

# Chapter 4

## PIC18 Assembly Language Programming

# 4.1 General Assembly Language Programming

# Outline

- Machine vs. assembly language
- Structure of assembly language
- The program counter and program memory in PIC
- Fetching and execution in PIC
- Some example directives: ORG, END, EQU and SET

# Machine and Assembly Language

- *Machine Language (opcode)*
  - A sequence of 0s and 1s that can be executed by the processor
  - Hard to understand, program, and debug for human being
- *Assembly Language (Source Code)*
  - Defined by assembly instructions
  - Assembly programs must be translated by an *assembler* before it can be executed
  - referred to as *low-level language* because the program needs to know the detailed interaction with the CPU

# Machine and Assembly Language

- e.g., in PIC

## Machine Code (Opcode)

## Assembly Code

---

0E32

movlw 0x32

6E05

movwf 0x05, A

0EDF

movlw 0xDF

2605

addwf 0x05, F, A

0E34

movlw 0x34

6E06

movwf 0x06, A

0E57

movlw 0x57

2206

addwfc 0x06, F, A

# Two Types of Assembly Statements

- *Directives (pseudo-instruction)*
  - Used to control the assembler
  - Do not generate machine code
  - e.g., ORG 0x0000 and END
- *Assembly language instructions*
  - Do generate machine code to perform various operations
  - e.g., all lines except the comments and directives

```
ORG 0x0000
goto Main

;-----
;Start of main program

Main:    movlw 0x32
         movwf 0x05, A
         movlw 0xDF
         addwf 0x05, F, A
         movlw 0x34
         movwf 0x06, A
         movlw 0x57
         addwfc 0x06, F, A

;*****
;End of program

END
```

# .lst file

```
000000      00018
000000      00019                                ORG      0x0000
000000 EF02 F000      00020                                goto    Main
000000      00021
000000      00022 ;-----
000000      00023 ;Start of main program
000000      00024
000004 0E32      00025 Main:    movlw   0x32
000006 6E05      00026                movwf   0x05, A
000008 0EDF      00027                movlw   0xDF
00000A 2605      00028                addwf   0x05, F, A
00000C 0E34      00029                movlw   0x34
00000E 6E06      00030                movwf   0x06, A
000010 0E57      00031                movlw   0x57
000012 2206      00032                addwfc  0x06, F, A
000012      00033
000012      00034 ;*****
000012      00035 ;End of program
000012      00036                                END
```

# 4 Elements of an Assembly Language Statement

## 1. Label (Optional)

- Must start in column 1
- e.g., `loop: addwf 0x20, W, A;`

## 2. Mnemonics

- Could either be an *assembly instruction mnemonic* or an *assembler directive*
- Examples of Mnemonics:
  - `org 0x0000`
  - `loop: addwf 0x20, W, A`



# 4 Elements of an Assembly Language Statement

## 3. Operands (Optional)

- `org 0x0000`
- `loop: addwf 0x20, W, A`

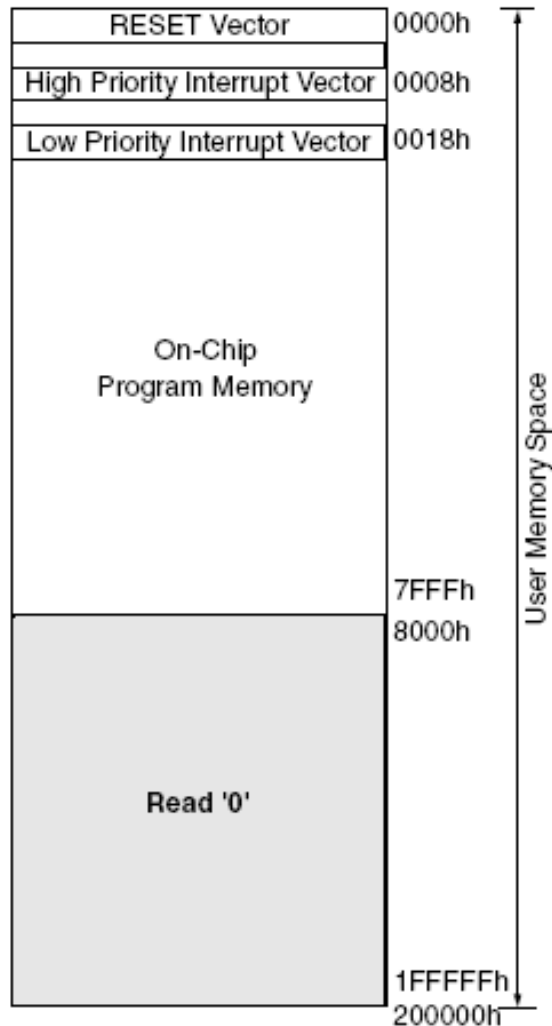
## 4. Comment (Optional)

- `loop: addwf 0x20, W, A ; add contents of WREG and file register`
- `; the whole line is comment`

# Program Memory

- Used primarily in storing assembly instructions, although it can be used to store fixed data (lookup table later)
- Separate from data memory
- 21-bit address bus, 16-bit data bus
- Address up to  $2^{21}=2\text{M}$  bytes of memory
- Not all memory locations are implemented

# Program Memory



- Each PIC18 member has a 21-bit address bus.
- Can address up to  $2^{21}=2\text{M}$  bytes program memory space.
- Implemented as Flash Memory (non-volatile)
- PIC18F452 implements only 32768 bytes, capable of storing 16384 instructions (most instructions are 2 bytes).

# Placement of Instructions in Program Memory

- Most instructions are 2 bytes. A few instructions are 4 bytes.

Program memory address	Opcode
000000	EF02 F000
000004	0E32
000006	6E05
000008	0EDF
00000A	2605
00000C	0E34
00000E	6E06
000010	0E57
000012	2206

```

00019      ORG                0x0000
00020      goto      Main
00021
00022      ;-----
00023      ;Start of main program
00024
00025      Main:
00026      movlw    0x32
00027      movwf    0x05, A
00028      movlw    0xDF
00029      addwf    0x05, F, A
00030      movlw    0x34
00031      movwf    0x06, A
00032      movlw    0x57
00033      addwfc   0x06, F, A
00034      ;*****
00035      ;End of program
00036
                                END

```

# Program Counter (PC)

- Used by the CPU to point to the address of the instruction being fetched.
- 21-bit stored as 3 bytes PCL, PCH, PCU.
- Since most instructions are 2 bytes, PC increments by 2 in each instruction cycle.

# Fetching and Execution in PIC18

- The instruction *fetch* stage gets the next instruction machine code from program memory.
- The *execution* stage does whatever the machine code calls for.
- *Execution*, which involves interaction with the data memory, does not interfere with *fetching* instruction from the program memory.

# PIC18 Pipelining

- Instead of taking two instruction cycles to first *fetch* and *execute* an instruction, both can be accomplished in one instruction cycle.
- This mechanism is called *pipelining*.

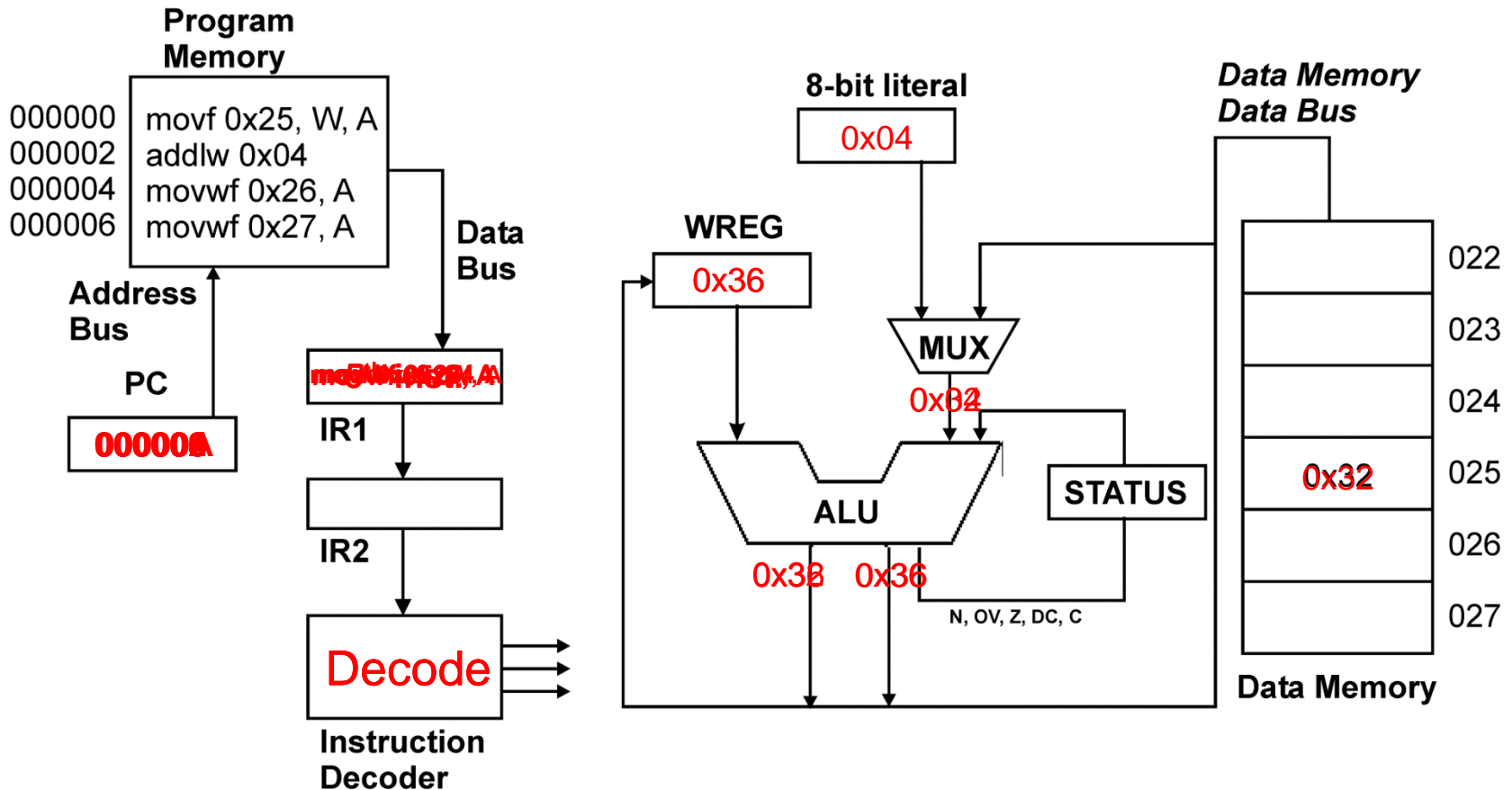
# Fetching and Execution Cycles

- Fetching Cycle
  - Increment PC
  - Fetch instruction into the instruction register (IR)
- Execution Cycle
  - Decode instruction
  - Read operands from data memory
  - Perform Arithmetic/Logic operation
  - Write the result to the destination.



# Fetch-And-Execute: A Complete View

Cycle 0	Cycle 1	Cycle 2	Cycle 3	Cycle 4
Fetch 1	Fetch 2	Fetch 3	Fetch 4	<b>Fetch 5</b>
	Execute 1	Execute 2	Execute 3	Execute 4



# Some important directives: EQU, SET

- EQU – associates a constant number with an address label.

- e.g., `COUNT equ 0x25`

.....

`movlw COUNT; [WREG] = 0x25`

- SET – identical to EQU, but value assigned by SET can be reassigned later.

- e.g., `COUNT set 0x00`

.....

`COUNT set 0x25`

.....

`movlw COUNT; [WREG] = 0x25`

- Note that `[RegA]` represents the value stored in the register `RegA`.

# EQU and SET Directives

**e.g., Move 22H into two file registers with addresses 0x05 and 0x06, add the contents of all three registers and put the result in WREG:**

## Without EQU

```
movlw 0x22
movwf 0x05, A
movwf 0x06, A
addwf 0x05, W, A
addwf 0x06, W, A
```

## With EQU

```
FirstReg EQU 0x05
SecReg    EQU 0x06
movlw 0x22
movwf FirstReg, A
movwf SecReg, A
addwf FirstReg, W, A
addwf SecReg, W, A
```

# CBLOCK Directive

- Defines a list of named constants.
- Format: `cblock <num>`  
    `<constant label> [:<inc>]`  
    `endc`
- e.g. 1, `cblock 0x50`  
    `test1, test2, test3, test4`  
    `endc`
- Values Assigned:  
    `test1 = 0x50, test2 = 0x51,`  
    `test3 = 0x52, test4 = 0x53.`

# You should be able to ...

- Describe the difference between machine code and assembly code
- Describe the difference between directives and assembly language instructions
- List the 4 elements of an assembly language statement
- Explain the program memory architecture in PIC18
- Examine program in the program memory window of MPLAB
- Describe the role of the program counter
- Describe the pipelining fetching and execution mechanism in PIC18

## 4.2 Unsigned and Signed Arithmetic, Logic Instructions

# Outline

- Subtraction of unsigned number
- Signed number arithmetic
- BCD addition
- Logic and compare instructions

# Subtraction

- Subtraction uses 2's complement
- In PIC18, four instructions are available for subtraction:

## Instructions

**sublw K**

**subwf fileReg, d, a**

**subwfb fileReg, d, a**

**subfwb fileReg, d, a**

## Function

**[WREG] = K – [WREG]**

**[fileReg] – [WREG]**

**[fileReg] – [WREG]  
with Borrow**

**[WREG] – [fileReg] with  
Borrow**




# Subtraction for unsigned numbers

- e.g., `movlw 0x23`

`sublw 0x3F`

3F	0011 1111	0011 1111
- 23	0010 0011	+1101 1101 (2's complement)
<hr/>		<hr/>
1C		1 0001 1100



- Or a quicker way to get 2's complement:
  - Subtract F from the most (MSN) and least significant nibble (LSN):
  - MSN:  $F - 2 = D$ ; LSN:  $F - 3 = C$ . Result is DC.
  - $DC + 1 = DD$ , equivalent to this
- $DC = 1, C = 1, Z = 0$
- Rule:  $C = 1, \text{Borrow} = 0$ .

# Subtraction for unsigned numbers

- How about this: 4C-6E?

- i.e., `movlw 0x6E`

`sublw 0x4C`

$$\begin{array}{r} \mathbf{4C} \quad \mathbf{0100\ 1100} \\ - \mathbf{6E} \quad \mathbf{+1001\ 0010\ (2's\ comp)} \\ \hline \mathbf{DE} \quad \mathbf{1101\ 1110} \end{array}$$

- In unsigned representation, we would not get a valid answer (which should be –ve). So how should we interpret our answer?
- You have a borrow (C=0).
  - i.e.,  $14C - 6E = DE$

# Subtraction using subwf

- e.g., Compute 4C – 6E by using the subwf instruction.
- Note subwf = sub WREG from f  
= [f] – [WREG]

```
MyReg equ 0x20
movlw 0x4C
movwf MyReg, A
movlw 0x6E
subwf MyReg, W, A
```

<b>4C</b>	<b>0100 1100</b>
<b>- 6E</b>	<b>+1001 0010 (2's comp)</b>
<b>DE</b>	<b>1101 1110</b>

# Multi-byte Subtraction: subwfb

- Subtract 2515 by 1FFF (hex)
- Do the LSB subtraction  
15 = 0001 0101  
-FF -> 2's complement = 01
- Note C = 0, Borrow = 1

Expected Result:

$$\begin{array}{r} 2515 \\ - 1FFF \\ \hline 0516 \end{array}$$

- Do the MSB subtraction  
25 = 0010 0101  
-1F -> 2's complement = E1
- MSB Result = 06
- Since we Borrow = 1 in LSB subtraction
- MSB Result = 06 - 1 = 05
- Note C = 1, Borrow = 0
- Final Result = 05 16

# Multi-byte Subtraction: subwfb

- 2515 – 1FFF

## Assembly Code

```
movlw 0xFF
sublw 0x15
movwf FirstReg, A
movlw 0x25
movwf SecondReg, A
movlw 0x1F
subwfb SecondReg, F, A
```

## Pseudocode

```
[WREG] = 15-FF
[FirstReg] = [WREG] (Borrow=1)
[SecondReg] = 25-1F-Borrow
```

# Representation of Signed Number

- Signed 8-bit representation:
  - D7 (MSB) represents sign : 1 if negative, 0 if positive
  - D0-D6 represent the magnitude of the number.
- Positive number
  - same as the unsigned number representation
  - ranges from 0 to  $2^7-1$  because 7 bits are used
- Negative number: 2's complement
  - Write the magnitude of the number as a unsigned 8-bit number
  - Invert each bit
  - Add 1 to it

# 8-bit Signed Number Representation

Decimal	Binary	Hex
+127	0111 1111	7F
.....	.....	.....
+2	0000 0010	02
+1	0000 0001	01
0	0000 0000	00
-1	1111 1111	FF
-2	1111 1110	FE
....	....	....
-127	1000 0001	81
-128	1000 0000	80

# Example

- e.g., `movlw 0x82`  
`addlw 0x22`

-7E	1000 0010	(2's comp) 0111 1110
+ 22	0010 0010	
<hr/>	<hr/>	
-5C	1010 0100	(2's comp) 0101 1100

**Flags in STATUS register will be set as:**

- N = 1 (D7 = 1)
- OV = 0 (No overflow. Result is correct.)
- Rule: Negative number + Positive number → OV = 0



# Signed and unsigned additions

- **The microcontroller does not know whether you are performing a signed or an unsigned addition. The result is the same either way.**
- You can interpret an addition operation as a signed or unsigned operation.
- e.g., 82+22 always gives a result A4
  - Unsigned:  $82 + 22 = A4$
  - Signed:  $-7E(\text{represented as } 82) + 22 = -5C$  (represented as A4). OV and N only make sense when an operation is interpreted as a signed operation.

# Overflow in signed operations

- e.g., `movlw D' 96'`  
`addlw D' 70'`

96	0110 0000
<u>+70</u>	<u>0100 0110</u>
-90	1010 0110

**Flags in STATUS register will be set as:**

- N = 1 (D7 = 1)
- OV = 1
- Sum = -90

# Answer the following questions:

Show the status of the N and OV flag bits for the following code:

(a) `movlw 0x67`

`addlw 0x99`

(b) `movlw 0x44`

`addlw 0x60`

(c) `movlw 0x56`

`addlw 0x8F`

(d) `movlw 0xFF`

`addlw 0xF2`

# Solutions

(a)

$$\begin{array}{r} 67 \\ 99 \\ \hline 00 \end{array}$$

$$DC = 1$$

$$C = 1$$

$$Z = 1$$

$$N = 0$$

$$OV = 0$$

$\therefore$  pos + neg

(c)

$$\begin{array}{r} 56 \\ 8F \\ \hline E5 \end{array}$$

$$DC = 1$$

$$C = 0$$

$$Z = 0$$

$$N = 1$$

$$OV = 0$$

$\therefore$  pos + neg

(b)

$$\begin{array}{r} 44 \\ 60 \\ \hline A4 \end{array}$$

$$DC = 0$$

$$C = 0$$

$$Z = 0$$

$$N = 1$$

$$OV = 1$$

$\therefore$  pos + pos = neg

(d)

$$\begin{array}{r} FF \\ F2 \\ \hline F1 \end{array}$$

$$DC = 1$$

$$C = 1$$

$$Z = 0$$

$$N = 1$$

$$OV = 0$$

$\therefore$  neg + neg = neg

# Binary Coded Decimal (BCD) System

- All additions are performed in binary format in PIC18 MCU
- Decimal numbers can be encoded in binary-coded decimal (BCD) format

<u>BCD</u>	<u>decimal</u>	<u>BCD</u>	<u>decimal</u>
0000	0	0101	5
0001	1	0110	6
0010	2	0111	7
0011	3	1000	8
0100	4	1001	9

# BCD Addition

- Binary addition of BCD numbers would not be correct if one of the following occurs:

1. If a sum digit is greater than 9

e.g.,

24	36	55
<u>+67</u>	<u>+47</u>	<u>+77</u>
8B	7D	CC

2. If a sum digit has a carry of 1 to the higher digit

e.g.,

29
<u>+47</u>
70

# BCD Addition

- Solution: Add 0x6 to these two types of sum digits.

e.g.,

24	36	55	29
<u>+67</u>	<u>+47</u>	<u>+77</u>	<u>+47</u>
8B	7D	CC	70
<u>+ 6</u>	<u>+ 6</u>	<u>+66</u>	<u>+ 6</u>
91	83	132	76

- `daw` would do this for you.

```
movf    0x10, W, A
addwf   0x14, W, A
daw
```

# daw Instruction

- daw instruction adds 0x06 to the upper or lower byte if necessary.
- **daw instruction works only on WREG**
- Lower nibble: If the lower nibble (4 bits) is greater than 9 or if DC = 1, add 0x06 to the lower nibble.
- Upper nibble: If the higher nibble (4 bits) is greater than 9 or if C = 1, add 0x06 to the upper nibble.

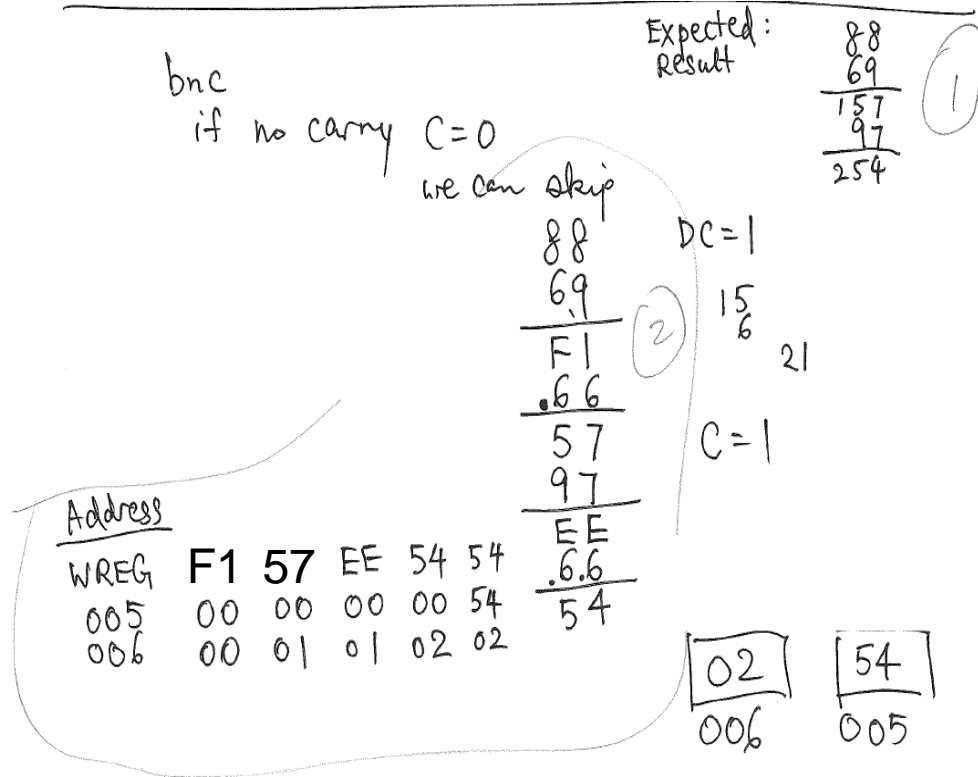


# Example: BCD Addition of 3 numbers

- e.g., Find the sum of the values 0x88, 0x69 and 0x97. Put the sum in file register 0x005 and 0x006.

```

H_byte equ 0x06
L_byte equ 0x05
    org    0x00
    clrf   H_byte, A
    clrf   L_byte, A
    movlw  0x88
    addlw  0x69
    daw
    bnc     N_1
    incf    H_byte, F, A
N_1: addlw  0x97
    daw
    bnc     Over
    incf    H_byte, F, A
Over: movwf L_byte, A
    end
    
```



# Multiplication

- PIC18 has two instructions for 8-bit multiplication: `mulwf f` and `mullw k`.
- The products are stored in the `PRODH:PRODL` register pair.
- The following instruction sequence performs 8-bit multiplication operation:

```
movf      0x10, W, A  
mulwf     0x11, A
```

# Logic Instructions

- PIC18 provides instructions to perform logic operations such as AND, OR, Exclusive-OR and complement.

## Instructions

andlw K

andwf fileReg, d, a

iorlw K

iorwf fileReg, d, a

xorlw K

xorwf fileReg, d, a

comf fileReg, d, a

negf fileReg, a

## Function

**AND K with [WREG]**

**AND [WREG] with [fileReg]**

**Or K with [WREG]**

**Or [WREG] with [fileReg]**

**Exclusive Or K with [WREG]**

**Exclusive Or [WREG] with [fileReg]**

**Complement [fileReg]**

**Produce 2's complement of [fileReg]**

# Compare instructions

- Compare instructions compare a value in the file register with the contents of the WREG register.
- Three instructions that compare [fileReg] with [WREG]:

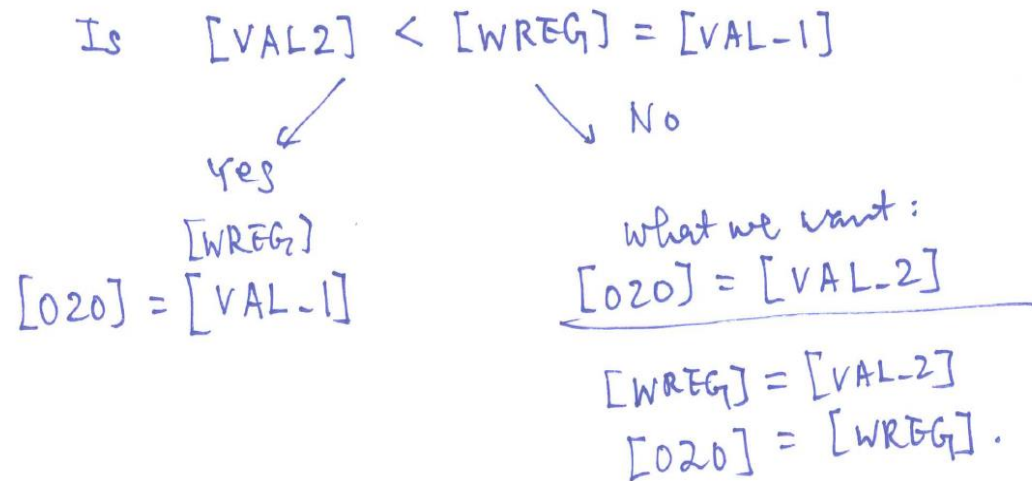
<u>Instructions</u>	<u>Function</u>
cpfsgt fileReg, a	<b>Skip if [fileReg] &gt; [WREG]</b>
cpfseq fileReg, a	<b>Skip if [fileReg] = [WREG]</b>
cpfslt fileReg, a	<b>Skip if [fileReg] &lt; [WREG]</b>

# Example: cpfslt

- e.g. Write a program to find the greater of the VAL\_1 and VAL\_2 registers and place it in file register location 0x020.

```
MyReg equ 0x20  
VAL_1 equ 0x00  
VAL_2 equ 0x01
```

```
movf VAL_1, W, A;  
[WREG] = [VAL_1]  
cpfslt VAL_2, A;  
; Is [VAL_2] < [WREG] =  
[VAL_1]  
movf VAL_2, W, A;  
[WREG] = [VAL_2]  
movwf MyReg, A;  
[020] = [WREG]
```



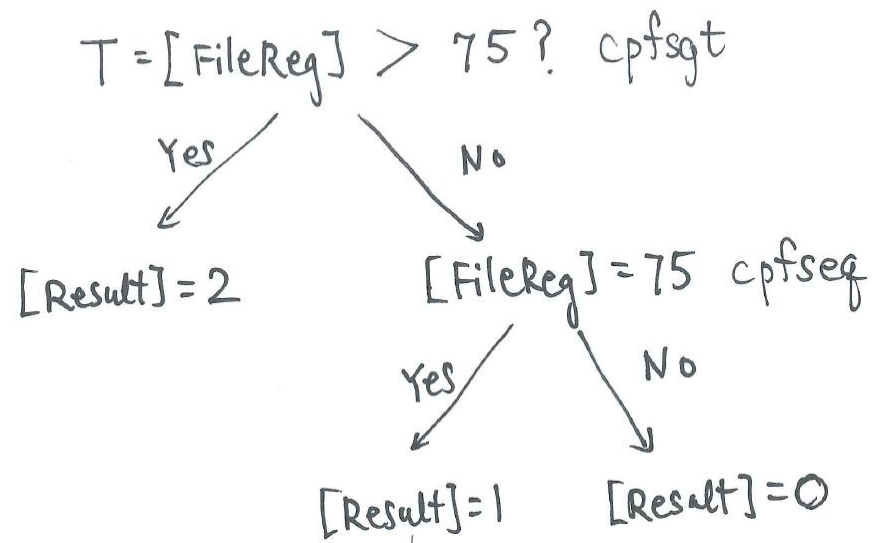
# Example

- e.g., Suppose a value  $T$  is stored in the register 0x020. Put a value to the register 0x021 according to the following scheme:
  - If  $T = 75$ , then  $[0x021] = 1$
  - If  $T > 75$ , then  $[0x021] = 2$
  - If  $T < 75$ , then  $[0x021] = 0$

# Example

```
FileReg equ 0x20
Result  equ 0x21
```

```
Main: movlw D'75'
      cpfsgt FileReg, A
      bra LEQ
GT:   movlw D'2'
      movwf Result, A
      bra Over
LEQ:  movlw D'75'
      cpfseq FileReg, A
      bra LT
EQ:   movlw D'1'
      movwf Result, A
      bra Over
LT:   movlw D'0'
      movwf Result, A
      bra Over
Over: .....
```



## 4.3 Looping and Branching



# Looping in PIC

- Loop – repeating a sequence of instructions a certain number of times.
- e.g., To add 3 to [WREG] five times, I could do:

```
movlw 0x00  
addlw 0x03  
addlw 0x03  
addlw 0x03  
addlw 0x03  
addlw 0x03
```

- Disadvantage: Too much code space is needed if repeating 100 times
- Use looping

# Three steps in writing a loop

- Step 1: Initialize variables
  - Count: specifies how many times you want to repeat the loop. (e.g., [Count] = d'5' in previous example)
  - Other variables depending on application.
- Step 2: Identify statement(s) to repeat
  - In the previous example, it would be:  
`addlw 0x03`
- Step 3: Determine whether to stop looping
  - Strategy 1: Use `decfsz Count` to decrement Count after one iteration (repetition) of the loop, and check whether Count is 0. Two possibilities:
    - If Count = 0 (i.e., finished the desire number of iterations), exit loop.
    - If Count  $\neq$  0, branch back to Step 2.

# decfsz instruction

- decfsz stands for **DEC**rement **F**ile register and **S**kip if **Z**ero.
- decfsz is a conditional skipping instruction, which skips the next instruction if a condition is satisfied (here if result is 0 *after* the decrement).
- Format: decfsz f, d, a  
where f is the 8-bit address of the file register and d indicates the destination of decremented value: W – WREG, F – File Register

# Looping using decfsz instruction

- e.g., Write a program to (a) initialize WREG to 0 (b) add 0x05 ten times to WREG and put the result in the PRODL register

```
count equ 0x00; Step 1 (Red)
```

```
    movlw d'10'
```

```
    movwf count, A
```

```
    movlw 0x00
```

```
AddMore: addlw 0x05; Step 2 (Green)
```

```
    decfsz count, F, A; Step 3 (Blue)
```

```
    bra AddMore
```

```
    movwf PRODL
```

# Perform arithmetic operation using a loop

- Write a program to compute  $1 + 2 + \dots + 10$  and save the sum at the data memory address 0x000.
- We will work on the three steps together and write code.

# 3-Step Plan

- Step 1: Initialization
  - $[\text{Count}] = 10$
  - $[\text{Numi}] = 1$
  - $[\text{Sum}] = 0$
- Step 2: Statements to repeat
  - $[\text{Sum}] = [\text{Sum}] + [\text{Numi}]$
  - $[\text{Numi}] = [\text{Numi}] + 1$
- Step 3: decfsz Count
  - $[\text{Count}] = [\text{Count}] - 1$
  - Stop repetition if  $[\text{Count}] = 0$

# Code

```
        cblock 0x00
            Count
            Numi
            Sum
        endc

        org 0x000000
Initialization: movlw d'10'
               movwf Count, A
               movlw d'1'
               movwf Numi, A
               clrf Sum, A
Here:        movf Numi, W, A;           [WREG] = [Numi]
               addwf Sum, F, A;         [Sum]  = [Sum] + [WREG]
               incf Numi, F, A;         [Numi] = [Numi] + 1
               decfsz Count, F, A;
               ;[Count] = [Count] - 1; skip if [Count] = 0;
               bra Here
               bra $
               end
```

# Strategy 1 for Step 3 in Looping

```
decfsz count, F, A; Step 3 (Blue)  
bra AddMore
```

## More analysis:

- `decfsz` is a conditional skipping instruction that skips the branching instruction (i.e., `bra AddMore`) if `count = 0` *after* decrement (i.e., finished the number of iterations we want)
- `bra AddMore` is an unconditional branching instruction (i.e., if this instruction is executed, it always branches)



# Strategy 2 for Step 3 in Looping

- Use a conditional branching instruction called `bnz`.
- `bnz` stands for **B**ranch if the Z flag in the status register is zero (i.e., **N**ot **Z**ero)
  - easy to get confused, but note:
    - Result  $\neq 0$ , Z = 0  $\rightarrow$  Branch is taken
    - Result = 0, Z = 1  $\rightarrow$  Branch is not taken
- Whether the branch is taken is determined by whether a condition is satisfied (here, Z = 0)
- Format: `bnz label`
- e.g., Implement the loop in the last slide using `bnz`.

# Comparison between Strategy 1 & 2

- Strategy 1: decfsz

```
count equ 0x00  
movlw d'10'  
movwf count, A  
movlw 0x00
```

```
AddMore: addlw 0x05
```

```
decfsz count, F, A
```

```
goto AddMore
```

```
movwf PRODL
```

- Strategy 2: bnz

```
count equ 0x00  
movlw d'10'  
movwf count, A  
movlw 0x00
```

```
AddMore: addlw 0x05
```

```
decf count, F, A
```

```
bnz AddMore
```

```
movwf PRODL
```

# Other conditional branching instructions

- These instructions jumps when C and Z in the STATUS register are 1 or 0
  - `bnz`: jump if  $Z = 0$  (previously described)
  - `bz`: jump if  $Z = 1$
  - `bnc`: jump if  $C = 0$
  - `bc`: jump if  $C = 1$

# Example on bz

- e.g., Write a program to determine whether file register 0x30 contains a value other than 0x00. If not, put 0x00 in it.

```
fileReg equ 0x30
movf fileReg, F, A
bz Next
movlw 0x00
movwf fileReg
```

```
Next: ...
```

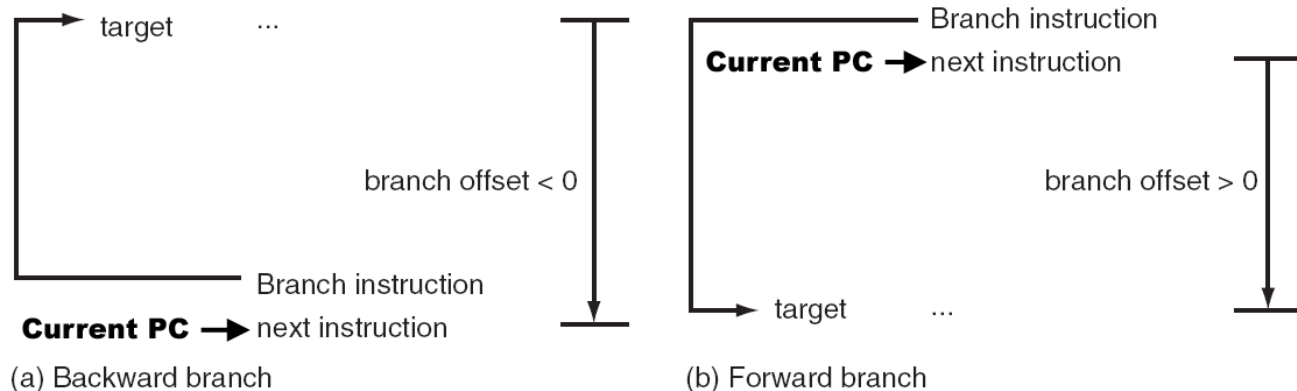
# Example on bnc

- e.g., Find the sum of the values 0x79, 0xF5 and 0xE2. Put the sum in file register 0x05 and 0x06.

```
H_byte equ 0x06
L_byte equ 0x05
        org 0x0000
        clrf H_byte, A
        clrf L_byte, A
        movlw 0x79
        addlw 0xF5
        bnc N_1
        incf H_byte, F, A
N_1:    addlw 0xE2
        bnc Over
        incf H_byte, F, A
Over:   movwf L_byte, A
        end
```

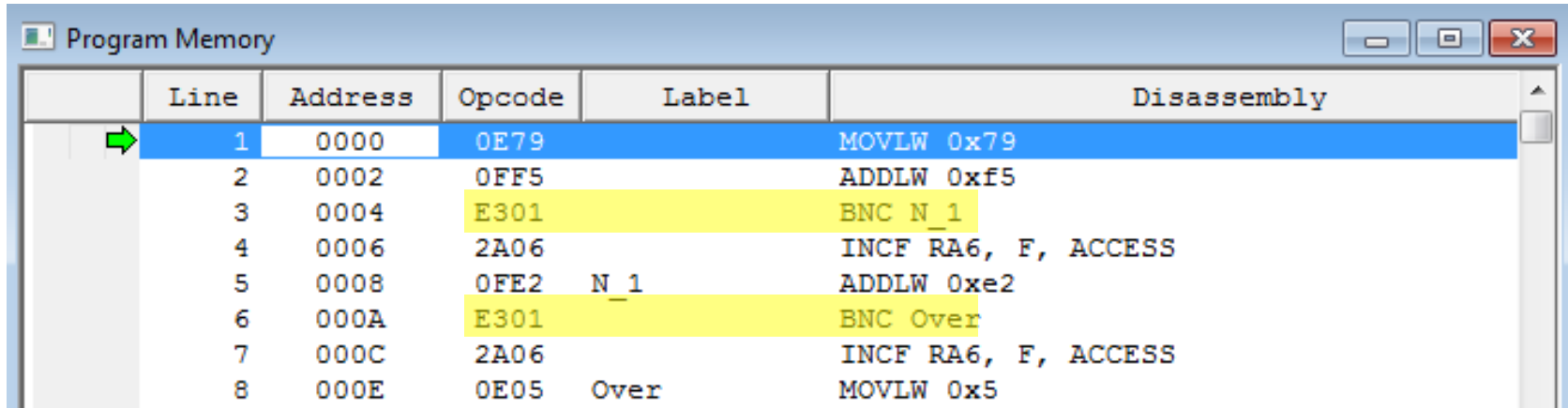
# How does a conditional branching instruction know where to jump to?

- The instruction `bnc label` is interpreted as `bnc n` by the assembler, where `n` is a signed 1-byte number within -128 to 127.
- `n` is the Relative Address of the branching destination with respect to the current PC. `n` is expressed in number of instructions (i.e., relative address =  $2n$  if expressed in address)



- Because the target must be within 256 instructions of the current PC, conditional branches are *short jumps*.

# Program Memory of the bnc Example



	Line	Address	Opcode	Label	Disassembly
➔	1	0000	0E79		MOVLW 0x79
	2	0002	0FF5		ADDLW 0xf5
	3	0004	E301		BNC N_1
	4	0006	2A06		INCF RA6, F, ACCESS
	5	0008	0FE2	N_1	ADDLW 0xe2
	6	000A	E301		BNC Over
	7	000C	2A06		INCF RA6, F, ACCESS
	8	000E	0E05	Over	MOVLW 0x5

- The label N\_1 is one instruction away (at 0008) from the current PC (0006) when branching occurs.
- Opcode = E301 → jump 1 instruction (or 2 bytes from 0006 to 0008) if branching occurs.

# How about `bra`?

- `bra`
  - allocated 11 bits for storing the Relative Address of the targeting instruction.
  - `bra` can jump forward for a max. of 1023 instructions and backward for a max. of 1024 instructions.

BRA	Unconditional Branch			
Syntax:	[ <i>label</i> ] BRA    n			
Operands:	-1024 ≤ n ≤ 1023			
Operation:	(PC) + 2 + 2n → PC			
Status Affected:	None			
Encoding:	1101	0nnn	nnnn	nnnn



# Exercise on bra address encoding

Calculate the relative addresses (marked by “???”) encoded in the opcode shown below.

```
0000    EF02                                goto    Main
0002    F000

;-----
;Start of main program

0004    0E4B    Main:    movlw D'75'
0006    6420                                cpfsgt FileReg, A
0008    D???                                bra    LEQ
000A    0E02    GT:      movlw D'2'
000C    6E21                                movwf Result, A
000E    D???                                bra    Over
0010    0E4B    LEQ:     movlw D'75'
0012    6220                                cpfseq FileReg, A
0014    D???                                bra    LT
0016    0E01    EQ:      movlw D'1'
0018    6E21                                movwf Result, A
001A    D???                                bra    Over
001C    0E00    LT:      movlw D'0'
001E    6E21                                movwf Result, A
0020    D???                                bra    Over
0022    D???    Over:    bra    Over
```

# Solutions

```

0000    EF02                goto    Main
0002    F000

```

```

;-----
;Start of main program

```

```

0004    0E4B  Main:  movlw D'75'
0006    6420                cplsgt FileReg, A
0008    D??? D003      bra LEQ ←  $R_{add} = \left( \frac{0010 - 000A}{2} \right) = \underline{3}$ 
000A    0E02  GT:    movlw D'2'
000C    6E21                movwf Result, A
000E    D??? D009      bra Over ←  $R_{add} = \left( \frac{0022 - 0010}{2} \right) = 9.$ 
0010    0E4B  LEQ:   movlw D'75'
0012    6220                cpfseq FileReg, A
0014    D??? D003      bra LT ←  $R_{add} = \left( \frac{001C - 0016}{2} \right) = 3$ 
0016    0E01  EQ:    movlw D'1'
0018    6E21                movwf Result, A
001A    D??? D003      bra Over ←  $R_{add} = \left( \frac{0022 - 001C}{2} \right) = 3$ 
001C    0E00  LT:    movlw D'0'
001E    6E21                movwf Result, A
0020    D??? D000      bra Over ←  $R_{add} = 0$ 
0022    D??? Over:   bra Over ←  $R_{add} = \left( \frac{0022 - 0024}{2} \right) = -1$ 

```

DTEF.

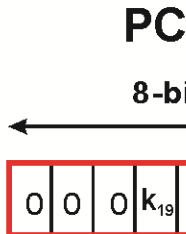
006 0000 0001  
 ↓ 2'comp  
 111 1111 1110  
 +  
 111 1111 1111  
 7 F F

# goto: Another unconditional branching instruction

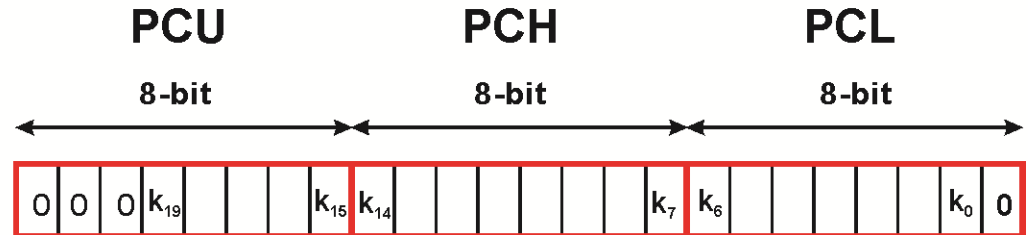
- `goto` specifies the Absolute Address of the branching destination and can jump to anywhere in the program memory.
- `goto` is a 4-byte (or 2-word) instruction.

GOTO	Unconditional Branch			
Syntax:	[ <i>label</i> ] GOTO k			
Operands:	$0 \leq k \leq 1048575$			
Operation:	$k \rightarrow \text{PC}<20:1>$			
Status Affected:	None			
Encoding:				
1st word ( $k<7:0>$ )	1110	1111	$k_7kkk$	$kkkk_0$
2nd word ( $k<19:8>$ )	1111	$k_{19}kkk$	$kkkk$	$kkkk_8$

## Example of a 4-byte instruction: goto

- Why 4-byte?
    - The machine code must contain the address of the destination in the program memory (21-bit)
  - Only the most significant 20 bits are included. Why?
    - Program memory are organized as 2-byte blocks. LSB must be 0.
- 
- The diagram shows a 21-bit PC field. The top 20 bits are labeled 'PC' and '8-b' (likely meaning 8-bit blocks). The bottom 1-bit is labeled 'k<sub>19</sub>'. The entire 21-bit field is enclosed in a red box. An arrow points to the left from the top of the field.

GOTO		Unconditional Branch			
Syntax:	[ <i>label</i> ] GOTO k				
Operands:	$0 \leq k \leq 1048575$				
Operation:	$k \rightarrow PC_{<20:1>}$				
Status Affected:	None				
Encoding:					
1st word ( $k < 7:0 >$ )	1110	1111	$k_7 kkk$	$kkkk_0$	
2nd word ( $k < 19:8 >$ )	1111	$k_{19} kkk$	$kkkk$	$kkkk_8$	



# Machine code of goto

- The `goto Main` line in our previous example forces PC to 0x000004, where the Main label is.
- The machine code is

**EF02**

**F000**

- Thus, the program goes to

**0000 0000 0000 0000 0010 0** in binary  
(i.e., 0x000004 in hex)

# Example for the `goto` instruction

- e.g., In our program, we want to branch to address `0x000020`. What is the machine code of the `goto` instruction?

# Comparison among 3 types of branching instructions

Conditional Branching	Unconditional Branching	
<b>bc, bnc, bz, bnz, bn, bnn, bov, bnov</b>	<b>bra</b>	<b>goto</b>
Branch according to the whether a flag in the STATUS register is raised.	Branch unconditionally	
The operand of these instructions specifies the address of the destination of the jump <i>relative to the position of the next instruction</i>	The operand of goto specifies the <i>absolute address of the destination</i>	
Able to jump -128 to +127 lines with respect to current instruction	Able to jump -1024 to +1023 lines with respect to current instruction	Able to jump to anywhere in the program memory

# Exercise on Relative/Absolute Addresses

Calculate the relative/absolute addresses (marked by “??”) encoded in the opcode shown in Lines 29, 37, 39 in the following program.

Program Memory Address	Machine Code	LINE	SOURCE
		00022	CBLOCK 0x20
		00023	Binary
		00024	Tens
		00025	Units
		00026	ENDC
		00027	
		00028	ORG 0x000000
000000	<u>EF?? F???</u>	00029	goto Main
		00030	ORG 0x000020
000020	0E4D	00031 Main:	movlw d'77'
000022	6E20	00032	movwf Binary, A
000024	6A21	00033 Bin_2_BCD:	clrf Tens, A
000026	6A22	00034	clrf Units, A
000028	5020	00035	movf Binary, W, A
00002A	0FF6	00036 Loop:	addlw -d'10'
00002C	<u>E3??</u>	00037	bnc Next
00002E	2A21	00038	incf Tens, F, A
000030	<u>D???</u>	00039	bra Loop
000032	0F0A	00040 Next:	addlw d'10'
000034	6E22	00041	movwf Units, A



# Solutions

$4 = 000\ 0000\ 0100$   
 $\downarrow 2' \text{comp}$   

$$\begin{array}{r} 111\ 1111\ 1011 \\ + \phantom{111\ 1111\ 1011} 1 \\ \hline 111\ 1111\ 1100 \\ \phantom{111\ 1111\ 1100} 9 \quad F \quad C \end{array}$$

Program Memory Address	Machine Code	LINE	SOURCE
		00022	CBLOCK 0x20
		00023	Binary
		00024	Tens
		00025	Units
		00026	ENDC
		00027	
000000	EF10 F000 <u>EF?? F???</u>	00028	ORG 0x000000
		00029	goto Main
		00030	ORG 0x000020
000020	0E4D	00031 Main:	movlw d'77'
000022	6E20	00032	movwf Binary, A
000024	6A21	00033 Bin_2_BCD:	clrf Tens, A
000026	6A22	00034	clrf Units, A
000028	5020	00035	movf Binary, W, A
00002A	0FF6	00036 Loop:	addlw -d'10'
00002C	<u>E3?? E302</u>	00037	bnc Next ←
00002E	<u>2A21</u>	00038	incf Tens, F, A
000030	<u>D???</u>	00039	bra Loop ←
000032	0F0A	00040 Next:	addlw d'10'
000034	6E22	00041	movwf Units, A

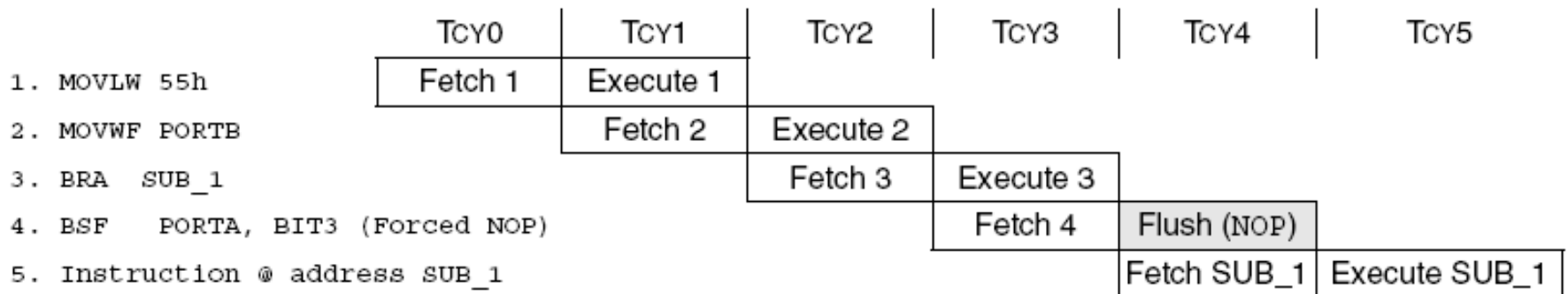
Dest  
 000020  
 0 - 0000 0000 0000 0000 0000 0000

$$R_{add} = \left( \frac{32 - 2E}{2} \right) = 2$$

$$R_{add} = \left( \frac{2A - 32}{2} \right) = -4$$

# Timing Requirement for branching/skipping instructions

- *Conditional/unconditional branching and conditional skipping* instructions change the program counter (PC).
- Since the fetch stage incorrectly gets the machine code for the next sequential instruction when branching, this machine code must be dumped.



# Timing requirement for unconditional branching instructions (`bra`, `goto`)

- Pre-fetched instruction must be dumped.
- Nothing can be executed in the second cycle
- *2 instruction cycles needed*

BRA	Unconditional Branch			
Syntax:	[ <i>label</i> ] BRA <i>n</i>			
Operands:	$-1024 \leq n \leq 1023$			
Operation:	$(PC) + 2 + 2n \rightarrow PC$			
Status Affected:	None			
Encoding:	1101	0nnn	nnnn	nnnn
Description:	Add the 2's complement number '2n' to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC+2+2n. This instruction is a two-cycle instruction.			
Words:	1			
Cycles:	2			
Q Cycle Activity:				
Q1	Q2	Q3	Q4	
Decode	Read literal 'n'	Process Data	Write to PC	
No operation	No operation	No operation	No operation	

# Timing requirement for conditional branching instructions (e.g., bz)

- If **No Jump**,
  - Sequential running of the program is not disrupted
  - *1 instruction cycle needed*
- If **Jump**,
  - Instruction is fetched incorrectly and must be dumped.
  - Nothing can be executed in the second cycle.
  - *2 instruction cycles needed*

BZ	Branch If Zero				
Syntax:	[ label ] BZ n				
Operands:	-128 ≤ n ≤ 127				
Operation:	if Zero bit is '1' (PC) + 2 + 2n → PC				
Status Affected:	None				
Encoding:	<table><tr><td>1110</td><td>0000</td><td>nnnn</td><td>nnnn</td></tr></table>	1110	0000	nnnn	nnnn
1110	0000	nnnn	nnnn		
Description:	If the Zero bit is '1', then the program will branch. The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC+2+2n. This instruction is then a two-cycle instruction.				
Words:	1				
Cycles:	1(2)				

Q Cycle Activity:

If Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	Write to PC
No operation	No operation	No operation	No operation

If No Jump:

Q1	Q2	Q3	Q4
Decode	Read literal 'n'	Process Data	No operation

# Timing requirement for conditional skipping instructions (e.g., decfsz)

- If **No Jump**,
  - 1 instruction cycle needed
- If **Jump followed by a 1-word (2-byte) instruction**,
  - 2 instruction cycles needed
- If **Jump followed by a 2-word (4-byte) instruction (e.g., goto)**,
  - 3 instruction cycles needed

DECFSZ	Decrement f, Skip if 0			
Syntax:	DECFSZ f {,d {,a}}			
Operands:	$0 \leq f \leq 255$ $d \in [0,1]$ $a \in [0,1]$			
Operation:	$(f) - 1 \rightarrow \text{dest}$ , skip if result = 0			
Status Affected:	None			
Encoding:	0010	11da	ffff	ffff

Cycles: 1(2)  
 Note: 3 cycles if skip and followed by a 2-word instruction.

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register f	Process Data	Write to destination
If skip:			
Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
If skip and followed by 2-word instruction:			
Q1	Q2	Q3	Q4
No operation	No operation	No operation	No operation
No operation	No operation	No operation	No operation

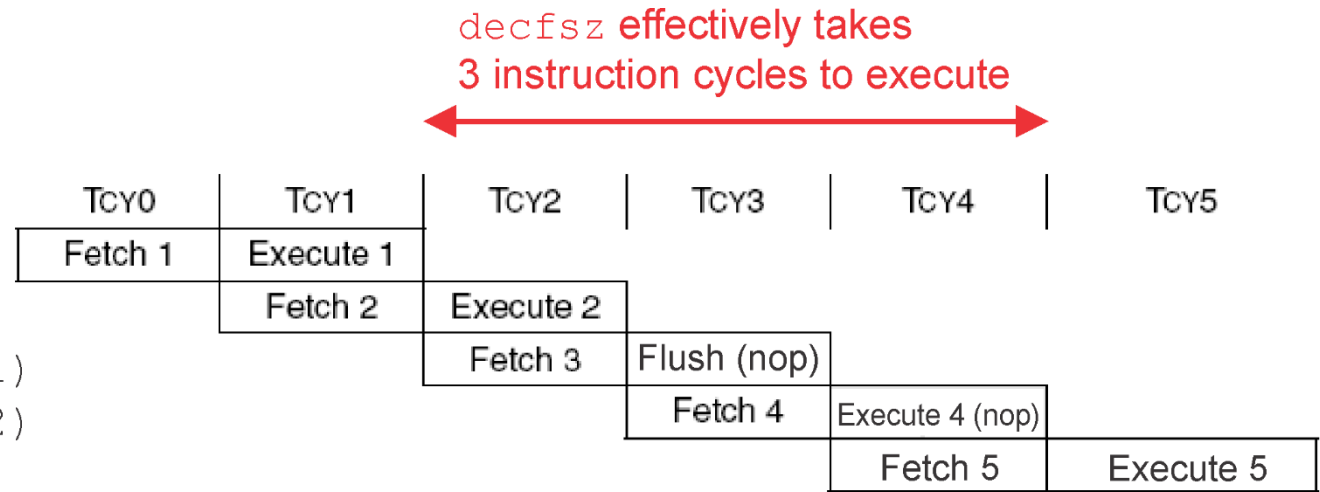
# Jump followed by a 2-word instruction

1. Instruction 1
2. `decfsz COUNT`
3. `goto Continue (1)`
4. `goto Continue (2)` ← if Skip
5. Instruction 5

- If `decfsz` skips, PC jumps to Instruction 4, which is the second word of the 4-byte (or 2-word) instruction. This jump takes 2 instruction cycles.
- Opcode with pattern `1111 xxxx xxxx xxxx` is treated as a `nop`. The second word of the `goto` instruction (in fact, all 2-word instruction) is of this pattern. This `nop` takes an extra 1 instruction cycle.
- In total, `decfsz` takes 3 instruction cycles to execute in this case.

# Jump followed by a 2-word instruction

1. Instruction 1
2. `decfsz COUNT`
3. `goto Continue (1)`
4. `goto Continue (2)`
5. Instruction 5



# Timing Requirement for branching/skipping instructions

Type	Instruction Cycles Requirement
Unconditional branching	2 always
Conditional branching	1 if not jump 2 if jump
Conditional skipping	1 if not skip 2 or 3 if skip: <ul style="list-style-type: none"><li>• 2 if the skipping instruction is followed by a 1-word instruction</li><li>• 3 if followed by a 2-word instruction</li></ul>



# Motivations of Creating Time Delays

- Microcontroller is a fast device. We want to put the microcontroller in a waiting state in some situations, e.g.,
  - Testing: For example, to visualize the 8-bit LED module is actually counting up, we have to implement delays between each increment.
  - Peripherals usually would not respond as quickly as the microcontroller. The microcontroller must be put on hold to give the peripheral enough time to respond.

# Using Program Loops to Create Time Delays

Consider the following code fragment:

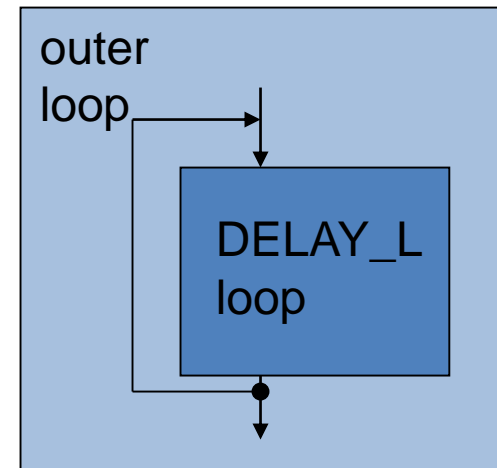
```
DelayLoop: decfsz DELAY_L  
           bra DelayLoop
```

	[DELAY_L]
1 <sup>st</sup> repetition	00 → FF
2 <sup>nd</sup> repetition	FF → FE
⋮	⋮
256 <sup>th</sup> repetition	01 → 00

← Skip out of loop

- The loop can be repeated for a maximum of 256 times. How do we get a longer delay?

Solution: Looping within a loop or *Nested Loop*



# Using Program Loops to Create Time Delays

```
DelayLoop: decfsz    DELAY_L  
           bra DelayLoop  
           decfsz    DELAY_H  
           bra DelayLoop
```

- *Notation [DELAY\_L] = value contained in register with address DELAY\_L.*
- e.g., [DELAY\_L] = 0x00, [DELAY\_H] = 0x02
- The first decfsz instruction will execute for 256 times.
- [DELAY\_L] will become 0 after the 256<sup>th</sup> decrement. The bra line will be skipped.
- [DELAY\_H] will be decremented to 1. Not zero. Branch back to the start.
- The first decfsz instruction will execute for 256 times.
- [DELAY\_H] will be decremented to 0. End program.
- In total, the first decfsz instruction executes 2x256 times.


# Time delay generated by the nested loop

- Initialize [DELAY\_L] = 0 and [DELAY\_H] = 2.

**DelayLoop:**

<code>decfsz DELAY_L</code>	1 instruction cycle (2 if skip)
<code>bra DelayLoop</code>	2 instruction cycles
<code>decfsz DELAY_H</code>	1 instruction cycle (2 if skip)
<code>bra DelayLoop</code>	2 instruction cycles

Total number of instruction cycles needed to run the *DELAY\_L* loop (i.e., the loop formed by the first two lines)

	[DELAY_L]	Instr. cycles	 Not skipping in the first 255 repetitions
1 <sup>st</sup> repetition	00 → FF	3	
2 <sup>nd</sup> repetition	FF → FE	3	
⋮	⋮		
256 <sup>th</sup> repetition	01 → 00	2 (decfsz skips)	
Total		255*3+2 = 767	

# Time delay generated by the nested loop

Sequence of events	Instr. cycles
DELAY_L loop executes for the <u>1<sup>st</sup> time</u>	767
Executes the last two lines with [DELAY_H] decrementing from 02 to 01 (decfsz not skipping)	3
DELAY_L loop executes for the <u>2<sup>nd</sup> time</u>	767
Executes the last two lines with [DELAY_H] decrementing from 01 to 00 (decfsz skips)	2
Total	1539

- How many times DELAY\_L loop will be executes depends on how you initialize [DELAY\_H] (e.g., [DELAY\_H] = 2, DELAY\_L loop executes for 2 times here)
- If clock frequency = 4MHz (instruction freq. = 1MHz and each instruction cycle takes 1 $\mu$ s), the whole loop lasts 1.539 msec.

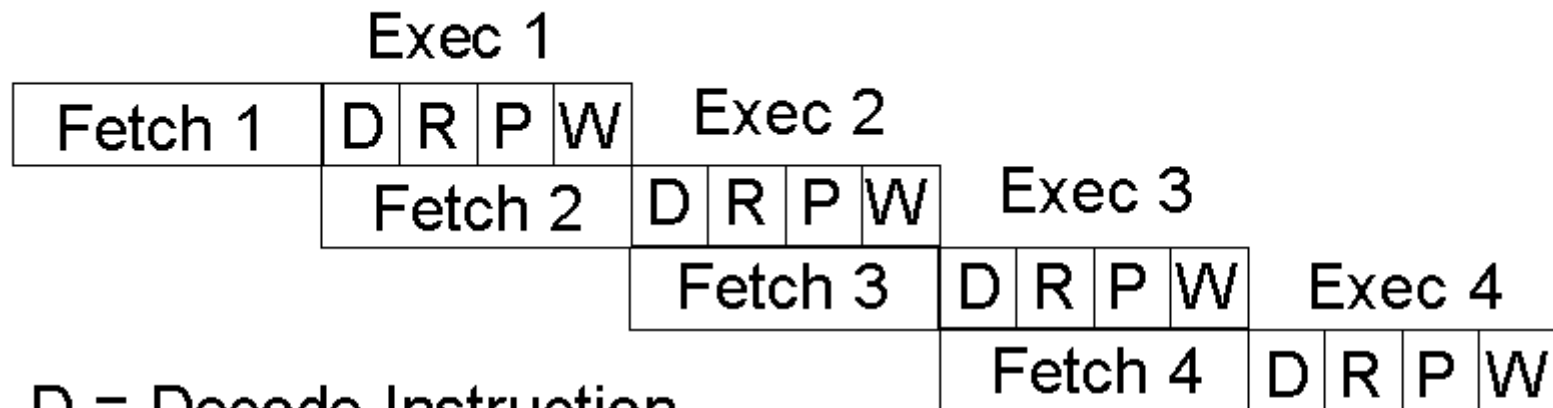
# Using Program Loops to Create Time Delays

- The PIC18 uses a crystal oscillator to generate the clock signal needed to control its operation.
- The instruction execution time is measured by using the instruction cycle clock.
- 1 instruction cycle = 4 clock cycles (i.e.,  
Instruction cycle frequency = Clock cycle frequency divided by 4)
- A desired time delay is created by repeating a certain set of instructions using loops.

# PIC Multistage Execution Pipeline

- Execution takes 4 clock periods of the oscillator.
- e.g., Clock cycle freq. = 4MHz; Instruction cycle freq. = Clock cycle freq./4 = 1MHz.
- Q1 – Decode the instruction that has been fetched
- Q2 – Operand is fetched from the file register
- Q3 – The operation is performed
- Q4 – The result is written in the destination register

# PIC Multistage Execution Pipeline



D = Decode Instruction

R = Read Operand

P = Process Instruction

W = Write the results to the destination register



# You should be able to .....

- Know the 3 components of a loop. Code PIC assembly language instructions to create loops.
- Know how a loop is terminated by Strategy 1 (conditional skipping + unconditional branching) and Strategy 2 (conditional branching).
- Explain how status bits affect the behaviour of conditional branching instructions.
- Compare the properties of conditional and unconditional branching instructions.
- Calculate the target addresses for conditional/unconditional branching instructions.
- Know the timing requirements for 3 types of instructions that change the PC: conditional/unconditional branching and conditional skipping instructions.
- Calculate time required to execute a fragment of assembly code.
- Explain in what situation *nested loop* is required in PIC programming.