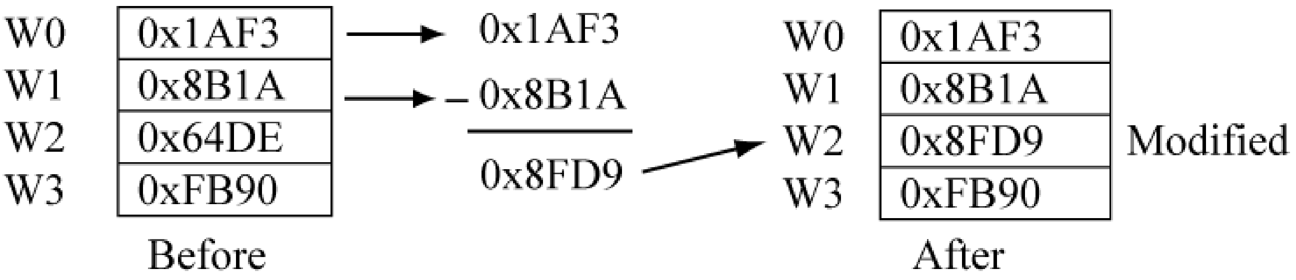
The same is not true for

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* Execution

(a) Execute: sub W0,W1,W2

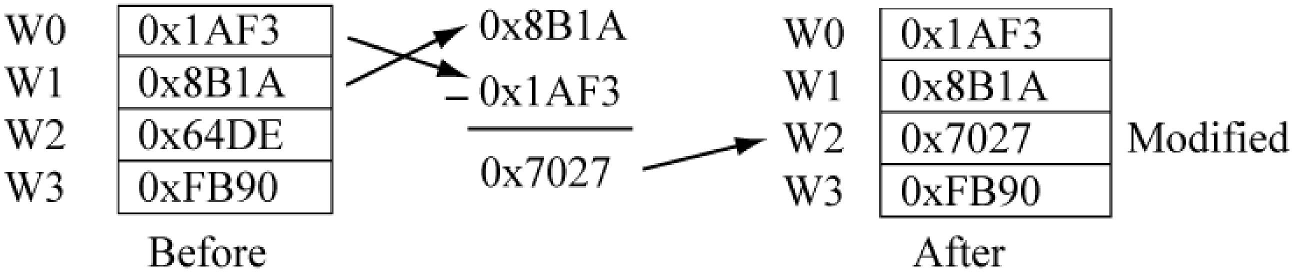


| Before | | After | |
|---|---|---|---|
| W0 | 0x1AF3 | W0 | 0x1AF3 |
| W1 | 0x8B1A | W1 | 0x8B1A |
| W2 | 0x64DE | W2 | 0x8FD9 Modified |
| W3 | 0xFB90 | W3 | 0xFB90 |

0x1AF3
– 0x8B1A
0x8FD9

---

(b) Execute: sub W1,W0,W2

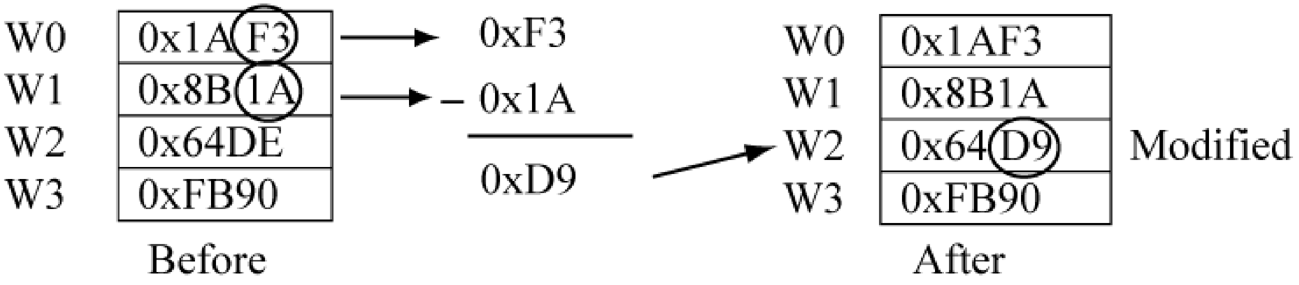


| Before | | After | |
|---|---|---|---|
| W0 | 0x1AF3 | W0 | 0x1AF3 |
| W1 | 0x8B1A | W1 | 0x8B1A |
| W2 | 0x64DE | W2 | 0x7027 Modified |
| W3 | 0xFB90 | W3 | 0xFB90 |

0x8B1A
– 0x1AF3
0x7027

---

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



| Before | | After | |
|---|---|---|---|
| W0 | 0x1AF3 | W0 | 0x1AF3 |
| W1 | 0x8B1A | W1 | 0x8B1A |
| W2 | 0x64DE | W2 | 0x64D9 Modified |
| W3 | 0xFB90 | W3 | 0xFB90 |

0xF3
– 0x1A
0xD9

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 |

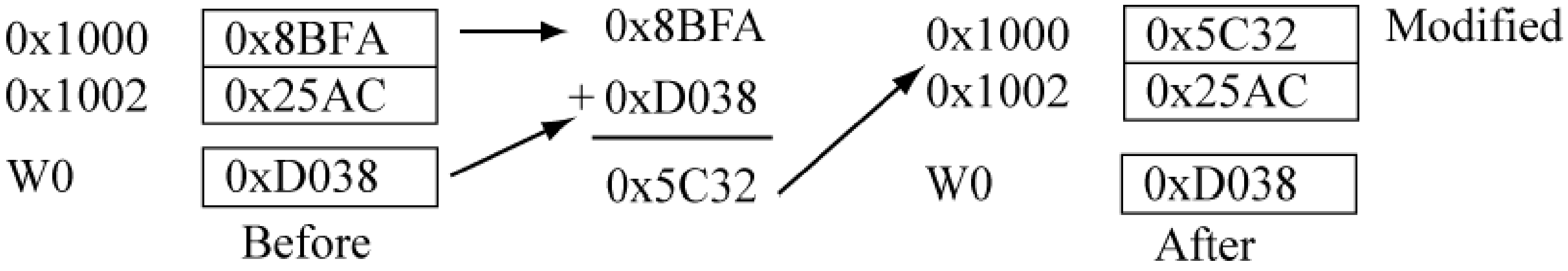
Can also be accomplished by:

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

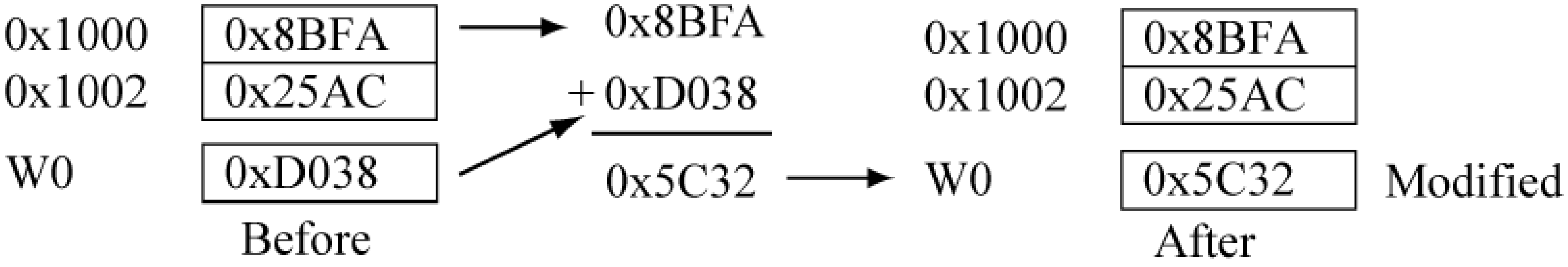
This takes three instructions and an extra register. However, in this class we are only concerned with the correctness of your assembly language, and not the efficiency. Use whatever approach you best understand!!!!!

# *ADD{.B} f {,WREG}* Execution

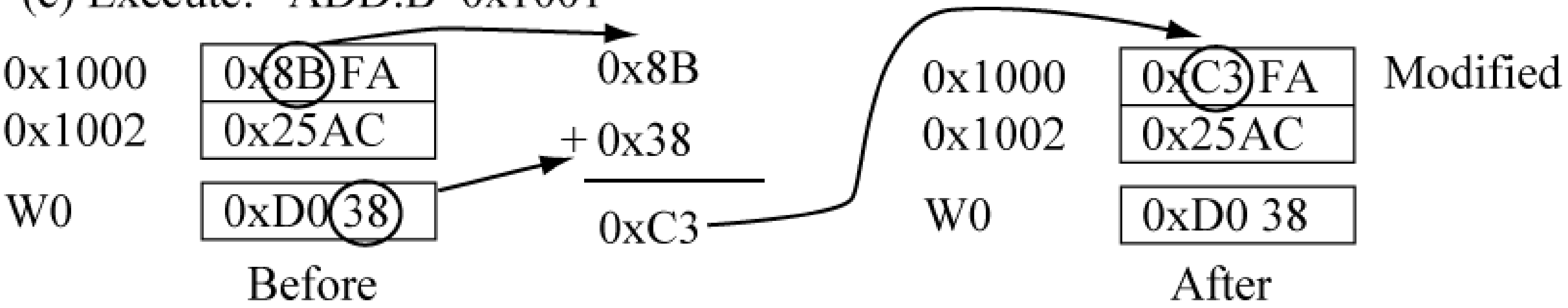
(a) Execute: ADD 0x1000



(b) Execute: ADD 0x1000, WREG



(c) Execute: ADD.B 0x1001



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

# *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:

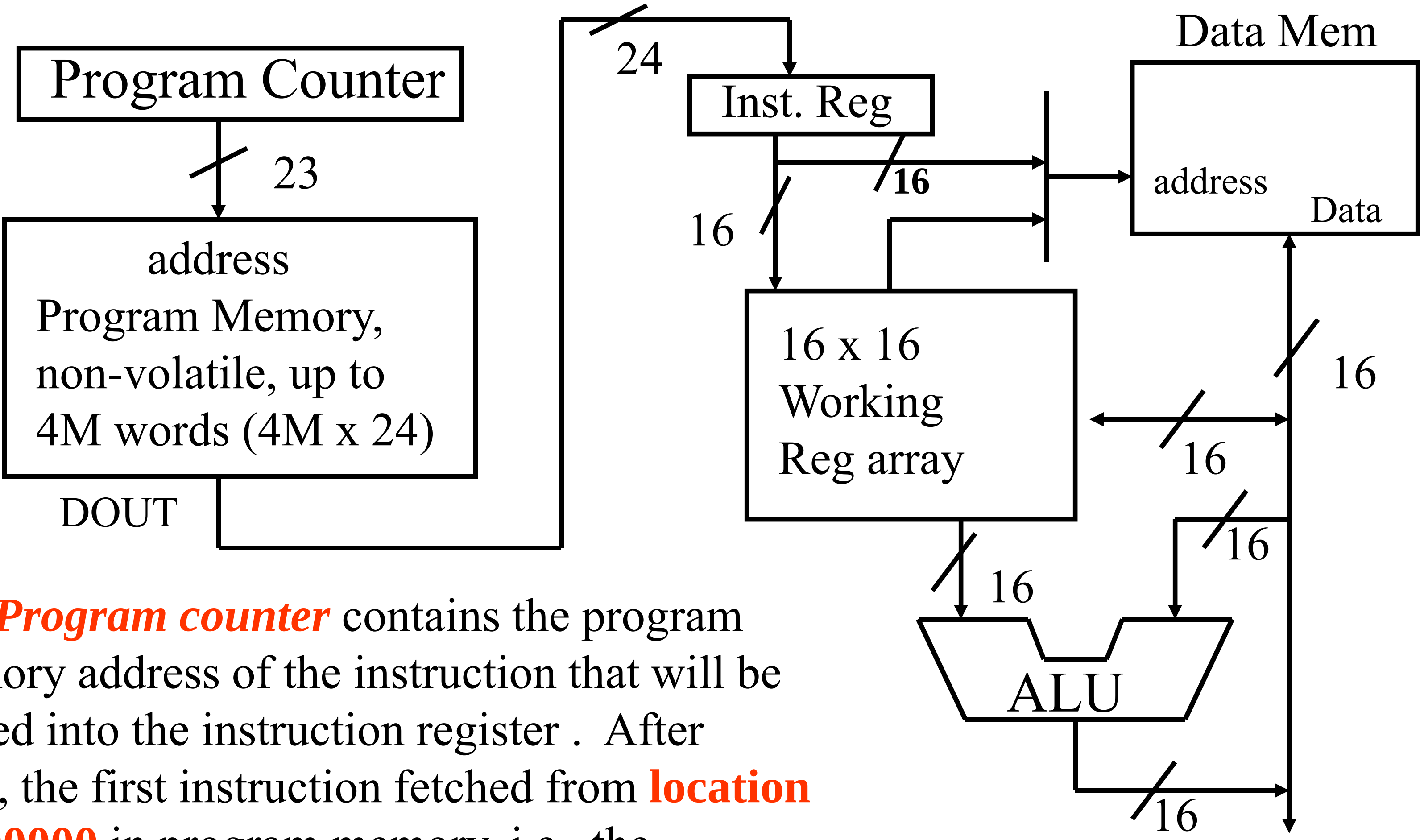
DEC W2, W4 (W2) – 1 → W4

DEC.B W3, W3 (W3.lsb) – 1 → W3.lsb

DEC 0x1000 (0x1000) – 1 → 0x1000

DEC.B 0x1001,WREG (0x1001) – 1 → WREG.lsb

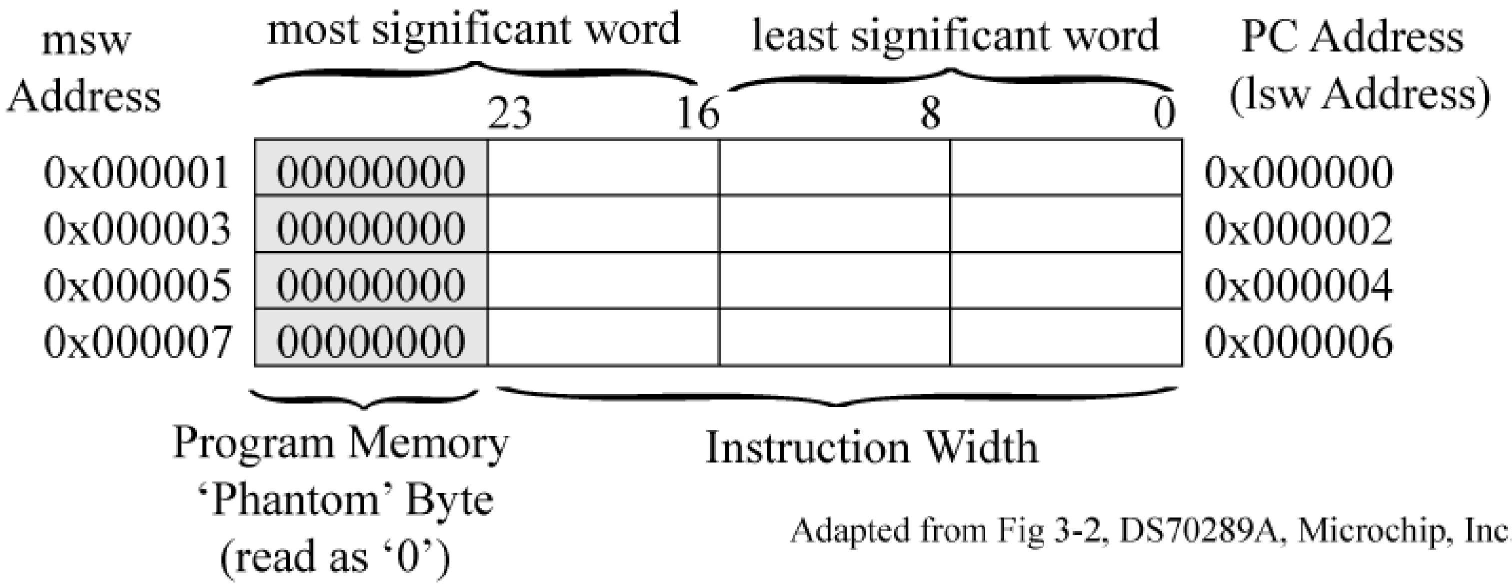
# How is the instruction register loaded?



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.**

17 x 17 Multiplier not shown

# Program Memory Organization



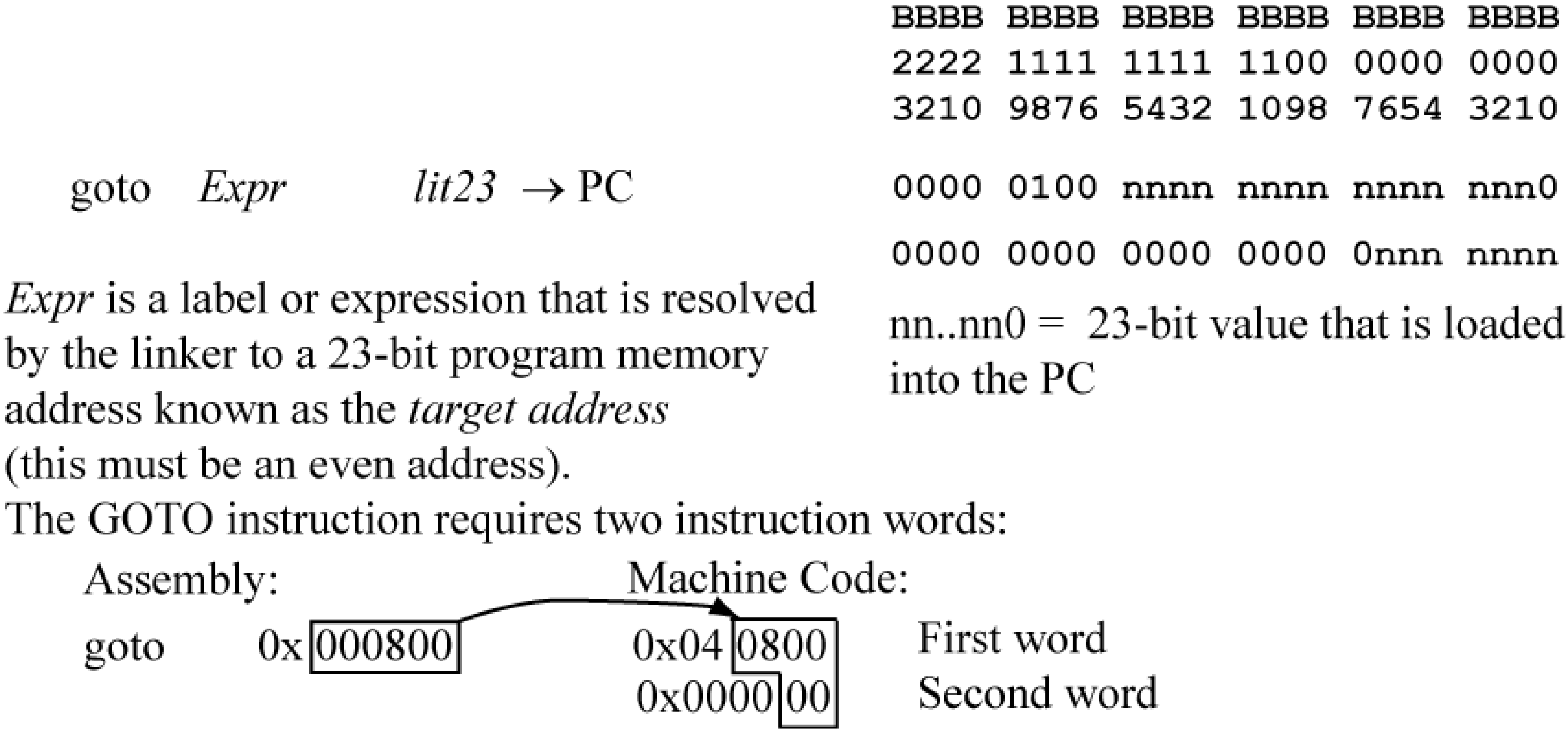
Adapted from Fig 3-2, DS70289A, Microchip, Inc.

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
```

nn..nn0 = 23-bit value that is loaded into the 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:                 Machine Code:
goto    0x000800          0x040800      First word
                          0x000000      Second word
```

A GOTO instruction is an unconditional jump.

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

# Valid addressing modes.

What are valid addressing modes for instructions?

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

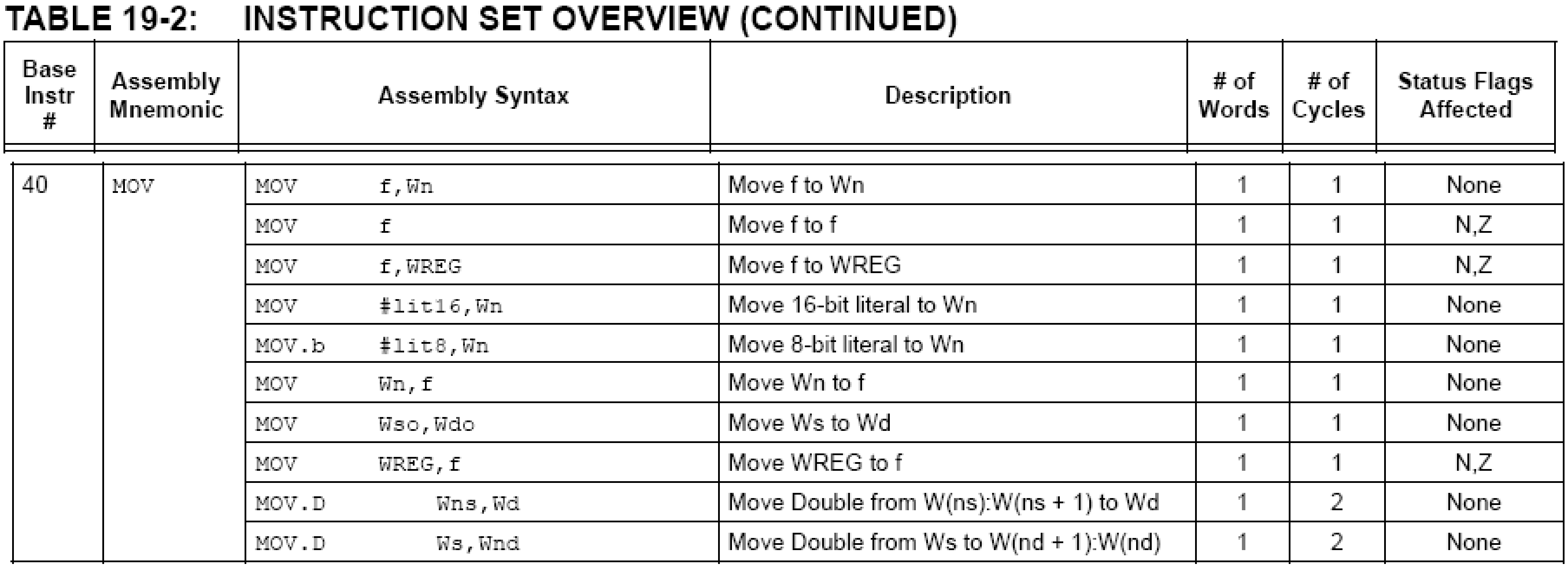


**TABLE 19-2: INSTRUCTION SET OVERVIEW (CONTINUED)**

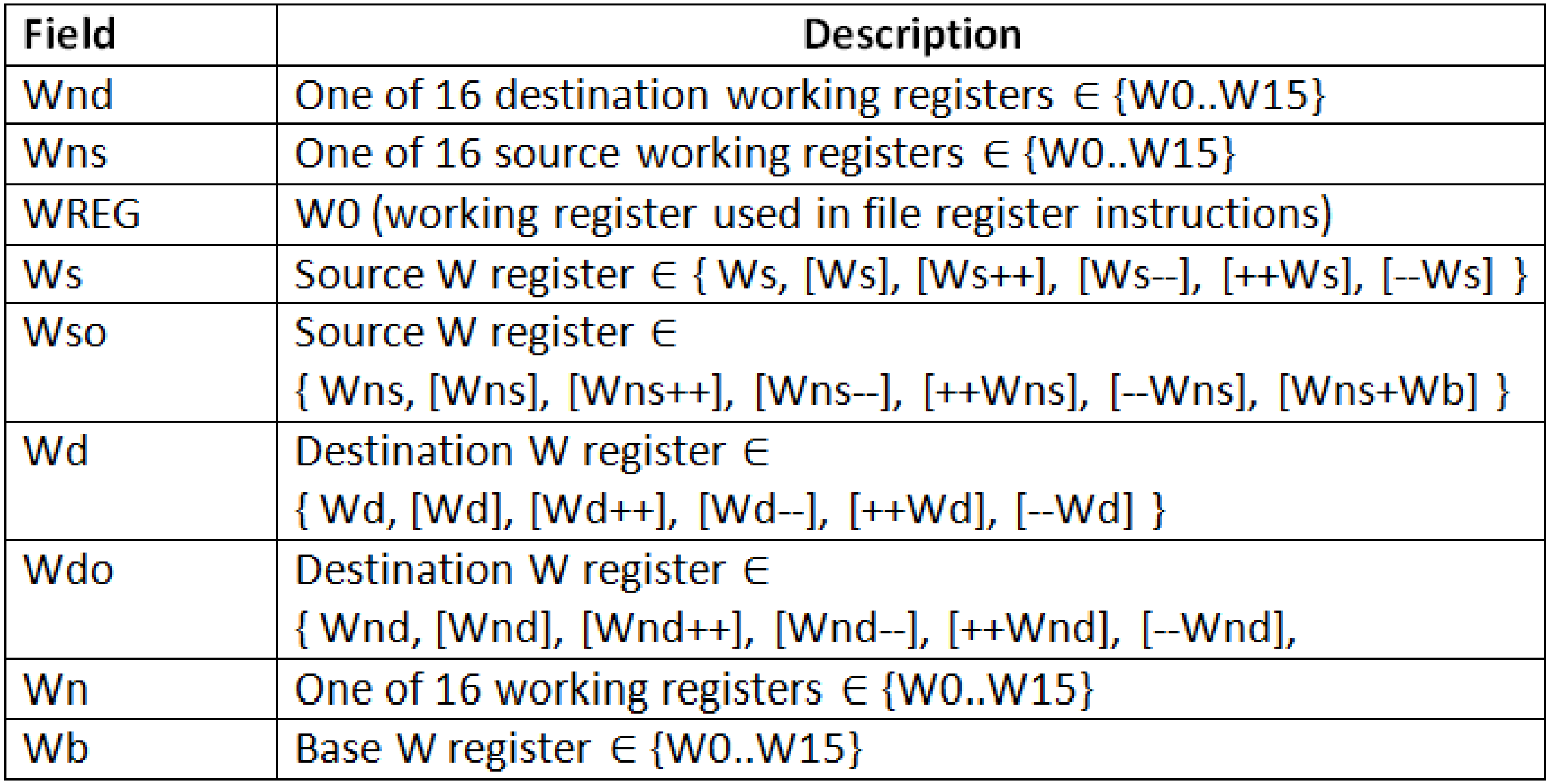
| Base Instr # | Assembly Mnemonic | Assembly Syntax | Description | # of Words | # of Cycles | Status Flags Affected |
|---|---|---|---|---|---|---|
| 40 | `MOV` | `MOV f,Wn` | Move f to Wn | 1 | 1 | None |
| | | `MOV f` | Move f to f | 1 | 1 | N,Z |
| | | `MOV f,WREG` | Move f to WREG | 1 | 1 | N,Z |
| | | `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”.

# What does ‘Wso’, ‘Wsd’, ‘Wn’ etc. mean?

MOV  Wso, Wdo

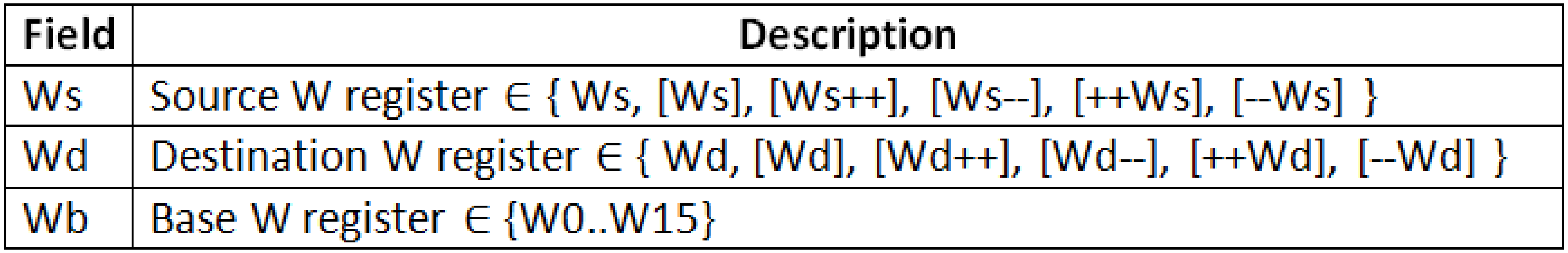
Table 19-1: Symbols used in opcode descriptions (partial list)



| **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 ∈<br>{ Wns, [Wns], [Wns++], [Wns--], [++Wns], [--Wns], [Wns+Wb] } |
| Wd | Destination W register ∈<br>{ Wd, [Wd], [Wd++], [Wd--], [++Wd], [--Wd] } |
| Wdo | Destination W register ∈<br>{ 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!
```

# 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

# A Simple Program

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

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

# 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"
```

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

```
.global __reset
```

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

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

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.

# 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**

```
#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. ← 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.

| | 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 |
| Total | 12 |

# What if we used 16-bit variables instead of 8-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
```

A uint16 variable is 16 bits (1 byte)

```
 .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
```

Previous approach took 9 instructions, this one took 8 instructions. In this case, this approach is more efficient!

```
;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
```

# 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 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:

| 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) |

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

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

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