University of Jordan
School of Engineering
Department of Mechatronics Engineering
Microprocessor and Microcontroller Laboratory
0908432
Exp. 5: Implementing Instructions (I)

Objectives

1. To be familiar with assembly language programming and the Microchip PIC 16 series instruction set.
2. To see an application of macros and methods of utilizing them.
3. To use the debugging facility of the MPLAB IDE to fix program bugs.

Pre-lab Preparation:

1. Read chapter 7 of the PIC16F84 data sheet.
2. Review the Status Register- Section 2.2.1 in the book
3. Study the assembly code listings of accompanying programs. (Very important).

Example1: - Counting the Number of Ones in a Register's **Lower Nibble** Introducing simple conditional statements.

```
1.  include "p16f84a.inc"
2.  cblock 0x20
3.  testNum             ;GPR @ location 20
4.  tempNum             ;GPR @ location 21
5.  endc
6.  cblock 0x30
7.  numOfOnes           ;GPR @ location 30
8.  endc
9.  org 0x00
10. clrf    numOfOnes       ;Initially number of ones is 0
11. movf    testNum, W      ;Since we only need to test the number of ones in the lower nibble, we mask
                            ;them by 0x0F (preserve lower nibble and discard higher nibble)
12. andlw   0x0F            ;in case a user enters a number in the upper digit. Save masked result
13. movwf   tempNum          ;in tempNum
14. rrf     tempNum, F      ;rotate tempNum to the right through carry, that is the least significant bit of
                            ;tempNum (bit0) goes into the C flag of the STATUS register, while the old
                            ; value of C flag goes into bit 7 of tempNum
15. btfsc   STATUS, C       ;tests the C flag, if it has the value of 1, increment number of ones and
16. incf    numOfOnes, F    ;proceed, else proceed without incrementing
17. rrf     tempNum, F
18. btfsc   STATUS, C       ;Same as above
19. incf    numOfOnes, F
20. rrf     tempNum, F
21. btfsc   STATUS, C
22. incf    numOfOnes, F
23. rrf     tempNum, F
24. btfsc   STATUS, C
25. incf    numOfOnes, F
26. nop
27. end
```

As you can see in the above program, we did not write instructions to load **testNum** with an initial value to test; this code is general and can take any input. So, **how do you test this program with general input?**

After building your project, adding variables to the watch window, and selecting MPLAB SIM simulation tool, simply **double click on testNum** in the **watch** window and fill in the value you want. Then Run the program.

Change the value of **testNum** and re-run the program again, check if **numOfOnes** hold the correct value.

Coding for efficiency: The repetition structures

You have observed in the code above that instructions from **14 to 25** are simply the same instructions repeated over and over four times for each bit tested.

Now we will introduce the repetition structures, similar in function to the "for" and "while" loops you have learnt in high level languages.

```
include "p16f84a.inc"
        cblock 0x20
                testNum
                tempNum
        endc
        cblock 0x30
                numOfOnes
                counter                 ;since repetition structures require a counter, one is declared
        endc
        org 0x00
        clrf    numOfOnes
        movlw 0x04                      ;counter is initialized by 4, the number of the bits to be tested
        movwf counter
        movf    testNum, W
        andlw   0x0F
        movwf tempNum
Again
        rrf     tempNum, F
        btfsc   STATUS, C
        incf    numOfOnes, F
        decfsz  counter, F              ; after each test the counter is decremented,
          goto      Again               ; if the counter reaches 0, it will skip to "nop" and program ends
        nop                             ; if the counter is > 0, it will repeat "goto Again"
        end
```

Modular Programming

Modular programming is a software design technique in which the software is divided into several separate parts, where each part accomplishes a certain independent function. This "Divide and Conquer" approach allows for easier program development, debugging as well as easier future maintenance and upgrade.

Modular programming is like writing C++ or Java **functions**, where you can use the function many times only differing in the parameters. Two structures which are like functions are **Macros** and **Subroutines** **which are used to implement modular programming.**

1)Subroutines

Subroutines are the closest equivalent to functions

* Subroutines start with a **Label (**subroutine_Name**)** giving them a name and end with the instruction **return.**

*Subroutines can be written anywhere in the program after the **org** and before the **end** directives.

*Subroutines are used in the following way: **Call** subroutine_Name.

*Subroutines are stored once in the program memory, each time they are used, they are executed from that location.

*Subroutines alter the flow of the program; thus they affect the **stack**.

2)Macros

Macros are declared in the following way (like the declaration of cblocks)

```
Macro       macro Name
Instruction 1
Instruction 2
.
.
Instruction n
endm
```

*Macros should be declared **before** writing the code instructions. **It is not recommended to declare macros in the middle of your program.**

*Macros are used by only writing their name: macro Name

*Each time you use a macro, it will be replaced by its body. Therefore, the program will execute sequentially, the flow of the program will not change. **The Stack is not affected**.

Example2: -

```
; ----------------------------------------------------------
; General Purpose RAM Assignments
; ----------------------------------------------------------
cblock          0x17
InputM2
Input_TempM2
InputM4
ResultM2
Result_TempM2
ResultM4
Endc
```

```
; ----------------------------------------------------------
; Macro Definitions
; ----------------------------------------------------------
Multiply2       macro
Movf            Input_TempM2,w
Addwf           Input_TempM2,w
movwf           Result_TempM2
Endm
; ----------------------------------------------------------
; Vector definition
; ----------------------------------------------------------
                        org 0x000
                        nop
                        goto Main

INT_Routine     org 0x004
                goto INT_Routine
; ----------------------------------------------------------
; The main Program
; ----------------------------------------------------------
Main            Movlw           d'15'
Movwf           InputM2
Movwf           InputM4
Movwf           Input_TempM2
Multiply2
movwf           ResultM2
Movwf           Input_TempM2
Call            Multiply4
Goto            finish
; ----------------------------------------------------------
; Sub Routine Definitions
; ----------------------------------------------------------
Multiply4
Multiply2
Movf            Result_TempM2,w
Movwf           ResultM4
Return
finish
                 nop
end
```

General Multiply function: -Result = Input1 * Input2

```
General_Multiply
                Clrf        Result
Again           movf        Input2,w
                Addwf       Result,f
                Decfsz      Input1, f
                Goto        Again
Finish
        Return
```

Exercise1:

Modify **Example2** to multiply by 3 Macro(Multiply3) and multiply by 9(Multiply9) function?

Exercise2:

Write a test code for General Multiplication function for two following cases: -

1- Input1=d'9', Input2=d'7'

2-Input1=0, Input2=15?

Discussion and Follow-up

Write a General Divide function that uses multiple subtract to perform following equation: -

Result = Counter + Reminder

Where; 0 < **Reminder** < Input1

Counter is integer number of **Input2 / Input1**